FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Molekuel-Rendering in Unity3D

mit Hilfe von Stylesheets

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Lukas Prost
Matrikelnummer 1225511

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Reinhold Preiner

Wien, 5. September 2016

Lukas Prost	Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Molecule Rendering in Unity3D

Using Stylesheets

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Lukas Prost
Registration Number 1225511

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Reinhold Preiner

Vienna, 5th September, 2016

Lukas Prost

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Lukas Prost
Taborstrasse 22/2/34, 1020 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. September 2016

Lukas Prost

Danksagung

Ich würde mich gerne bei meinem Supervisor Reinhold Preiner dafür bedanken, dass er mich so sehr bei meiner Arbeit unterstützt hat und dafür, dass er mich mit meiner Arbeit zur CESCG gebracht hat, was ein besonderes Erlebnis für mich war.

Außerdem möchte ich mich bei Philipp Wissgott und Klemens Senn wie auch bei Matthias Maschek für die Unterstützung im praktischen Teil meiner Arbeit bedanken.

Weiters danke an Michael Wimmer für die Unterstützung.

Ein großer Dank geht ebenfalls an meine Familie für die ganze Unterstützung während des Studiums und der Arbeit. Ohne sie hätte ich es nicht geschafft, diese Arbeit abzuschließen.

Acknowledgements

I want to thank my supervisor Reinhold Preiner for his ordinary support and for bringing me to CESCG with my thesis, which was a great experience for me.

Moreover, I want to thank Philipp Wissgott, Klemens Senn as well as Matthias Maschek for their support during the practical part of my thesis.

Also, I would like to hank Michael Wimmer for his support.

Lastly, I want to thank my family for their support. Without them, my studies and therefore this thesis would not have been possible.

Kurzfassung

Aufgrund der immer weiteren Verbreitung von Smartphones werden sie immer häufiger in den Unterricht eingebunden. Neben der Verwendung für Recherche Tätigkeiten können sie dabei auch für komplexere Aufgaben verwendet werden, wie z.B. zur Betrachtung von Molekülen im Chemie Unterricht.

Der Stil, mit welchem die Moleküle gerendert werden sollen, ist dabei oft schwer festzulegen und variiert abhängig von der Altersgruppe. Noch schwieriger wird es, wenn nicht nur für Schüler, sondern auch für z.B. Wissenschaftler ein ansprechender Stil gefunden werden muss. In solchen Fällen sind oft verschiedene Designs erforderlich. Diese zu erstellen liegt in der Regel an Designern. Jenen mangelt es jedoch häufig an den ausreichenden Programmierkenntnissen, um ihre Designs umzusetzen, was dazu führt, dass sie einen Entwickler für die Umsetzung und Bearbeitung ihrer Designs benötigen. Selbst wenn es sich dabei nur um kleine Arbeiten handelt. In dieser Arbeit präsentieren wir ein konfigurierbares System für Rendering Effekte entwickelt mit Unity3D, welches die visuelle Gestaltung von Molekülen via eines JSON Stylesheets erlaubt. Programmierkenntnisse werden dabei keine vorausgesetzt.

Wir zeigen die technische Umsetzung verschiedener Rendering Effekte für mobile Geräte. Das Resultat demonstrieren wir anhand einer kommerziellen Molekül Visualisierungs App, in die wir unser System als Erweiterung eingebaut haben. Dabei erstellen wir mit unserem System verschiedene Rendering Stile für Moleküle in der App, die für Schüler, aber auch für Wissenschaftler und Marketing, ansprechend sind.

Abstract

Due to their omnipresence and ease of use, smart phones are getting more and more utilized as educational instruments for different subjects, for example, visualizing molecules in a chemistry class.

In domain-specific mobile visualization applications, the choice of the ideal visualization technique of molecules can vary based on the background and age of the target group, and mostly depends on the choice of a graphical designer. Designers, however, rarely have sufficient programming skills and require an engineer even for the slightest adjustment in the required visual appearance. In this thesis we present a configuration system for rendering effects implemented in Unity3D, that allows to define the visual appearance of a molecule in a JSON file without the need of programming knowledge.

We discuss the technical realization of different rendering effects on a mobile platform, and demonstrate our system and its versatility on a commercial chemistry visualization app, creating different visual styles for molecule renderings that are appealing to students as well as scientists and advertisement.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

During the last decade mobile devices have seen an unrivalled rise in popularity. From the bulky and cumbersome early smartphones, mobile technology nowadays is the de facto leader in terms of customer electronics. This technological leadership is accompanied with revolutionary miniaturization and, consequently, strong computation power and high resolution screens. Furthermore, their scope of application has widened such that nowadays they can be also used for complex renderings and visualizations in real time.

Apart from pure mobile gaming, smartphones have also become important for educational and scientific purposes recently. Here, interactive 3D visualization can help tremendously in terms of understanding and presentation. For example, the 3D visualization of molecules can be useful for many different groups e.g. scientists and students for analysing or learning about molecules and their structure. Enabling these groups to learn and work on a mobile device, simplifies the access to such information enormously.

By being so useful for different groups, the target audience for mobile molecule visualization is very diverse. A single visual style can be hardly appealing to cover all the bases of the audience. E.g. a plain rendering of molecules may be ideal for scientists, but may be too boring for students. But a rendering that looks great for students may be too cluttered for scientists. Hence, optimized styles are required for the different user groups.

Often it is up to a designer to create the visual appearance. And in the case stated before, he even has to design many of them. Designers, however, rarely have the technical skills to realize their design in a graphical rendering framework on their own, resulting in the need for an engineer. And for every design the designer creates, the engineer is needed to implement the development.

Moreover, the first drafts of a designer are often not final. Yet they need to be implemented and evaluated. And every time a slight adjustment is made, the engineer is needed to do the same work of building and deploying the application with a new style. And even after the application is released, updates to the style are still possible. This results in the need of at least two people to maintain an application's update life cycle.

In this thesis, we present an extension for the mobile molecule visualization app *Waltzing Atoms* implemented in Unity3D, that allows to easily modify the visual appearance of molecules with the help of JSON stylesheets. Designers can change the rendering style by setting parameters in these stylesheets e.g. which shaders to use or where lights should be placed with no required programming skill whatsoever. In the following chapters we will elaborate on this extension and how the shaders available in the stylesheet work. We will explain how to apply high quality rendering effects like screen space ambient occlusion, depth of field, comic shading and outline rendering. Those effects were chosen based on the three target groups of the app, namely students, scientists and companies who want to use it for advertisement.

The rest of this thesis is structured as follows:

The first chapter *Background and Related Work* will give some background information on molecular visualization e.g. which different types of visualization are used for molecules. Moreover, the techniques used to achieve the different styles will be presented with an overview of common methods. Because the app was developed with the *Unity3D* game engine, the implemented extension will also be developed in the *Unity3D* framework. Therefore a section describing the *Unity3D* engine is enclosed.

The second chapter, *Applying Different Rendering Styles in an Mobile Molecule Visualization App*, will explain the implementation in more detail. Firstly, the extension processing and applying the stylesheets will be presented. Then, the rendering techniques implemented for the different styles are described in detail (*Screen Space Ambient Occlusion*, *Comic Shading*, *Outline Rendering* and *Depth of Field*).

The third chapter *Results and Evaluation* presents the final styles for named target groups. They will be benchmarked to the standard appearance which is given by the app.

Finally, there will be an conclusion and an outlook for future work.

CHAPTER 2

Background and Related Work

2.1 Overview

To allow a designer to create various styles for the different target groups using a stylesheet, a toolset of rendering techniques is required. *Comic shading* and *outline rendering* should make molecules look more like a cartoon, making them appealing to students and younger users, whereas more realistic rendering techniques like *depth of field* and *ambient occlusion* are provided for advertisement styles. For scientists, no special rendering techniques are required, since a plain style not hiding any information is desired.

Before those techniques will be presented below, an overview over molecular visualization, related software and the game engine Unity is given.

2.2 Molecule Visualization and Related Software

Molecules are a group of two or more atoms. They can have complex structures with different properties depending on their composition of the atoms and the relation between them.

The aim of molecular visualization is to show their structure and make their properties visible. Chemists and pharmacists can use those visualizations to design new molecules. Molecular visualization, however, has the problem that a realistic representation of molecules does not exist in the real world because atoms are smaller than the wave length of light. Therefore, metaphors are used for the sake of understanding. Defining such metaphors is the main research issue of molecule visualization. [Goo05]

Precursors of molecule visualization appeared in the 17th century as hand drawn pictures and go back to Kepler (1611) and Huygens (1690). They used spheres to show the structure of crystals by arranging them in a three dimensional array. Other precursors were Frankenheim (1842) and Bravais (1848) who used a ball-and-spoke representation. [Smi60] Van der Waal (1873) used a representation that took the "size" of atoms into

account by using spheres for atoms with the "size" of the atoms encoded in their radius. [KKL+15]

Nowadays three metaphors are mainly used for molecular visualization: bond diagrams, ribbon diagrams and space-filling diagrams. Those are shown in Figure 2.1. Combinations of these three are also common. [Goo05]

Bond diagrams are used to show covalent bonds. It is a bond-centric model, discarding electrons to focus on atom pairs and the structure of the molecule. [KKL+15] A common form of this model is the ball-and-stick representation. Atoms are represented as spheres, whereas their connections are represented as sticks. [Goo05]

Space-filling diagrams are use to display the properties of electrons. Spheres are placed at the positions of the atoms, with a radius corresponding to the distance between atoms. [Goo05] This metaphor is one of the most often used visualization methods. [KKL+15]

Ribbon diagrams are mainly used to present bio molecules. A ribbon diagram ignores single atoms and focuses on their structure instead. Mainly on chains and their orientation to show molecule folding. [Goo05] In contrast to bond and space-filling diagrams, ribbon diagrams are abstract. They focus on showing information that is not clearly visible in models which show single atoms. [KKL+15]

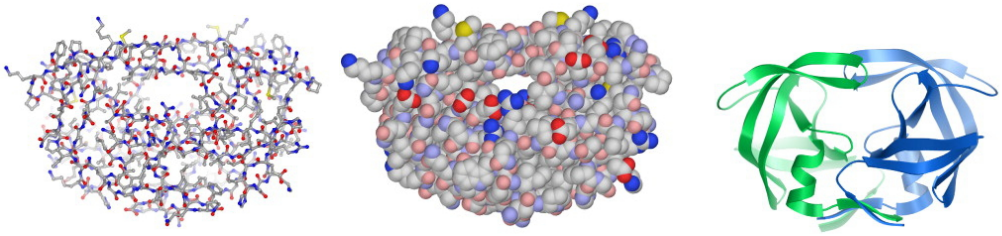


Figure 2.1: A HIV protease rendered using three different visual metaphors. FLTR: Bond diagram, Space-filling diagram and Ribbon diagram. Image taken from [Goo05].

Nowadays, many mobile applications are available to visualize molecules. To name some: *Molecules*, *Atomdroid*, *Molecular Viewer 3D* and *NDKMol* as well as *ESMol*. All these apps can visualize molecules and some, e.g. *Molecules*, can download molecular data directly from PubChem and Protein Data Bank. These apps, however, are not as flexible as programs running on computers, meaning that they often provide only one or two metaphors for the user to choose from. *Atomdroid* renders molecules using the ball-and-stick metaphor, whereas *NDKMol* and *ESMol* are able to convert polymer structures into ribbon diagrams. [LH13]

Another app for molecular visualization is presented by Quinn et al. [QBC+15]. *RCSB PDB Mobile* is designed for using it as an access point for the RCSB Protein Data

Bank. Molecules can be searched, downloaded and visualized. The visualization is based on *NDKMol*, which is open source.

Waltzing Atoms, the app we developed the extension for, uses Bond diagrams. Elements are encoded in color and the radii of the spheres which represent the atoms. The use of this metaphor is determined by the planned use of the application. Besides just visualizing molecules, the app also allows to render molecules with missing atoms. To exercise and learn, students can fill this holes with atoms like in a game. That excludes ribbon diagrams as a possible model because they do not show single atoms. Space-filling diagrams could be used. However, the ball-and-stick model seems to be more suitable because sticks without a sphere at the end indicate missing atoms very intuitive, whereas showing such information in space-filling models would be more difficult.

2.3 Unity3D

Unity3D (Unity) is a general-purpose game engine. It can be used for creating games of all genres but also to build scientific and business applications. An important feature of Unity is its ability to build and deploy software for many different platforms including Android, iOS, Windows Phone and consoles. (All of them with a single code base.) Overall, Unity supports 23 different platforms. In comparison, the *Unreal Engine* supports nine platforms [uef] and *Cryengine*, another game engine, only six [cry]. The broad support of different platforms and the good documentation are the reasons why Unity was used for the app.

A Unity project consists of different *scenes* which can be thought of as levels in a game. An object appearing in the scene is called a *Game Object* (GO). A GO is a container for components and the type of a GO is defined by the combination of those. Some examples for components are: Transform, Mesh, Light, Camera and so on. Each component has its own parameters and can interact with other components of the same GO. Such components are provided by the engine or can be written by the developer as scripts.

2.3.1 Rendering in Unity

Some of the platforms supported in Unity have dedicated graphic cards while others have SoCs with an integrated graphics unit. Certain platforms may support only OpenGL, whereas others depend on Direct3D. Considering this variety, Unity provides its own shader language *ShaderLab*, which is syntactically similar to CgFX and the Direct3D shader language. Shaders written in *ShaderLab* are transpiled and compiled by Unity to shaders usable by the target platform.

A *ShaderLab* shader consists of sections. E.g. the first one is a property section, defining the data available in the shader like colors or textures. This section is then followed by a list of subshader sections. Each subshader is, contrary to its name, a complete shader. This gives developers the option to write the same shadings techniques with different complexities. For example the first subshader calculates complex light

simulations using modern graphics API, whereas the second subshader just approximates the functionality using simpler methods. As soon as Unity has to render a mesh using this shader, it takes the first subshader that does not use API calls or functionality that are not supported by the user's device and uses it for rendering. If none of the given subshaders is supported, a fallback shader (often a standard Unity shader) is used.

A subshader itself has a list of tags which support some basic settings like enabling z-buffer writing or backface culling. After the tags pass sections are defined, which can be seen as normal rendering passes.

Shaderlab supports three abstraction levels. [unia] These abstraction levels are not bound to a whole subshader, but can vary from pass to pass:

- Fixed Function Shaders

  These are the most abstract shaders. They can only be used for simple effects and allow little to none customization.

- Surface Shaders

  A surface shader is less abstract than a fixed function shader. In a surface shader, the developer writes a method which defines the lighting and a *surf* function. The *surf* function receives the needed input parameters (Unity determines them by analyzing the code) like the UV coordinate of the currently processed fragment or light directions, does some transformations/calculations with it and saves it in an output structure. The values therein can then be used in the lighting method. Unity, however, already provides basic lighting methods simplifying shader writing with surface shaders.

- Vertex and Fragment Shaders

  These shaders are written by creating a vertex and a fragment function that are transpiled to normal vertex and a fragment shader. Shaders written that way are highly customizable. Compared to Surface Shaders, however, more code has to be written.

To assign a shader to an object, it needs to be combined with a material component. A material in Unity defines how an object is rendered. Shaders are attached to materials and materials are attached to GOs.

2.4 Ambient Occlusion

Ambient occlusion (AO) adds shadows to points on a surface that are more or less occluded by other points like crevices or valleys, since they do not receive as much light as a surfaces directly exposed to a light source. An example is shown in Figure 2.2. AO is a part of *Ambient Environment*, which is a substitute for "bounce" and "fill" lights which are used to give more realism to a scene by adding shadows and highlights based on the form and the environment of an object. Instead of using fill light that would be

Figure 2.2: A scene with *ambient occlusion*, where the AO term is set as color. Image taken from [PG04]

needed to be placed by hand, an *environmental map* (EM) is used for defining them. Landis [Lan02] describes an AO algorithm to determine the darkening of different surface points. Many rays are casted in a hemisphere around each point. The amount of rays that collide with a point or a surface is then proportional to the occlusion factor.

The directions of the rays that do not collide are then used to calculate the average direction from where the available light is coming from. The new calculated direction is called *bent vector* which is used as the new lookup direction in the EM.

Figure 2.3 shows an AO calculation in 2D for a point P. Five rays are casted, from which231 three collide with nearby geometry. The other two are used to calculate the *bent vector* B.

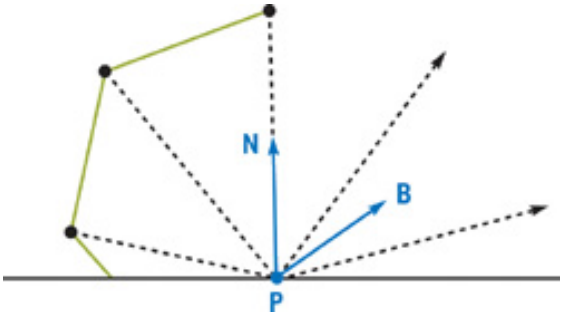


Figure 2.3: The calculation for the occlusion factor and average light direction. Image taken from [PG04]

The example in Figure 2.2 shows the darkening of the crevices. The influence of the *bent vector* can be seen as a darkening at the bottom of the model. The image is rendered by Pharr and Green [PG04] and the used algorithm is based on the description of Landis.

The method described by Landis (casting rays for each surface point) was developed during the production of the movie *Pearl Harbor*. The approach is, however, not suitable for real time rendering. Scenes can become complex and ray casting is very expensive.

Moreover, it needs up to 256 rays to calculate a good AO effect [Chr03]. This cannot be done in real time with the currently available hardware.

An approach that avoids ray casting is given by Bunnell [Bun05]. His method interprets a mesh as a set of surface elements which can emit, transmit and reflect light. Such surface elements are disks, created for each vertex and orientated along the normal vector of the vertex. They are used to calculate the occlusion factor by summing up the amount off occlusion of all individual disks above the point the occlusion is calculated for.

Hoberock [Hob07] provides modifications for this algorithm which reduces artifacts. Some examples for other solutions for AO are given by Evans [Eva06], Hegeman et al. [HPAD06] and Ren et al. [RWS+06].

All of these AO algorithms are working in the object-space and have according to Engel [Eng09] one or more of the following problems:

- Expensive Preprocessing

  E.g. AO algorithms using ray casting have to do this more than a hundred times for each triangle.

- Dependency of the scene complexity

  Since the occlusion factor must be calculated for every point of a model, it becomes more expensive to calculate AO for more geometry.

- Difference in preprocessing of static and dynamic geometry

  If a model is static, AO can be precomputed as ambient lighting because it does not change. Therefore, more complex and realistic methods can be used than on models which are dynamic and need to be processed every frame.

- Complexity

  The implementation of an AO algorithm usually is a complicated and time consuming procedure.

In Contrast to the object-space AO methods presented above, screen space ambient occlusion (SSAO) operates in the screen space. This has many advantages. Firstly, no preprocessing is required. (The algorithm calculates AO for the visible part of the scene in real time.) Secondly, the computing effort is independent of the geometry in the scene. SSAO has to process a constant amount of data every frame that is only defined by the size of the screen space. Thirdly, the geometry does not need to be stored in special data structures as in most object-space AO algorithms.

SSAO was developed by Vladimir Kajalin in 2007 and first used in *Crytec*'s game *Crysis*. Mittring [Mit07] describes the method. SSAO simulates occlusion from nearby surfaces by using the depth buffer to approximately reconstruct local geometry. For the occlusion determination, random samples are placed around each fragment's view space position which are then compared against the depth of the surrounding geometry using

depth buffer lookups. The more samples are covered by the surrounding geometry, the more the fragment is occluded. Filion and McNaughton [FM08] describe an improved version of Mittring [Mit07] by aligning the samples on a hemisphere around the surface normal reducing self occlusion dramatically.

Méndez [Mén10] presents an even more primitive approach. Instead of comparing samples with the depth buffer, he simply calculates the occlusion factor for a pixel based on its distance and angle to its sample pixels randomly chosen from the sample area.

2.5 Comic Shading

Figure 2.4: Comic shading can be seen on the characters in *Ni no Kuni* by *Bandai Namco Games*.

Comic shading (see Figure 2.4 for an example) aims to render objects such that they look like cartoons, which are in general two dimensional with little to no details. To achieve such a style, comic shader (also called toon shader) often use just one or two solid colors without a color gradient. For two colors, Lake et al. [LMHB00] presents the *hard shading* technique. One Color is used for the lighted areas and a second one, usually a darker tone of the first color, is used for surfaces lying in the shadow. This method adds more details to the geometry than a one color approach by adding a sense for the form of the object and its position in the scene. The colors are defined in a texture that has two texels in its simplest form. To determine which color to use for a point on a model, a transition boundary is defined based on the dot product between the light direction and the surface normal. Depending on the position relative to the transition boundary, a point on the model is either rendered with the light or the shadow color, as seen in Figure 2.5). Note that hard shading is not limited to two colors.

Mitchell, Moby and Eng [MFE07] provide a different approach by using a texture map, seen in Figure 2.6, with a value for the whole visible dot product range from 0 to 1.

In contrast to the method provided by Lake et al. [LMHB00], the looked up value is not used as the resulting color of the point on the object, but as a substitution for the *Lambert* term in their lighting equation resulting in an artistic style.

A two dimensional texture map is used by Barla, Thollot and Markosian [BTM06]. Figure 2.7 shows an example for such an texture map.

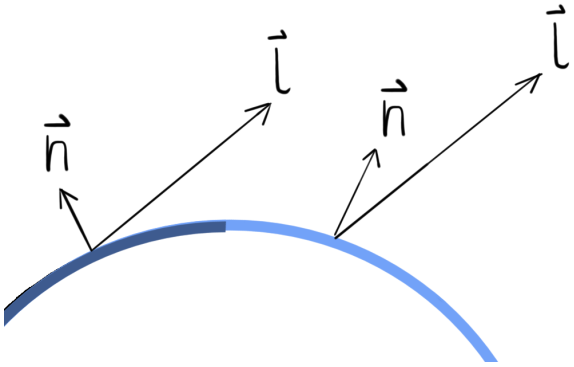


Figure 2.5: Methodology of the hard shading algorithm.

Figure 2.6: Texture map used by *Team Fortress 2* for the light calculation. Image taken from [MFE07]

The horizontal axis is still used for the lookup of the resulting value for a given dot product between the light direction $\hat{l}$ and the surface normal $\hat{n}$. The additional vertical axis is used for the level of detail (LoD) lookup. The D in Figure 2.7 denotes detail. For every level of detail a different one-dimensional texture map is used. Therefore, the two-dimensional texture map can be seen as a stack of one dimensional texture maps. An example for a scene rendered using this algorithm is shown in Figure 2.8.

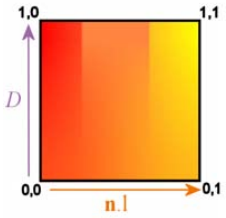


Figure 2.7: A two dimensional texture map. Image taken from [BTM06]

2.6 Outline Rendering

Outline rendering focuses on emphasizing the important edges and silhouettes of objects by rendering them with a defined outline color. An example is shown in Figure 2.9. Since those depend on the position of the observer, they need to be determined every frame. Consequently, the used algorithms need to be fast and efficient. [IFH+03] To provide a better understanding of outlines and support a more fine distinction between different

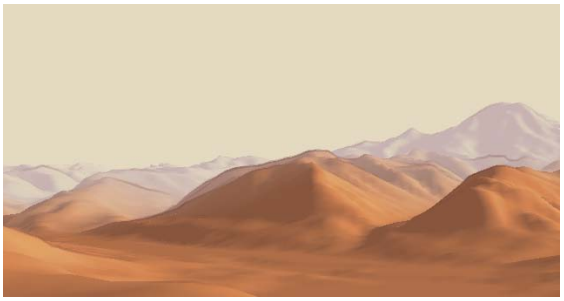


Figure 2.8: A landscape shaded with a 2D texture map and depth-based attribute mapping. Image taken from [BTM06]

Figure 2.9: Outline rendering in *Ōkami* by *Capcom*.

types of outlines, Akenine-Möller, Haines and Hoffman [AMHH08, p.510] provide an overview of types of edges and silhouettes based on the box shown in Figure 2.10:

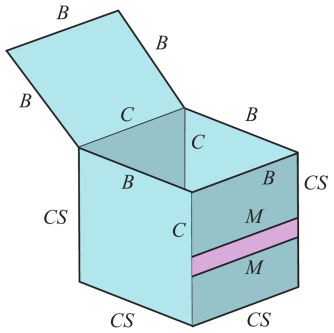


Figure 2.10: An open box with its edges labeled: (B) boundary, (C) crease, (M) material and (S) silhouette. Image taken from [AMHH08]

- Boundary

  Boundary edges, also called border edges, exist only on non-closed polygons. A solid object usually does not have these. [IFH+03] [AMHH08, p.510]

- Crease

  Creases can be defined as edges where both adjacent polygons confine an angle, also called *dihedral angle* [AMHH08, p.510], greater than a predefined threshold. [IFH+03] Creases are also called hard edges or feature edges. [AMHH08, p.510].

- Material

  These are lines between to polygons that do not fulfill the other given edge definitions. They are no real edges but are still important. Also called *self-intersection lines* [IFH+03], they can be lines between to polygons with different materials or different colors. Also, it can be a designer's choice to place them somewhere on the model. [AMHH08, p.511]

- Silhouette

  Isenberg et al. [IFH+03] suggest the following definition: ".. we define the silhouette edges of a polygonal model as edges in the mesh that share a front- and a back-facing polygon". Another way to define a silhouette is given by Akenine-Möller, Haines and Hoffman [AMHH08]. They define it via the dot product between the normal vector of the processed point and the vector defined by the direction to the viewpoint.

A heuristic solution for outline rendering is presented by Akenine-Möller, Haines and Hoffman [AMHH08, p.512]. They render a surface point as an outline if the dot product between the point's normal and the view vector is near zero. As long as the rendered model does not consist of large polygons it returns fine results, but misses creases, material edges or boundaries. If the object has large polygons, the method tends to render big parts of the model as silhouette. Moreover, the thickness of the silhouette is varying, "depending on the curvature of the surface" [AMHH08, p.512]. This method is computationally cheap, making it a good candidate for the use on mobile devices.

Another solution is given by Isenberg et al. [IFH+03]. Using the viewpoint every polygon is categorized either facing forwards or backwards. Then all edges that are shared by a front-facing and a back-facing polygon are rendered as outline. These edges represent the silhouette. Boundaries, ceases and material edges are not detected.

A solution using a special data structure is shown by Buchanan and Sousa [BS00]. They introduce an *edge buffer*: an *a-priori* defined buffer that classifies an edge either as a part of the silhouette or not. To do so, the edge buffer stores at least two bits for every edge. These bits are representing a front facing flag and a back facing flag. If both flags are set, the edge belongs to the silhouette. This buffer has to be updated every frame. Like the method before, this method only detects silhouette edges.

Card and Mitchell [CM02, p.328] describe an algorithm working in image space using the depth values as well as the normal vectors. Those are given in a normal and a

depth buffer, making it easy to implement this method if deferred rendering is used. To detect edges, the gradients of the buffers are determined by convoluting them with an image filter e.g. the Sobel operator. Big normal buffer gradients indicate a strong curvature whereas big depth buffer gradients indicate big depth changes. Since this are properties of edges, points with big gradients are considered as being part of an edge. The edges detected by processing the world-space normal vectors are creases and boundaries, whereas the processing of the depth values detects the silhouette.

The silhouette however, is not guaranteed to be found. For instance, if the differences in the depth buffer are too little, e.g. a sheet of paper lying on a surface, silhouettes may not be detected since the gradient may be to small. On the contrary, if a point is processed that is near an edge, it can be falsely be classified as silhouette point [AMHH08]. Also, there is little possibility to stylize the resulted lines [IFH+03].

Another approach is the "halo" or "shell" method also presented by Akenine-Möller, Haines and Hoffman [AMHH08]. This method renders the object in two passes. The first pass renders the object's back faces with the outline color, where vertices are translated alongside their normal vectors such that the object is enlarged. This creates a hull around the object, which becomes visible as an outline after the projection. In the second render pass, the front faces are rendered normally. "The method is simple to implement, efficient, robust, and gives steady performance" [AMHH08].

2.7 Depth of Field Rendering

Figure 2.11: Depth of field in *Divinity Original Sin Enhanced Edition* by *Larian Studios*.

Depth of field (DoF) is a physical effect created by the refraction of light inside a lens, which causes objects that are in focus to appear sharp on the image plane, whereas objects out of focus to appear blurry. An example is shown in Figure 2.11.

The distance u an object can have to the lens to appear sharp is depending on the lens's *focal length* f (see Figure 2.12). u is called *focal distance* (the distance from the lens where an object is in focus) and can be calculated using the lens Equation 2.1:

$$\frac{1}{u} + \frac{1}{v} = \frac{1}{f} \tag{2.1}$$

v describes the distance between the image plane and the lens [RTI04].

If the distance of an object from the lens equals the focal distance, the lens projects every point of the object 1:1 onto the image plane. If the distance is larger or smaller than the focal distance, its projection onto the image plane appears blurry. The amount of blur is defined by the *circle of confusion* (CoC) (see Figure 2.12). If the radius of the CoC equals 0, the point is projected sharp.

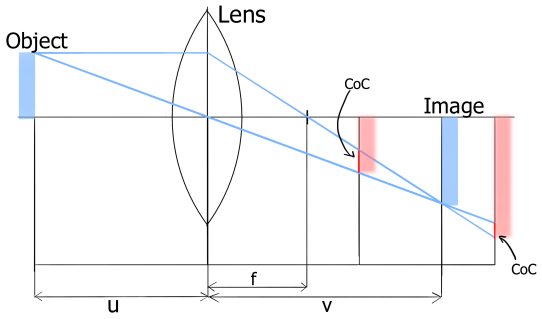


Figure 2.12: Visualization of a lens that projects an object (blue to the left) onto the image plane to the right. The red objects to the right denote possible results of this projection if the object would not be in the focus of the lens. In such a case the object is blurry based on the *circle of confusion* (CoC).

The radius c of the CoC can be calculated with Equation 2.2 using Equations 2.3 and 2.4 [CPC84]. It is calculated for an object with the distance d to the lens. n denotes the aperture number.

$$c = |V_d - V_u| \frac{f}{2nV_d} \tag{2.2}$$

$$V_u = \frac{fu}{u - f} \quad for \quad u > f \tag{2.3}$$

$$V_d = \frac{fd}{d - f} \quad for \quad d > f \tag{2.4}$$

In computer graphics, rendering is done using a projection model that simulates a pinhole camera. A pinhole camera creates images by receiving emitted light rays of the scene through an infinitely small hole, causing that each point in the scene projects to exactly one point on the image plane. As a result, a perfectly sharp image is created. To achieve a DoF effect, the corresponding lens physics have to be explicitly simulated. Many different algorithms exist to do so.

Cook, Porter and Carpenter [CPC84] present a method using ray tracing for a realistic simulation of DoF. To achieve the DoF effect, they trace rays through an artificial lens, which is described by its *focal length* and and the aperture number. Latter one defines how much light the lens receives. By tracing the rays through the lens, they simulate light refraction and create realistic DoF.

Haeberli and Akeley [HA90] present a method using the accumulation buffer. In contrast to Cook's et al. method, the scene is rendered from different positions of an artificial lens. The created images are then accumulated in the accumulation buffer resulting in an image exhibiting a DoF effect. The quality is depending on the number of accumulated images. Effects like ghosting or duplicated images are visible, if too few images are rendered.

Potmesil and Chakravarty [PC82] present a method (Demers [Dem04] refers to it as Forward-Mapped Z-buffer DoF), which is a post-processing effect creating DoF using the color and the depth buffer. To generate the effect, pixels are blended from the color buffer into the frame buffer as circular sprites, where the size of the circle depends on the CoC of the blended point. The CoC for a point is calculated based on its depth.

Another method is shown by Arce and Wloka [AW02] (Demers [Dem04] refers to it as Reverse-Mapped Z-buffer DoF), which is also a post-processing effect using a color and the depth buffer. Like in Forward-Mapped Z-buffer DoF, the depth buffer is used to calculate the CoC for every pixel. But instead of blending sprites, pixels are blurred with different kernel sizes depending on the size of the CoC. An example for an implementation is given by Riguer, Tatarchuk and Isidoro [RTI04, p.529–p.556].

Filion and McNaughton [FM08] present Layered DoF. The scene is divided into a set of depth levels which can be defined arbitrarily. A *focal distance* is also defined. All elements around this distance are sharp, whereas objects at a different distance are blurred. Based on the depth level a object belongs to, it is rendered into different image buffers. The DoF image is created by using four color buffers and a depth buffer. The color buffers contain pre-blurred images of the scene at a specific level. The resulting image is created by interpolating between these images depending on the CoC calculated from the depth buffer.

Different variations of this method exist. E.g. the Unreal Engine [unr] uses Layered DoF, but its DoF implementation divides the scene only into three sections. Another version presented by Riguer, Tatarchuk and Isidoro [RTI04, p.529–p.556] uses only two depth buffers, one containing the normal rendered and one containing the blurry scene. This algorithm is very fast but does not consider the spatial distance of objects.

We implemented Layerd DoF. The ray tracing method was dismissed because it is expensive and not suitable for mobile devices. The same goes for the accumulation buffer method. Forward-Mapped Z-buffer DoF was not chosen since the task of blending millions of sprites from one buffer to another is not well supported by graphics hardware and needs therefore be done on the CPU with software [Dem04]. Reverse-Mapped *z-buffer* DoF would have been an eligible option, yet Layered DoF is used by many popular real-time applications like the Unreal Engine or the computer game Starcraft *II* because of its good performance and results.

CHAPTER 3

Applying Different Rendering Styles in a Mobile Molecule Visualization App

In this section we go into further detail by explaining how the extension and the shaders work with focus on the implementation. The extension parses and applies a stylesheet (a file provided by a designer) that defines a molecule rendering style. We will describe the structure of the stylesheet and how the extension processes and applies it on the molecules. Afterwards, we will show how our implementations of screen space ambient occlusion (SSAO), comic shading, outline rendering and depth of field (DoF) are developed using ShaderLab by Unity with focus on mobile devices.

3.1 Using Stylesheets for Molecule Rendering

We implemented an extension that parses and applies a stylesheet whenever a molecule is loaded, resulting in the change of the rendering style of the molecule. The stylesheet allows to define all parameters important for rendering such as:

- declaring what shaders should be used to render which type of object and what the parameters of these shaders should be.
- placing different types of lights with various options like its color or whether it should cast a shadow.
- defining the projection of the camera either as orthographic or perspective.

In the following, we first introduce the sheet's structure before dwelling deeper on the implementation. A template of a JSON stylesheet is shown in Listing 3.1.

For the sake of simplicity, we use JSON as the syntax of the stylesheet. Data is stored as *name/value* pairs. While the *name* is always a string, the *value* stores different types of data, ranging from simple types (number, string) to complex types like arrays or objects. An array can contain values, arrays and objects. Objects can store name/value pairs. For more details about JSON and its syntax, see the JSON specification [jso].

Listing 3.1: Template of a JSON stylesheet used to configure a molecule rendering style.

```
{
        "camera" : {
                "orthographic" : <boolean> ,
                "bgcolor" :
                [ <integer> , <integer> , <integer> ]
                },
        },
        "shaders" : [
                <{
                        "name" : <string>  ,
                        "shader" : <string>  ,
                        "properties" : { ... } },
                }>*
        ],
        "mapping" : {
                <<string> : <string>>*
                "post_effects" : [ <string>* ] },
        },
        "lights" : [
                <{
                        "type" : <string>,
                        "position" :
                                [ <integer> , <integer> , <integer> ],
                        "color" : [ <integer> , <integer> , <integer> ],
                        "intensity" : <float> ,
                        "shadow" : <string> ,
                        "strength" : <float> ,
                        "movable" : <boolean> ,
                }>*
        ]
}
```

The stylesheet consists of four sections:

- *camera*

  The camera key holds an object with the two keys *orthographic* and *bgcolor*. The former defines the projection type of the camera (whether it should be orthographic or perspective). The latter defines the background color of the scene which is the clear color of the camera with a 3-dimensional vector and values within the range of $[0, 255]$.

- *shaders*

  This key stores an array of shader objects. A shader object defines a name for the shader valid in the context of the stylesheet and a second name valid in the context of the application referencing the implemented shader to use. Moreover, it holds a "properties" object, which is shader specific.

- *mapping*

  In this section, the user defines the shaders used to render an object type (e.g. atoms or bonds). *mapping* maps shaders to objects. For the *Waltzing Atoms* app, there are three entries: *atom*, *connector* and *post_effects*. The first two each have as value a shader name which needs to be present in *shaders*. *post_effects* is an array storing names of shaders used for post-processing. These shaders are applied in the order of their appearance in *post_effects* from top to bottom.

- *lights*

  The last key holds an array of light objects, of which each defines one light source in the scene. Such an object stores various parameters: its position, color and intensity, type (point light or directional light) and a shadow strength. Moreover, a light object allows to define whether a light should cast a shadow and whether a light is static or moves with the camera as it rotates.

An overview over the structure of our extension is given in Figure 3.1. It consists of three independent modules. Each of them changes a specific part of the scene based on the used stylesheet.

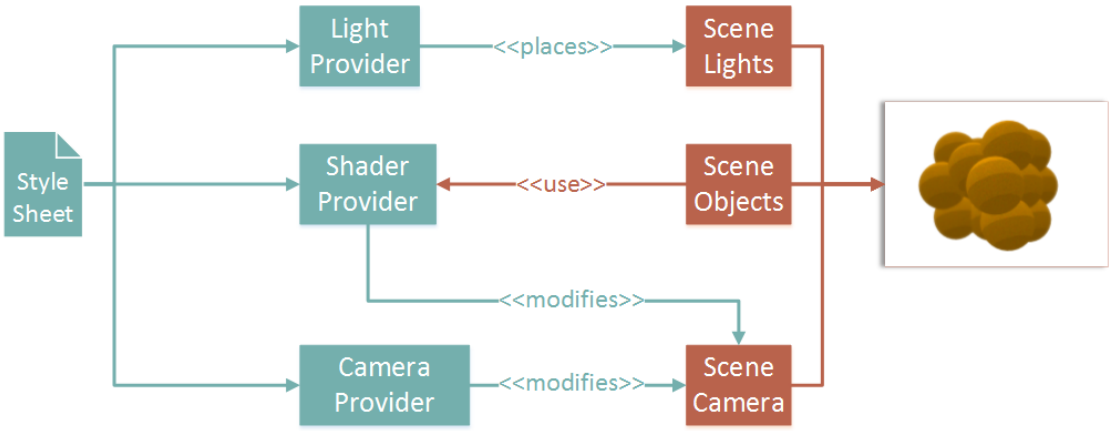


Figure 3.1: Relation of the extension (blue) to the Unity scene elements (orange)

The *LightProvider* as well as the *CameraProvider* are both small modules. The first one works with the *lights* section of the stylesheet and consists of a single class with one method: *placeLights*. When the method is called, all default lights are removed and new lights are placed according to the stylesheet.

The *CameraProvider* module consists only of the *CameraPropertySetter* component. This component is attached to the *Game Object* (GO) that holds the main camera component. Since components of a GO can interact with each other, the *CameraPropertySetter* can change the parameters of the camera component according to the *camera* section of the stylesheet.

The last and by far largest module is the *ShaderProvider*. Its UML class diagram can be seen in Figure 3.2. It loads and holds the shaders for GOs as well as post-processing shaders for the camera. As soon as a GO or the camera request a shader, the *ShaderProvider* passes them the shaders accordingly to the stylesheet's *mapping* section with the corresponding shader parameters set.

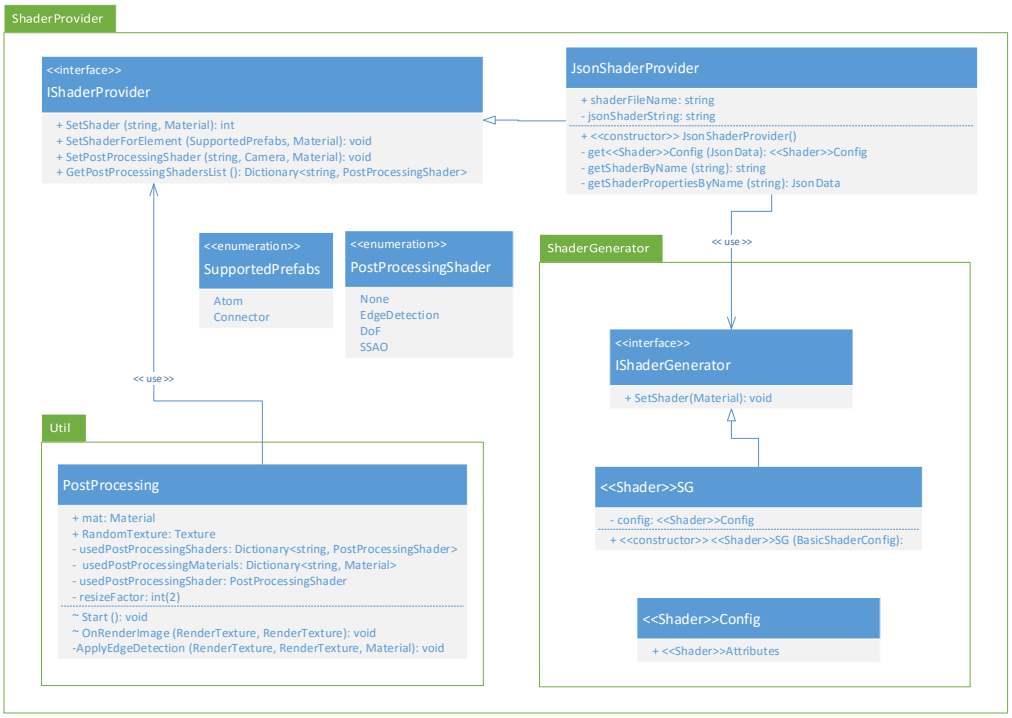


Figure 3.2: UML diagram of the shader provider module

The operation of the module can be seen by considering an example of a single atom GO which is going to be rendered as a part of a molecule. It has a material component that defines how it is rendered based on the shader that the material component holds. This shader, however, is not set yet since it depends on the stylesheet which defines the shader used for the rendering of atoms.

Therefore, the atom GO uses the *SetShaderForElement (SupportedPrefabs, Material): void* of the *IShaderProvider* to ask for the shader it shall use. The input is *Atom* of the *SupportedPrefabs* enumeration and the material of the atom GO.

The *JsonShaderProvider*, provided as *IShaderProvider*, currently used by atoms and connectors resorts to the *mapping* section in the stylesheet to find a shader for atoms and connectors. The example assumes that a toon shader is set. Now the provider calls *SetShaderForElement (string, Material): void* with the toon shaders name gathered from the stylesheets *mapping* sections and the material of the atom GO as input parameters.

This method uses a shader generator (SG) to load the shader and set the properties given by the stylesheet.

Every available shader has to provide a SG, abbreviated with the shader name as prefix and SG, like *ToonSG*. A SG takes a data object as constructor parameter containing all the properties the set shader should have. Those properties are provided by the *JsonShaderProvider* and are parsed from the *shader* section. After the corresponding SG, in the example's case the *ToonSG*, is generated, the material of the atom GO is passed to the SG's *SetShader (Material): void* method. This method sets the shader with the parameters defined in the stylesheet in the material.

After this whole procedure, the atom GO has the correct shader assigned and can be rendered accordingly to the stylesheet. Shaders, however, are not solely available for GOs to be rendered. The stylesheet also provides the option to define post-processing effects in the *mapping* section, which also have to be managed by the *Shader Provider*. For this reason, the module provides the *PostProcessing* component that is attached to the GO that holds the main camera component.

The *PostProcessing* component takes the *JsonShaderProvider*, provided as *IShaderProvider*, to get the list of the used post-processing shaders which is set in the *mapping* section. The component then creates a material for every shader. The shader for the material is set using the *SetPostProcessingShader (string, Camera, Material): void* method. This procedure is similar to the atom GO example before. The only difference is the additional *Camera* parameter. Depending on the post-processing effect, the *ShaderProvider* tells the camera to provide either a depth or a normal buffer or both for the post-processing shaders. During a rendering step, the *PostProcessing* component cycles through the created materials applying the shaders on the rendered image in the given order of the stylesheet.

3.2 Screen Space Ambient Occlusion

Screen space ambient occlusion (SSAO) calculates the occlusion of the scenery per fragment using the depth buffer as a discretized representation of the visible scene. For every fragment, random samples from its neighbourhood are collected. This neighbourhood has a constant size in view space, meaning that the size of the projected neighbourhood is inversely proportional to the depth values in the depth buffer. The SSAO methods presented by Mittring [Mit07] and Filion and McNaughton [FM08] transfer those samples into view space, where their depth is compared to the depth stored in the corresponding depth buffer position. A sample that is farther away from the observer than the depth stored in the depth buffer is considered to be occluded by the geometry, otherwise it is considered unoccluded. The final occlusion factor for a fragment depends on the amount of samples that are considered as inside the geometry. The higher this number is, the less light is received by the surface point stored in a given fragment. The process is shown in Figure 3.3.

For performance reasons, however, we chose to implement the approach given by Méndez [Mén10]. He calculates the occlusion factor *ao* using Equation 3.1, where $\vec{n}$

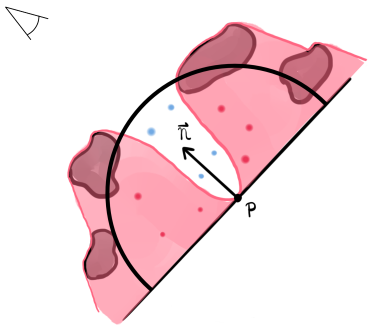


Figure 3.3: Visualization of SSAO with the observer to the top left. The scene is represented by the depth buffer (red line). To calculate the occlusion factor for point P, random samples are placed around it. The distances of these samples to the observer are compared to the corresponding depth values in the depth buffer. The red samples are farther away and are therefore considered to be inside the geometry, even if it is not the case in the scene. The ratio of red samples to blue samples defines the occlusion factor for the point P. The higher this ratio, the more is the point occluded.

denotes the normal vector of the point P.

$$ao = \frac{1}{|Samples|} \sum_{S \in Samples} \frac{max(0, \vec{n} \cdot \frac{(S-P)}{\|S-P\|})}{1 + \|S - P\|} \tag{3.1}$$

In our implementation, we use a shader with one subshader and four passes. The first pass creates the buffer where the occlusion factors are stored per fragment. The second and third pass apply one dimensional Gaussian filters. The last pass blends the occlusion texture into the rendered image.

The first pass is written with the vertex and fragment shader method provided by ShaderLab. We want to focus on the fragment shader (see Listing 3.2), since the vertex shader only transforms the vertices using the MVP matrix and passes the uv-coordinates.

The fragment shader has a hard coded array of offset vectors describing the samples. For every frame and every fragment, the shader iterates over those samples. For performance reasons, we use only 8 random samples, which is a lot less compared to the at least 256 required according to to Engel [Eng09]. The lack of samples causes a visible pattern defined by the samples.

A solution for this problem is given by Filion and McNaughton [FM08]. They use a 2D random texture that stores a random vector per texel. Each sample is reflected on the respective random vector, creating a individual set of random samples for each fragment. Consequently, the pattern is no longer visible.

Our random texture is shown in Figure 3.4. A random vector fetched from this texture is referred to as *rand_norm* and used in Line 15 of Listing 3.2 where it reflects the random sample creating a pseudo random direction.

Figure 3.4: Noise texture used for SSAO kernel randomization.

The next step is to calculate final offset vector using Equation 3.2.

$$\overrightarrow{offset} = \frac{\overrightarrow{rand_{dir}} * Radius}{1 + depth} \tag{3.2}$$

The $\overrightarrow{rand_{dir}}$ vector is scaled by a radius that can be chosen by the user. Then the offset vector is perspectively foreshortened based on the depth of the processed fragment. Using this offset vector, we generate a new sample and calculate its view space position as seen in 19. Finally, we use Equation 3.1 to calculate the occlusion factor. To simplify the blending process in the last pass, we store the inverse occlusion factors. The buffer holding them can be seen in the left image of Figure 3.5.

Some puncturing artifacts remain due to the small sample count. They can simply be removed by applying a low pass filter, in our case a separated Gaussian. High-frequency details are lost, yet the gain in overall quality makes omitting them reasonable. It is, however, important that the blur does not simply filter every pixel equally. It has to respect edges of different objects inside the scene to avoid ambient transitions between different objects. Filion and McNaughton [FM08] use a smart Gaussian filter that extends the original one by taking the normal vector and depth value of each sampled texel into account. If the difference between the depths or the dot product between the normals of a neighbor sample and the center of the blurring kernel is too big, the neighbor sample is omitted. The result of the high pass filtered texture is shown in the right image of Figure 3.5.

In the final step the image needs to be combined with the occlusion buffer. As seen in Equation 3.3, the final color of the pixel c is calculated by multiplying the given color c_S with the inverse ambient occlusion factor ao raised to the power of a scale value k defined by the user in the stylesheet to strengthen or weaken the effect.

$$c = c_S * ao^k \tag{3.3}$$

The final result can be seen in Figure 3.6, where it is compared with the unprocessed rendering. It can be seen that shadows are added at the crevices between the spheres.

Listing 3.2: Code sample showing the calculation of the occlusion factor in the fragment shader.

```
1  float3 norm;
2  float depth;
3  DecodeDepthNormal (tex2D (_CameraDepthNormalsTexture, i.uv),
4                  depth, norm);
5
6  half4 pos = depthToViewPos(i.uv);
7
8  half3 rand_norm = normalize(tex2D (_RandomTexture, i.uv_rand).xyz
9                  * 2.0 - 1.0);
10
11 float ambient_occlusion = 0.0f;
12
13 for(int s = 0; s < RANDOM_SAMPLES_COUNT; ++s)
14 {
15         half3 rand_dir = reflect(RANDOM_SAMPLES[s], rand_norm);
16
17         float2 offset = rand_dir.xy * _Radius / (1 + depth);
18
19         half4 samplePos = depthToViewPos(i.uv + offset);
20
21         half3 diff = half3(samplePos - pos);
22         half distance = length(diff);
23         half3 direction = normalize(diff);
24
25         ambient_occlusion += max( 0.0, dot( direction, norm) )
26                         * ( 1.0 / ( 1.0 + distance ) );
27 }
28
29 ambient_occlusion /= RANDOM_SAMPLES_COUNT;
30
31 ambient_occlusion = 1 - ambient_occlusion;
```

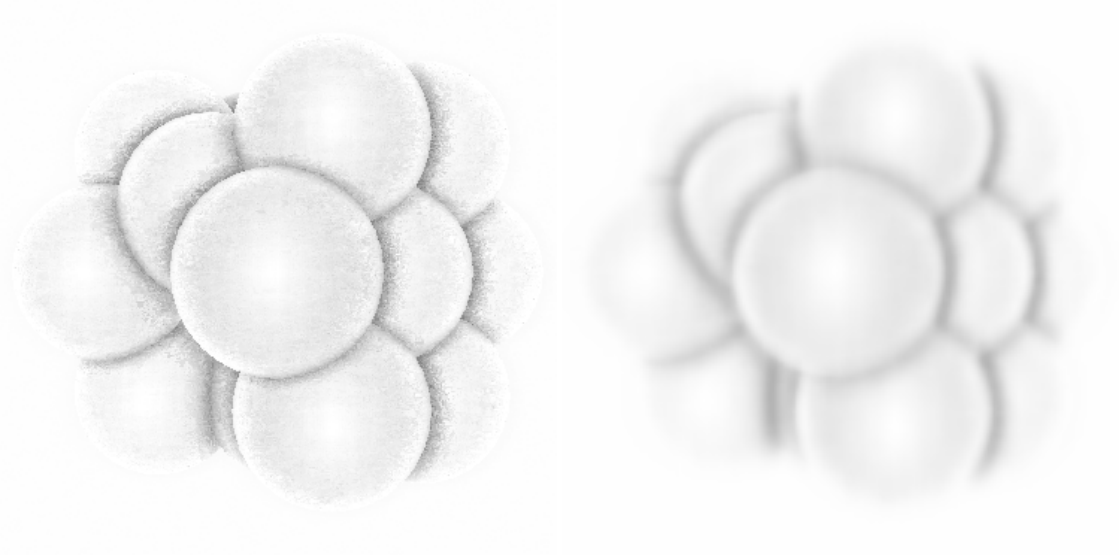


Figure 3.5: Left: Generated occlusion buffer. Right: Blurred using smart Gaussian filter [FM08].

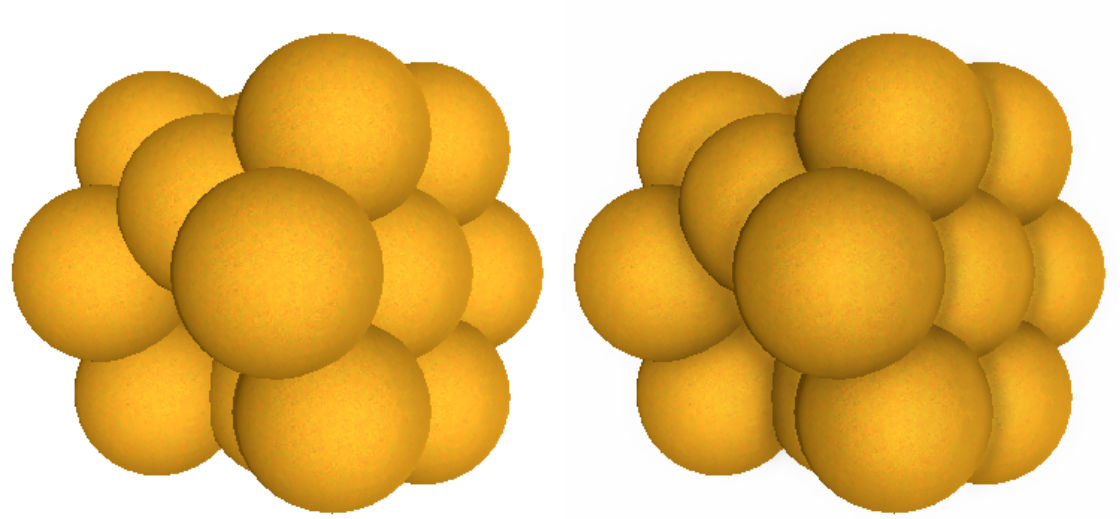


Figure 3.6: Rendered magnesium without (left) and with (right) SSAO.

3.3 Comic Shading and Outline Rendering

We implemented one comic shading method and three techniques for outline rendering, giving a designer many different options on creating a cartoon style. Our comic shader uses the hard shading technique presented by Lake et al. [LMHB00], but since the stylesheet should be the only file required to define a visual style, we are not using

lookup textures. Using such textures would have required external image files besides the stylesheet. Instead, we decided to discretize the gradient and evaluate the shading color on the fly in the shader. Regarding the three outline rendering techniques, we implemented the dot product and the halo method presented by Akenine-Möller, Haines and Hoffman [AMHH08, p.512], as well as the post-processing technique shown by Card and Mitchell [CM02, p.328].

Two shaders have been implemented for this effects. One comic shader performing the hard shading as well as the dot product and halo method and a post-processing shader for the post-processing technique. The comic shader consists of one pass using the vertex and a fragment function where the hull for the halo method is rendered and a surface shader method with a modified lighting method, which will be transpiled to passes by Unity.

3.3.1 Hard Shading and the Dot Product Method

The hard shading as well as the dot product method are done using the surface shader method. It passes the color and the albedo to the lighting method shown in Listing 3.3.

This method applies the hard shading as well as the outline rendering using the dot product method. Line 7 and 20 are calculating the basic color using a modified Lambert term based on the work of Mitchell, Francke and Eng [MFE07] who use Equation 3.4 for their cartoon shading.

$$k_d \left[a(\hat{n}) + \sum_{i=1}^{L} c_i w \left((\alpha(\hat{n} \cdot \hat{l}) + \beta)^\gamma \right) \right] \tag{3.4}$$

The *warp function* $w(x)$ maps the modified Lambert term $(\alpha(\hat{n} \cdot \hat{l}) + \beta)^\gamma$ with the constants α, β and γ to a texture value (see Figure 2.6). Those constants can be set in the stylesheet. The result is multiplied with the light color c_i and added to the ambient term $a(\hat{n})$. Finally the result is multiplied with the albedo k_d sampled from the object's texture map.

In our implementation, we alter this Equation to a discretized version that can be seen in Equation 3.5. The discretization results in the hard shading look.

$$k_d * \frac{(\alpha + \beta)^\gamma}{s} * \left\lfloor \frac{(\alpha(\hat{n} \cdot \hat{l}) + \beta)^\gamma * s}{(\alpha + \beta)^\gamma} \right\rfloor \tag{3.5}$$

The parameter s, which can also be set in the stylesheet, defines the number of hard shading borders for the rendered object. The result with different s can be seen in Figure 3.7.

The discretization is linear and does not provide the same degree of freedom as a lookup texture would. Yet it is still possible to modify the distribution of the hard shading borders by choosing different β and γ, as shown in Table 3.1.

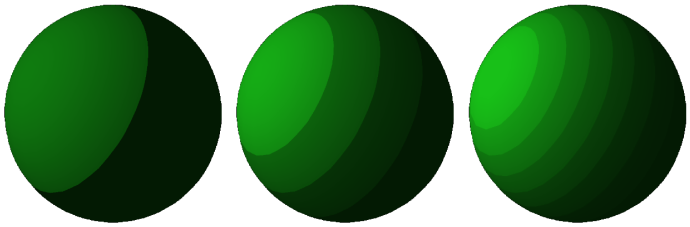


Figure 3.7: Resulting images of shading using Equation 3.5 with $\alpha = 0.5$, $\beta = 0.5$, $\gamma = 2$ and $s = \{2, 4, 8\}$ (from left to right).

Listing 3.3: Lighting method used by the surface shader for cell shading and outline rendering

```
1  half4 LightingModifiedLambert (SurfaceOutput s, half3 lightDir,
2                                  half3 viewDir, half atten)
3  {
4          half4 c = _OutlineColor;
5          if (dot(viewDir, s.Normal) > _OutlineBias) {
6
7          half modNdotL =
8                  pow(s * dot (s.Normal, lightDir) + _Bias
9                  , _Exponent);
10
11         fixed3 albedo;
12         half maxValue = pow(s + _Bias, _Exponent);
13
14         half stepDelta = maxValue / _Steps;
15         half inverseStepDelta = 1 / stepDelta;
16
17         albedo =  s.Albedo * stepDelta
18                 * int(modNdotL * inverseStepDelta);
19
20         c.rgb =   albedo * _LightColor0.rgb
21                         * (modNdotL * atten * 2);
22         }
23
24         c.a = s.Alpha;
25         return c;
26 }
```

Besides the shading, Listing 3.3 also shows the dot product outline rendering method. The resulting color c is initialized with the outline color provided by the stylesheet (seen at Line 4). This color is only replaced if the dot product between the view vector and the normal vector exceeds a user defined threshold. Renderings using different thresholds can be seen in Figure 3.8.

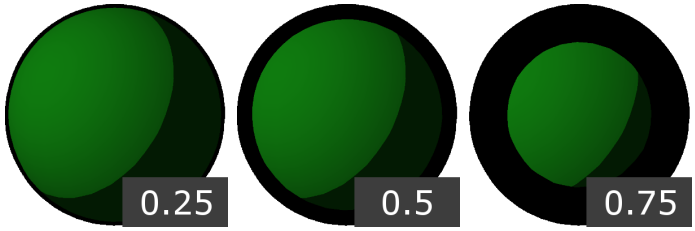


Figure 3.8: Resulting images of shading using different outline thresholds given by the numbers to the lower right.

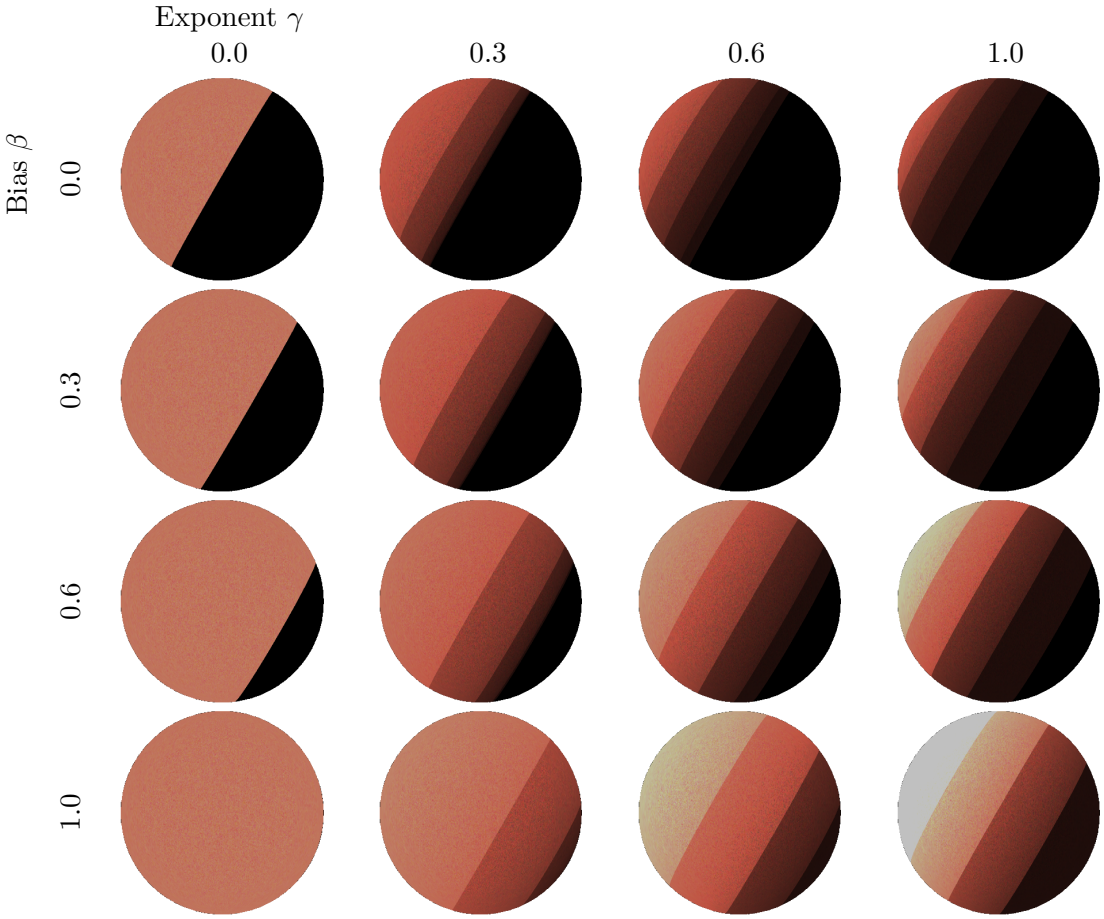


Table 3.1: Different distributions of the hard shading borders caused by varying combinations of the bias β and the exponent γ. $\alpha = 1$ and $s = 4$.

3.3.2 Hull Method

The hull method presented by Akenine-Möller, Haines and Hoffman [AMHH08] is calculated in a separate pass before the hard shading is applied. It is implemented using the vertex and fragment shader methods of ShaderLab. The fragment shader only forwards the color given by the vertex shader. Therefore, we take a closer look at the vertex shader, shown in Listing 3.4. The implementation of this method is straightforward, with little variety in the concrete realization, which is why the implementation of the vertex shader of the Unity Wiki [unib] was used.

Listing 3.4: Vertex shader creating a hull for outline rendering by Unity Wiki [unib].

```
1  fragIn vert(vertIn v) {
2
3    fragIn o;
4
5    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
6
7    if(_HullSize > 0.0001) {
8          float3 norm   =
9          normalize(mul ((float3x3)UNITY_MATRIX_IT_MV, v.normal));
10
11         float2 offset = TransformViewToProjection(norm.xy);
12
13         o.pos.xy += offset * o.pos.z * _HullSize;
14
15         o.color = _OutlineColor;
16   } else {
17         o.color.a = 0;
18   }
19
20   return o;
21 }
```

Firstly, the normal of the vertex is transformed into view space using the inversed transposed model view matrix provided by Unity, $UNITY\_MATRIX\_IT\_MV$. Then the x and y coordinate of the normal vector are transformed into projection space using Unity's method. The z coordinate can be discarded because the hull shall be enlarged along the object's height and width, not its depth. This offset is multiplied with a user defined hull size scale and the z position of the vertex after its transformation for scaling. Note that front-face culling has to be activated in advance, by using an according ShaderLab tag. To allow the disabling of the technique, the outlines are only generated if the user defined hull size exceeds 0.0001, an arbitrarily chosen small value. The results of this outline rendering technique can be seen in Figure 3.9.

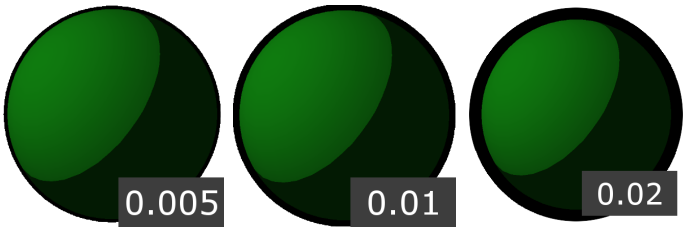


Figure 3.9: Resulting images of shading using the hull method. Hull size given by the number to the lower right.

3.3.3 Outline Rendering Using Post-Processing

The post-processing method for outline rendering shown by Card and Mitchell [CM02, p.328] is done by calculating the gradients of the depth buffer and normal vector buffer for every fragment. If those gradients exceed a certain threshold, the fragment is rendered as an outline.

In our implementation, we use a simple 4-connected neighborhood for their computation. Their uv-coordinates are calculated in the vertex shader and are there stored in an array as seen in Listing 3.5. Since the array is interpolated during the rasterization stage, we avoid to calculate the used neighbours in every fragment.

Listing 3.5: Calculation of the sample uv coordinates in the vertex shader

```
1 o.uv[1] = uv + _MainTex_TexelSize.xy
2           * half2( 1, 1) * _SampleDistance;
3 o.uv[2] = uv + _MainTex_TexelSize.xy
4           * half2(-1,-1) * _SampleDistance;
5 o.uv[3] = uv + _MainTex_TexelSize.xy
6           * half2(-1, 1) * _SampleDistance;
7 o.uv[4] = uv + _MainTex_TexelSize.xy
8           * half2( 1,-1) * _SampleDistance;
```

Listing 3.6: Determination whether the fragment is part of an edge or not in the fragment shader. Sample A and B are samples of the opposite direction (top and bottom, left and right).

```
1 float normDiff = length(sampleANorm - sampleBNorm) < 0.9;
2
3 float depthDiff = abs(sampleADepth - sampleBDepth) <
4         0.045 * (sampleADepth + sampleBDepth);
```

To make sure that the samples are actually scaled in pixel size, the offset vectors are multiplied with the by Unity provided $\_MainTex\_TexelSize$. Its x and y value hold $1/viewportWidth$ and $1/viewportHeight$.

To determine whether a fragment belongs to an edge or not, we calculate the gradients in the fragment shader using the neighbors as seen in Listing 3.6. The threshold 0.9 was arbitrarily chosen. For the depth gradient, we adjust the threshold perspectively. To do

so, we choose the mean of the two samples. The division by 2 is already considered in the term 0.045.

To obtain a simple control over the thickness of the outline rendered with the method of Card and Mitchell, we extended their technqiue by allowing a varying sampling stride for the estimation of the gradients. The results of this simple, yet effective extension is depicted in Figure 3.10 for different sampling strides.

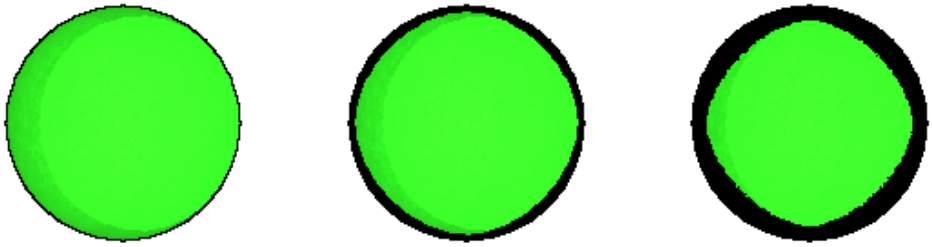


Figure 3.10: Resulting images of post-processing edge detection. Sample distance from left to right 1, 3 and 5.

3.4 Depth of Field

For the depth of field (DoF) effect we implemented Layerd DoF as presented by Filion and McNaughton [FM08] and Riguer, Tatarchuk and Isidoro [RTI04, p.529–p.556]. Our implementation uses three main layers (near, focus and far) and two transition layers providing a smooth conversion between the main layers.

Our algorithm first divides the scene into the main layers based on the z-buffer, storing each of them in a particular frame buffer. The near and the far layer are blurred and finally composed back together with the focus layer to the result image.

The boundaries of the layers can be defined in the stylesheet. Since we use the depth buffer for the decomposition of the scene, these intervals have to be within the range $[0, 1]$. The layer distribution is given as an array of size four, referred to as $\_TransitionRanges$ (TR) in the following code listings. TR[0] defines where the focal layer starts, TR[1] where the near layer ends. The far layer starts blending in at TR[2], and the focal layer ends at TR[3]. The transition layers are given implicitly by the overlaps. (TR[0] $\leq$ TR[1] $\leq$ TR[2] $\leq$ TR[3])

Our DoF shader consists of one subshader with six passes: one pass for each layer segmentation (near, focus and far), two passes for a separated Gaussian filter and one pass for the final composition. All these passes are written using vertex and fragment shader functions.

The shaders used to divide the image into different layers are equivalent. Therefore, the fragment shader segmenting the near layer is shown as an example in Listing 3.7.

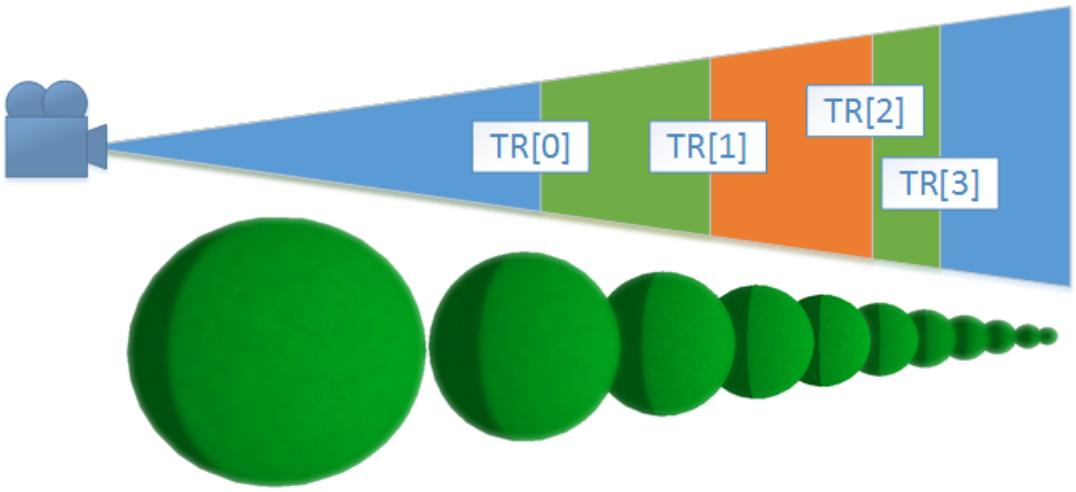


Figure 3.11: DoF layers (l.t.r): near, transition near to focus, focus, transition focus to far, far.

Firstly, the depth value for a fragment is read from the depth buffer. Then it is compared to the according transition range parameters. If the pixel (stored in the _MainTex) does belong to the near layer, its color is stored in the corresponding frame buffer. This happens also for the focus and the far layer.

An example for a result of the layer segmentation with the final composition is shown in Figure 3.12. It can be seen that the focus layer has elements which are part of the near layer as well as of the far layer due to their overlap.

The next step is to blur the near layer and the far layer by applying a separated Gaussian filter. Finally, all layers are composed back together into a single image. The composition is done using the fragment shader shown in Listing 3.8. The near layer is referred to as $\_NearTex$, the focus layer as $\_MainTex$ and the far layer as $\_FarTex$.

To create the final image, texels are fetched from the corresponding layer frame buffers based on the depth value of the processed fragment. If the depth value lies outside of any transition range, a single layer texture is used. Otherwise, the final fragment color is created by interpolating between two layers. This is done by using the provided *lerp* (linear interpolation) function.

Listing 3.7: Fragment shader used to create the near layer of the scene

```
1  half4 frag (v2f i) : SV_TARGET {
2
3    float3 normalValues;
4    float depthValue;
5
6    DecodeDepthNormal(
7          tex2D(_CameraDepthNormalsTexture, i.uv.xy),
8                depthValue, normalValues);
9
10   if(easeDepth(depthValue) < _TransitionRanges[1])
11         return tex2D(_MainTex, i.uv);
12   else
13         return _ClearColor;
14 }
```

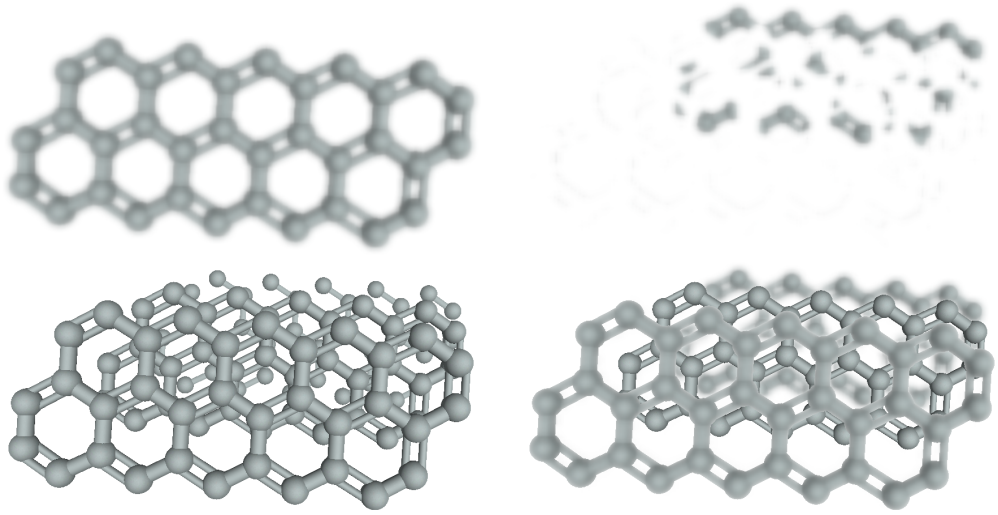


Figure 3.12: Layers after separation and blurring and final composition from top left to bottom right: near layer, far layer, focus layer and final composition. Transition range parameters: $[0.5, 0.59, 0.6, 0.9]$

Listing 3.8: Fragment shader used to composite all layers into a final image

```
1  half4 c;
2
3  if(depthValue <= _TransitionRanges[0]) {
4
5          c = tex2D(_NearTex, i.uv);
6  }
7  else if (depthValue > _TransitionRanges[0]
8          && depthValue <= _TransitionRanges[1]) {
9
10         float v = (depthValue - _TransitionRanges[0])
11         / (_TransitionRanges[1] - _TransitionRanges[0]);
12
13         c = lerp(tex2D(_NearTex, i.uv),
14                 tex2D(_MainTex, i.uv), v);
15 }
16 else if ( depthValue > _TransitionRanges[1]
17         && depthValue <= _TransitionRanges[2]) {
18
19         c = tex2D(_MainTex, i.uv);
20 }
21 else if ( depthValue > _TransitionRanges[2]
22         && depthValue <= _TransitionRanges[3]) {
23
24         float v = (depthValue - _TransitionRanges[2])
25         / (_TransitionRanges[3] - _TransitionRanges[2]);
26
27         c = lerp(tex2D(_MainTex, i.uv),
28                 tex2D(_FarTex, i.uv), v);
29 }
30 else {
31         c = tex2D(_FarTex, i.uv);
32 }
33
34 return c;
```

CHAPTER 4

Results and Evaluation

4.1 Visual Styles

The final styles generated for the three target groups (students, scientists and advertisement) are shown in Table 4.1, where we rendered selected molecules for demonstration. The corresponding summarized stylesheets can be seen in Table 4.2.

4.1.1 Education Style

The comic shader is used to give the molecules a cartoon style amplifying the app's gamification effect. For the hard shading we settled on four gradient steps (since this is a subjective choice, any other number of hard shading borders is also valid). For edge rendering, however, the choice to use this technique is based on the strengths and weaknesses of the different outline rendering methods. For spheres alone, all methods would be suitable. But the connectors cut into the atom spheres producing creases, making a rendering of their outlines not trivial.

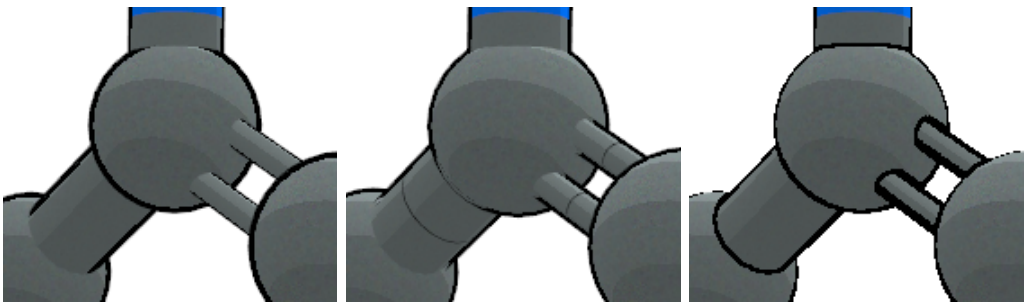


Figure 4.1: Different edge rendering methods from left to right: via dot product, via hull method and via post-processing.

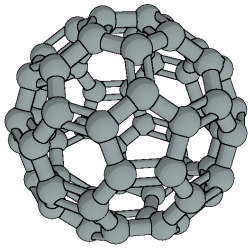
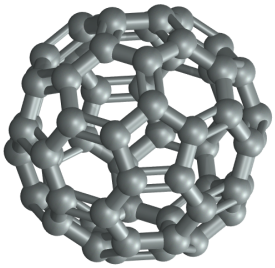
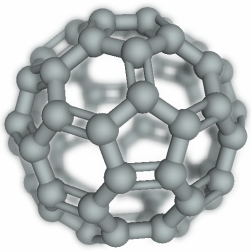
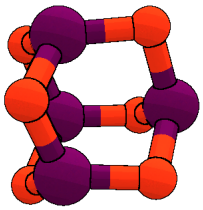
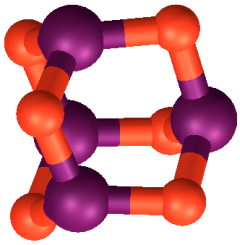
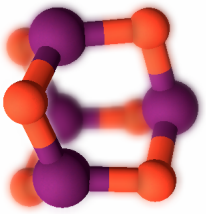
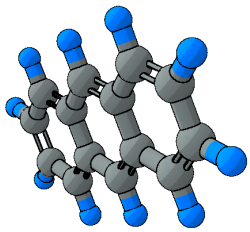
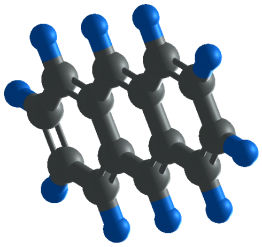
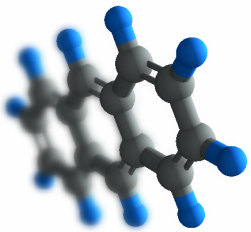
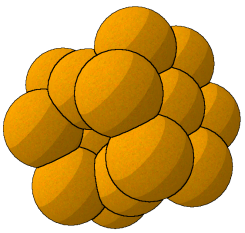
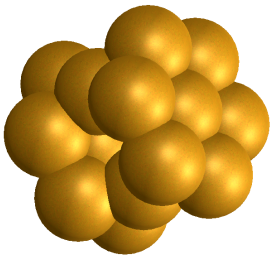
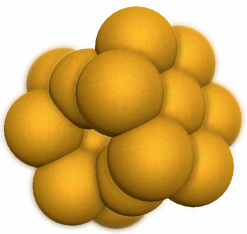
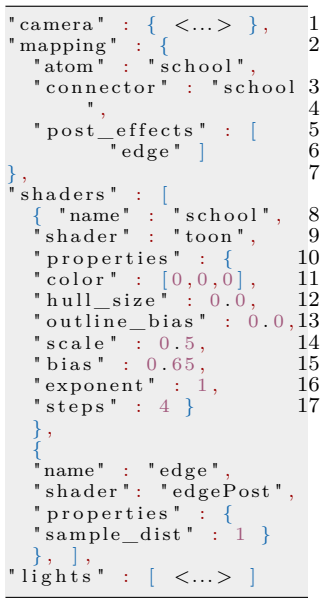
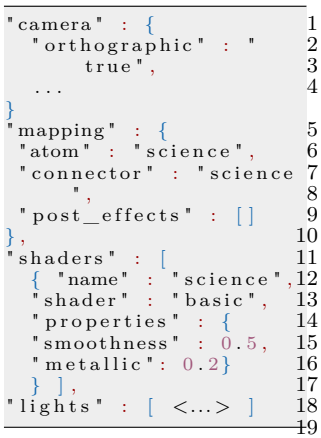
	Education	Scientific	Advertisement
Fulleren(C_{60})			
Arsenik(As_2O_3)			
Anthracen($C_{14}H_{10}$)			
Magnesium(Mg)			

Table 4.1: Different molecules rendered with different styles defined by the summarized stylesheets in Table 4.2.

	Education	Scientific	Advertisement
			

```
"camera" : { <...> },
"mapping" : {
  "atom" : "school",
  "connector" : "school",
  "post_effects" : [
      "edge" ]
},
"shaders" : [
  { "name" : "school",
  "shader" : "toon",
  "properties" : {
  "color" : [0,0,0],
  "hull_size" : 0.0,
  "outline_bias" : 0.0,
  "scale" : 0.5,
  "bias" : 0.65,
  "exponent" : 1,
  "steps" : 4 }
  },
  {
  "name" : "edge",
  "shader": "edgePost",
  "properties" : {
  "sample_dist" : 1 }
  }, ],
"lights" : [ <...> ]
```

```
"camera" : {
  "orthographic" : "true",
  ...
}
"mapping" : {
 "atom" : "science",
 "connector" : "science",
 "post_effects" : []
},
"shaders" : [
  { "name" : "science",
  "shader" : "basic",
  "properties" : {
  "smoothness" : 0.5,
  "metallic": 0.2}
  } ],
"lights" : [ <...> ]
```

```
"camera" : { <...> },
"mapping" : {
  "atom" : "advert",
  "connector" : "advert",
  "post_effects" : [
   "dof", "ssao"] },
"shaders" : [
  { "name" : "advert",
  "shader" : "basic",
  "properties" : {
  "smoothness" : 0.15,
  "metallic": 0.3 }
  },
  { "name" : "dof",
  "shader" : "dofPost",
  "properties" : {
  "layers" :
  [0.1,0.2,0.37,0.43]}
  },
  { "name" : "ssao",
  "shader" : "ssaoPost",
  "properties" : {
  "radius" : 0.03,
  "ssao_factor" : 1}
  } ],
"lights" : [ <...> ]
```

Table 4.2: Summarized stylesheets defining the visualizations in Table 4.1.

The dot product method was the first choice because it provides a good performance and pleasant results. Its visual appearance compared to other techniques is shown in Figure 4.1. The main drawback, however, is the handling of large polygons, which turn entirely black when exhibiting a particular orientation in view space. This tends to be a problem for the connectors as shown in Figure 4.2. Another minor disbenefit is the variance of the line strengths as seen in the leftmost image of Figure 4.1.

The hull method was the second choice, since it only adds a pass doing vertex transformations. The result can be seen in the middle image of Figure 4.1. Compared to the first method, all lines have approximately the same thickness. Yet, this method also has drawbacks due to the outline being an object in the 3D space. The first one can be seen in Figure 4.3, where the outline unintentionally appears under certain viewing angles. The second one, visible in the middle image of Figure4.1 and also in Figure 4.3, are the rendered outlines in the middle of connectors. They appear because a connector consists two cylinders. Since the enlarging of the hull is done per vertex without consideration of the context, the base of the cylinders is also enlarged, resulting in the hull of the closer connector overlapping the more distant one.

The post-processing method renders outlines with approximately same thickness, yet without the artifacts of the hull method. Moreover, more edge types are outlined using

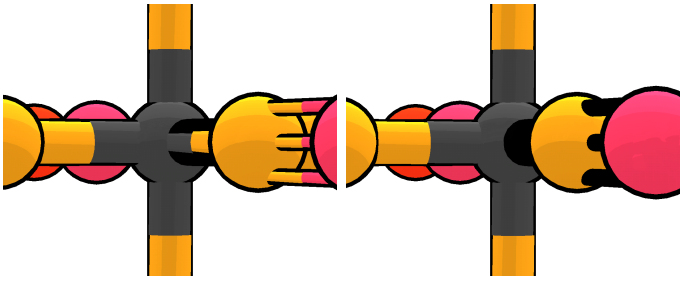


Figure 4.2: Using the dot product outline rendering technique, connecters turn black if the viewing angle is to steep.

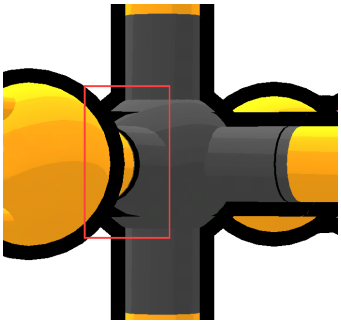


Figure 4.3: Using the hull method outline rendering technique, the outline unintentionally appears under certain viewing angles.

this method than case with the other two. This quality arguments made this approach the method of choice for the implementation in the app. A style that uses this method can be seen in the first column of Table 4.1 and in the last image of Figure 4.1.

4.1.2 Scientific Style

The scientific style was designed to be functional (see second column in Table 4.1). The shading is kept plain such that the spectator is not distracted from the information given by the structure of the molecule. Moreover, an orthographic projection is used instead of a perspective one to support the perception of the structure, because perspective projection makes it difficult to see relative sizes. For shading, the default Unity surface shader is used. Post-processing effects are not applied.

4.1.3 Advertisement Style

The purpose of the advertisement style is to make molecules look spectacular. The idea was to achieve this by applying techniques from photorealistic rendering The rendering of the atoms and bonds is done by the standard Unity surface shader. Realism was added using the implemented post-processing effects SSAO and Layered DoF. The result can be seen in the third column of Table 4.1.

The DoF effect is applied by blurring the part farthest away to set the front of the molecules in focus. We decided to do so to hide artifacts of Layered DoF. Because the effect is applied during post-processing, it misses some information leading to approximation errors, especially in the near layer. If an object is blurred because it is too close to the camera, the edges should also be blurred. Yet, this only can be done correctly if the occluded objects are known, which is not possible after the scene has already been rendered. As a result, the implemented DoF effect does not blur the borders of near objects correctly.

Figure 4.4 shows a close up look on the visual effects of the transition layer. The marked atom is a part of both the focus layer and the far layer, placing it in the transition layer. This results in the atom being rendered sharp, with a glow around the borders. This is another approximation artifact of this DoF method. However, it is only noticable at a closer lookup and can hardly be seen on mobile devices.

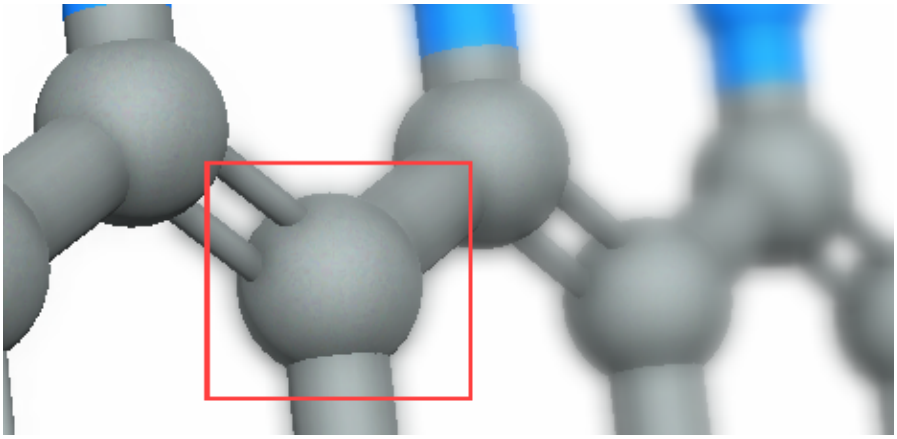


Figure 4.4: Close up look at the advertising style.

Figure 4.4 also shows the use of SSAO in the advertisement style. We decided to use it subtly, such that it is almost unnoticeable, but still enhancing the visual style of the molecule. This can be best seen at the silhouettes of the atom spheres (see, e.g., the Magnesium example in Table 4.1).

4.2 Rendering-Performance on Mobile Devices

To evaluate the applicability of the used rendering techniques, data was gathered to benchmark the performance of the different styles. For this reason, the app was modified to run the same set of molecules with varying visual styles. In the meantime, performance data (frames per second (FPS)) was sent to a server for analysis.

This test setup allowed to benchmark on many different devices without the need for supervision. The app could simply be distributed to the devices of test users who just needed to start the app and run the benchmarks.

Five molecules with varying size were used to test the performance: Water, Anthracen, NaCl, C60 Fulleren and Haemoglobin (see Table 4.3).

Figure 4.5 shows the results of this performance benchmarks. Performance information is given per exercise as FPS, the default rendering performance is used as reference.

We collected performance data from 15 devices. The great variety of test devices can be seen in the high standard deviation almost every performance data shows. High-end tablets were used as well as entry-level smart phones.

To set the collected benchmarks in relative reference, we will compare the performance of the education, scientific and advertisement styles with the performance of a default rendering that only applies a simple Phong illumination without any advanced shader effects. Using the comic style, the mean of the FPS drops by 24% and the median decreases by 20%. Although the comic style only uses a slightly modified shading and one post-processing effect, it has a relative high cost. Looking at the data of the water molecule, however, both the median and the mean are above 30 FPS, suggesting that the performance is still good enough such that the style is usable.

The scientific style exhibits a slightly better performance than the default rendering. The mean of the FPS increases by 9% and the median by 5%. It is without doubt performant enough to be used even on weaker devices.

The advertisement look is the style with the highest performance cost. The FPS mean drops by 76% and the median by 80%. No device was able to provide a smooth interaction. The high-end devices were able to still provide interactive framerates, yet it is not suitable for real-time purposes on mobile devices in general. Nevertheless, it generates nice pictures, which was the main purpose of this style.

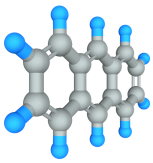
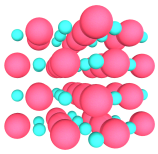
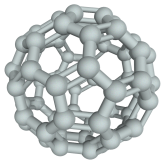
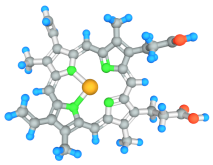
	Molecule	Vertices	Triangles
Water(H_2O)		4,900	6,800
Anthracen($C_{14}H_{10}$)		36,400	47,500
Sodium chloride($NaCl$)		68,100	101,400
Fulleren(C_{60})		104,100	130,600
Haemoglobin		117,200	156,600

Table 4.3: Molecules used for benchmarking the visual styles.

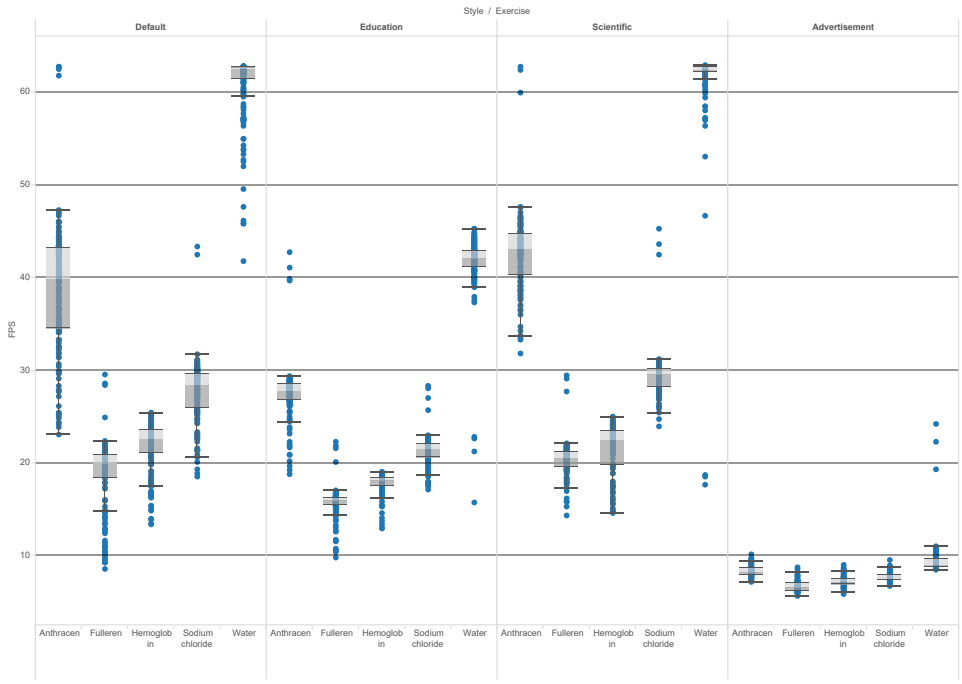


Figure 4.5: Benchmarks of the different styles measured in FPS presented in a Box-Whiskers plot. The blue points show the measured FPS. The dark grey boxes cover the interval $[1^{st}quartile, median]$ and the lighter grey boxes $[median, 3^{rd}quartile]$. The extending lines, the whiskers, show the minimum and the maximum of the data with a maximum distance from the according quartile of 1.5 times of the interquartile range. Points outside the whiskers are outliers.

CHAPTER 5

Conclusion and Future Work

This work implemented a setup to define styles for a mobile molecule visualization app. This setup was then used to define styles for three different target user groups. One for students, one for scientist and one for marketing. Six well known rendering techniques were implemented in Unity and made available for defining styles in the stylesheet.

These rendering techniques are mainly screen spaced methods, because they are the optimal choice for mobile devices with regards of performance. However, using a screen space effect does not guarantee a good performance on mobile devices.

For example, the comic style we defined in Section 3.3 uses only one screen space effect, but is already 20% slower than the default rendering. The advertisement style uses two more complex screen space effects and is not suitable for real-time purposes and thus only used for generating.

However, the styles that have to run at real-time framerates are the comic style and the scientific style. As shown by the performance analysis in Section 4.2, they are actually suitable for this task.

With the implementation of this work, the app is now capable of changing its appearance even during runtime. This feature was used during the tests to change the styles between the runs of the benchmark exercises. For defining new styles, no programming or shader knowledge is necessary. It is enough to define a configuration file with the required parameters. Yet, there are challenges that are beyond the scope of this work.

Firstly, the manipulation of the stylesheet could be even more user friendly (e.g. with a GUI). Moreover, the implemented techniques were all developed because at the beginning of the project, we already had the required styles in our mind. More available shaders would make the stylesheet even more powerful, resulting in more creative freedom for the designer. n the future, we also want to optimize the given shaders for mobile use.

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[AW02] Tomas Arce and Matthias Wloka. In-game special effects and lighting, 2002.

[BS00] John W. Buchanan and Mario C. Sousa. The edge buffer: A data structure for easy silhouette rendering. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, NPAR '00, pages 39–42, New York, NY, USA, 2000. ACM.

[BTM06] Pascal Barla, Joëlle Thollot, and Lee Markosian. X-toon: An extended toon shader. In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '06, pages 127–132, New York, NY, USA, 2006. ACM.

[Bun05] Michael Bunnell. Dynamic ambient occlusion and indirect lighting. *Gpu gems*, 2(2):223–233, 2005.

[Chr03] Per H Christensen. Global illumination and all that. *SIGGRAPH 2003 course notes*, 9:31–72, 2003.

[CM02] Drew Card and Jason L Mitchell. Non-photorealistic rendering with pixel and vertex shaders, 2002.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[cry] Cryengine cross plattform. `http://cryengine.com/features/cross-platform`. Accessed: 17.10.2015.

[Dem04] Joe Demers. Depth of field: A survey of techniques. *GPU Gems*, 1(375):U390, 2004.

[Eng09] Wolfgang Engel. Shaderx7. *Charles River Media*, 2009.

[Eva06] Alex Evans. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, pages 153–171. ACM, 2006.

[FM08] Dominic Filion and Rob McNaughton. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 133–164, New York, NY, USA, 2008. ACM.

[Goo05] David S Goodsell. Visual methods from atoms to cells. *Structure*, 13(3):347–354, 2005.

[HA90] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 309–318, New York, NY, USA, 1990. ACM.

[Hob07] Jia Hoberock. High-quality ambient occlusion. *GPU Gems*, 3, 2007.

[HPAD06] Kyle Hegeman, Simon Premože, Michael Ashikhmin, and George Drettakis. Approximate ambient occlusion for trees. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 87–92. ACM, 2006.

[IFH+03] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte. A developer's guide to silhouette algorithms for polygonal models. *Computer Graphics and Applications, IEEE*, 23(4):28–37, July 2003.

[jso] Json specification. `http://www.json.org/`. Accessed: 13-02-2016.

[KKL+15] Barbora Kozlikova, Michael Krone, Norbert Lindow, Martin Falk, Marc Baaden, Daniel Baum, Ivan Viola, Julius Parulek, and Hans-Christian Hege. Visualization of biomolecular structures: State of the art. *EuroVisSTAR2015*, pages 061–081, May 2015.

[Lan02] Hayden Landis. Production-ready global illumination. *Siggraph course notes*, 16(2002):11, 2002.

[LH13] Diana Libman and Ling Huang. Chemistry on the go: review of chemistry apps on smartphones. *Journal of chemical education*, 90(3):320–325, 2013.

[LMHB00] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 13–20. ACM, 2000.

[Mén10] José María Méndez. A simple and practical approach to ssao. *Graphics Programming and Theory, gamedev. net.(May 2010). http://www. gamedev. net/page/resources/\/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753*, 2010.

[MFE07] Jason Mitchell, Moby Francke, and Dhabih Eng. Illustrative rendering in team fortress 2. In *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pages 71–76. ACM, 2007.

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007.

[PC82] Michael Potmesil and Indranil Chakravarty. Synthetic image generation with a lens and aperture camera model. *ACM Trans. Graph.*, 1(2):85–108, April 1982.

[PG04] Matt Pharr and Simon Green. Ambient occlusion. *GPU Gems*, 1:279–292, 2004.

[QBC+15] Gregory B Quinn, Chunxiao Bi, Cole H Christie, Kyle Pang, Andreas Prlić, Takanori Nakane, Christine Zardecki, Maria Voigt, Helen M Berman, Philip E Bourne, et al. Rcsb pdb mobile: ios and android mobile apps to provide data access and visualization to the rcsb protein data bank. *Bioinformatics*, 31(1):126–127, 2015.

[RTI04] Guennadi Riguer, Natalya Tatarchuk, and John Isidoro. Real-time depth of field simulation. *ShaderX2: Shader Programming Tips and Tricks with DirectX*, 9:529–556, 2004.

[RWS+06] Zhong Ren, Rui Wang, John Snyder, Kun Zhou, Xinguo Liu, Bo Sun, Peter-Pike Sloan, Hujun Bao, Qunsheng Peng, and Baining Guo. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *ACM Transactions on Graphics (TOG)*, 25(3):977–986, 2006.

[Smi60] Deane K. Smith. Bibliography on molecular and crystal structure models, 1960.

[uef] Unreal Engine faq. `https://www.unrealengine.com/faq`. Accessed: 17.10.2015.

[unia] Unity3d Documentation materials and shaders. `http://docs.unity3d.com/Manual/Shaders.html`. Accessed: 07-07-2015.

[unib] Unity3d Wiki silhouette-outlined diffuse. `http://wiki.unity3d.com/index.php/Outlined_Diffuse_3`. Accessed: 16-11-2015.

[unr] Unreal Documentation depth of field. `https://docs.unrealengine.com/latest/INT/Engine/Rendering/PostProcessEffects/DepthOfField/index.html`. Accessed: 04-10-2015.