FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Framework For Real-Time Global Illumination Algorithms

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Markus Klein
Matrikelnummer 01426483

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr. Michael Wimmer
Mitwirkung: Dipl.-Ing. Hiroyuki Sakai

Wien, 31. Jänner 2021

Markus Klein	Dr. Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Framework For Real-Time Global Illumination Algorithms

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Markus Klein
Registration Number 01426483

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Michael Wimmer
Assistance: Dipl.-Ing. Hiroyuki Sakai

Vienna, 31st January, 2021

Markus Klein	Dr. Michael Wimmer

Erklärung zur Verfassung der Arbeit

Markus Klein
Bauernfeldstraße 75, 2231 Strasshof an der Nordbahn

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, 31. Jänner 2021

Markus Klein

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, um Danke zu sagen.

Allen voran möchte ich Dr. Michael Wimmer und Dipl.-Ing. Hiroyuki Sakai, den Betreuern dieser Bachelorarbeit, meine Dankbarkeit aussprechen. Sie hatten immer ein offenes Ohr für meine Fragen und selbst, wenn ich beizeiten mein Studium aus den Augen verloren hatte, konnte ich bis zum Schluss immer auf ihre Unterstützung zählen.

Außerdem möchte ich mich bei meinen Eltern Ursula und Michael bedanken, die mich schon früh mit ihrer Begeisterung für die Welt der Informatik anstecken konnten. Von Ihnen habe ich das Durchhaltevermögen und den Ehrgeiz geerbt, welche sich im Verlauf des Studiums als durchwegs nützlich herausgestellt haben.

Ich darf mich auch bei meinen Freunden aus dem Studium Jonas, Samuel, Timon und Thomas dafür bedanken, dass dank ihnen auch in stressigen Zeiten der Spaß am Studentenleben niemals zu kurz gekommen ist.

Besonders erwähnen möchte ich hier auch meine Mitbewohnerin und langjährige Freundin Veronika, die mich oftmals mit ihren ausgezeichneten Sprachkenntnissen beim Schreiben von fremdsprachigen Texten unterstützt hat und mich mit ihrem übermäßigen Ehrgeiz immer wieder motivieren konnte, auch selbst mein Bestes und mehr zu geben.

Und zu guter Letzt möchte ich mich bei allen bedanken, die mich über die Jahre hinweg durch mein Studium begleitet haben und dieses zu einem Kapitel in meinem Leben gemacht haben, auf das ich auch in Zukunft gerne zurückblicken werde.

Acknowledgements

I would like to use this opportunity to say “Thank you”.

“Thank you” to the advisors of this thesis Dr. Michael Wimmer and Dipl.-Ing. Hiroyuki Sakai for always supporting me during my work and for their patience at times I lost sight of my studies.

“Thank you” to my parents Ursula and Michael for introducing me to the world of computer science at an early age. The endurance and ambition I inherited from them have proven to be very useful during my studies.

“Thank you” to the friends I made during my time at university: Jonas, Samuel, Timon, and Thomas. Even during challenging times they never failed to show me the fun side of being a student.

“Thank you” to my roommate and long-time friend Veronika for proof-reading most of my works written in foreign languages and for motivating me by being a model student herself.

And lastly, a big “Thank you” to everyone else who accompanied me during my studies for making it a chapter in my life I like to look back to.

Kurzfassung

Über die Jahre hinweg wurde eine Vielzahl an Algorithmen zur globalen Beleuchtung entwickelt, welche in Echtzeit rechnen, Interaktivität erlauben und einen hohen Grad an Realitätstreue bestreben. Jedoch ist es oftmals schwierig, die unterschiedlichen Herangehensweisen direkt miteinander zu vergleichen.

Daher wird in dieser Arbeit ein Framework vorgestellt, welches sich genau dieser Problemstellung annimmt und versucht, ein Fundament zum Vergleich von Algorithmen zur Berechnung von globaler Beleuchtung in Echtzeit zu schaffen. Dabei wird auf eine vereinheitlichte Behandlung der Algorithmen gesetzt, unter der Voraussetzung, diese dadurch nicht einzuschränken. Alle Module des Frameworks sind dahingehend konzipiert, um so anpassbar, generisch, erweiterbar und wiederverwendbar wie möglich zu sein, um die wiederholte Implementierung von ähnlichen Konzepten zu vermeiden. Zudem zieht sich ein einheitliches Konzept zur Konfiguration durch das Framework, um auch während der Laufzeit die Anpassung möglichst vieler Parameter zu garantieren.

Abstract

If someone were in need of a real-time global illumination algorithm regarding their specific requirements, they would have no issue finding many possible options nowadays. There are many algorithms that are unmatched in realism, interactivity or performance. However, it might be challenging to compare different approaches side by side.

In this thesis, a framework is proposed that is capable of building a foundation for the comparison of real-time global illumination algorithms. This framework depends on an unified handling of various algorithms while aiming to be nonrestrictive towards them. All modules of the application are designed to be as mutable, generic, extendable, and reusable as possible to avoid the reimplementation of similar concepts. A consistent concept is integrated into the framework to provide a great amount of configurability, even at runtime.

Contents

CHAPTER 1

Introduction

Nowadays, there is a vast amount of global illumination algorithms that operate in real time, support interactivity, and look as realistic as possible. For example, Crassin et al. [CNS+11] introduced voxel-cone tracing in 2011, where a hierarchical voxel octree representation of the scene is used for fast estimation of visibility and illumination intensity. Another approach based on voxel cone tracing was suggested by Sugihara et al. [SRS14], where the radiance is looked up in prefiltered layered reflective shadow maps. An algorithm by Kaplanyan [Kap09] uses the idea of light propagation volumes to create indirect lighting. Silvennoinen et al. [SL17] introduced the reconstruction of radiance from a sparse number of precomputed radiance probes.

However, it might be a difficult task to compare different approaches side by side. For example, it could be interesting to see which algorithm performs best in terms of realism and speed. To complicate the idea of comparing multiple global illumination algorithms further, each one might have slightly different preconditions. They may use different data structures, varying numbers of render passes or use either forward rendering or deferred rendering.

We recognize that there is the need for a tool that facilitates this kind of comparison. This tool should be capable of meeting various requirements of different algorithms to represent a foundation for their comparison. Furthermore, it has to provide the user with a wide range of configuration possibilities and little to no unmodifiable constants. Lastly, such an application would allow switching between implementations of different approaches at runtime to directly compare them against each other. The goal of this thesis is the development of such an application in order to build a foundation for the comparison of real-time global illumination algorithms.

1.1 Related Works

The idea of comparing rendering algorithms is not new. A paper published by Meißner et al. [MHB+00] focuses on four different volume rendering algorithms — raycasting, splatting, shear-warp, and hardware-assisted 3D texture-mapping. The researchers tested those algorithms against each other using different datasets while algorithm-independent parameters were unaltered. Tiede et al. [THB+90] compared four rendering algorithms to find the one that produces the best medical visualizations.

The Mitsuba renderer [Jak10] is a rendering system created by Wenzel Jakob for experimenting with different rendering techniques and their parameters to produce photorealistic images. Since it is an offline-renderer, it neglects real-time illumination and interactivity, though. Its configurability has been a great inspiration for this work.

Naturally, there are also industry-driven projects that can be used to implement and compare rendering techniques. Two rather well-known industrial engines might be Unity [Tec20] and Unreal Engine [Gam20]. They are real-time 3D development tools that allow creators to build their own 3D applications by providing a vast amount of general functionality needed for rendering. They aim for a high level of mutability and configurability to give developers the freedom to create applications regarding their specific requirements.

By examining the related works it can be observed that they do not meet all the requirements stated above. The projects of Meißner et al. and Tiede et al. restrict their comparison to a small amount of rendering algorithms. The Mitsuba renderer does not provide the comparison of real-time global illumination algorithms. Due to the extensiveness of the industrial engines Unity and Unreal Engine, they may come along with a higher complexity and more effort at each introduction of a new algorithm to the comparison.

The tool proposed in the following chapters aims to be tailored to the comparison of real-time global illumination algorithms while being highly mutable and configurable, even at runtime.

CHAPTER 2

Overview

The purpose of this thesis is to develop a tool to simplify the comparison of multiple real-time rendering algorithms. Therefore, it is essential to keep all parts of the application as mutable, generic, extendable, and reusable as possible, so there would not be the need for restructuring the architecture to be able to add a new algorithm.

The application was written in C++, using OpenGL [Gro20] as graphics interface, and the Qt framework [Com20a] for the GUI.

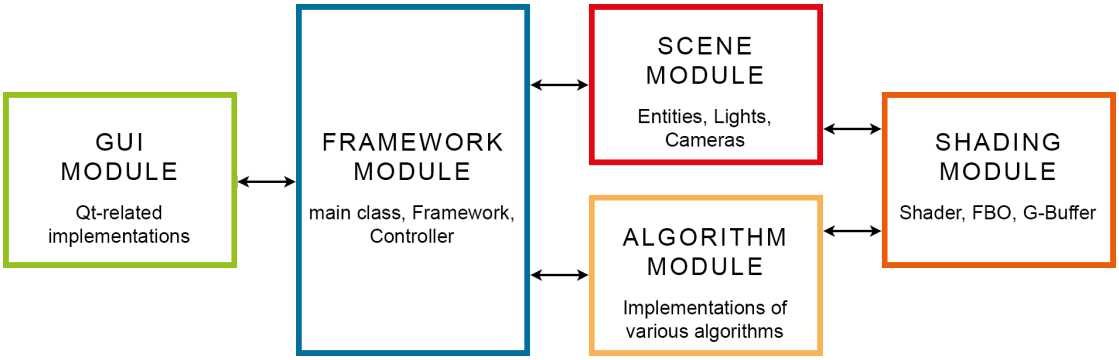

Figure 2.1: All modules and their connections.

Generally, the application is divided into five major modules, as depicted in Figure 2.1:

1. **Framework Module.** The first — as well as the most important — module combines all other modules and makes them accessible to each other. It also handles the initialization of predefined settings and the application itself (Chapter 3).

2. **Scene Module.** Second, there is a module for all scene-related components, combining definitions of entities, light sources, and cameras (Chapter 4).

3. **Shading Module.** The third module facilitates the creation of shaders for either forward or deferred rendering (Chapter 5).

4. **Algorithm Module.** The fourth module includes the different rendering algorithms and additionally a mandatory super class that unifies their handling and allows the user to switch between them at runtime (Chapter 6).

5. **Graphical User Interface Module.** The last module contains Qt-related configurations and implementations for the GUI (Chapter 7).

In the following sections the mentioned modules will be explained in more detail.

CHAPTER 3

Framework Module

The framework module represents the core of the application. It is mainly responsible for initializing, managing, and connecting all other modules. It only consists of three classes: `main.cpp`, `Framework.cpp`, and `Controller.cpp`.

The main class itself is only needed for starting the application by creating an instance of `QApplication` for Qt and one of the framework class which is then accepting responsibility for further procedures.

By running the application, the program starts initializing the GUI and implementations concerning the Qt framework including the OpenGL context. In the course of setting up OpenGL, the scene is loaded from a configuration file and a default algorithm is being prepared. When all components are set up properly, a repeated process for drawing one single frame — the rendering loop — is started. In simplified terms, this process consists of four steps:

1. First, all user inputs — key press events or manipulations of the GUI — are handled and propagated to the responsible module.
2. Afterwards, an update procedure is performed to handle possible changes in the scene or in the current illumination algorithm.
3. The third step is the actual rendering according to the chosen algorithm and execution of all the procedures that need to be done to draw the updated scene to one frame.
4. Lastly, logging and display of debug information for the user are taken care of.

These four steps are repeated in quick succession to draw as many FPS as possible. The more frames are drawn the more interactive the rendering can be, which is an important criterion for the practicability of real-time global illumination algorithms.

3.1 Framework

As stated before, the framework class is also responsible for connecting all modules with each other. The goal was to have bidirectional connections, so any module can access the framework module and further access other modules via the framework module. To achieve this goal, circular dependencies were needed (Section 9.1).

Bidirectional connections were established between the framework class and each of the following: the controller class (Chapter 3.2), the scene class (Chapter 4.5), and each algorithm (Chapter 6.1). A connection to the shading module was not necessary, because it is only needed by the scene module and the algorithms. The GUI module represents a special case, because its components have direct access to the framework and all connected modules, but most of them can only be accessed over the signal-slot-mechanism of Qt that will be explained in Chapter 7.

The last major responsibility of the framework class is loading the configuration file and a scene file. Input streams were used to load these files correctly. The configuration file holds definitions of global constants. Its correct syntax can be found in Appendix A.2. A scene file is used to define all scene objects and their placement within a scene. The correct syntax of a scene file can be found in Appendix A.3.

3.2 Controller

The controller class was designed to be a container for all commonly used global constants and variables. For example, it contains values for camera settings and the sizing of the OpenGL context. Most of these variables are loaded from the configuration file.

Furthermore, the controller class is meant to be an interface for communication with external components. Therefore, it is responsible for handling user inputs such as GUI related events, key press events, or mouse movements. The controller is also capable of propagating debug information to the user.

The default key bindings and other connections to input devices can be found in Appendix A.4.

CHAPTER 4

Scene Module

The scene module contains all scene-related implementations. Additionally, it is responsible for the administration of all scene objects — either entities, light sources, or cameras — and provides mechanisms for updating and drawing the scene. Each scene object can be identified by its unique name and provides a parameter map for configurability (Chapter 9.2).

4.1 Entities

Entities are objects of any shape that can be rendered and seen on the screen. Besides the unique name and a parameter map, all entities contain the following information:

- one or more polygon meshes. A description of the representation of meshes can be found in Appendix A.1.
- a VAO that holds information about vertex positions, surface normals, and texture coordinates
- a number of materials to describe their reflectance characteristics
- a model matrix to determine their position, rotation, and scaling within the scene
- a definition of their outer bounding box

There are two available means to initialize an entity in the framework: the creation of a simple cuboid or the loading of a more complex entity from a file.

The first option is to describe an cuboid parallel to the coordinate system by only two points — a minimum point and a maximum point. This type makes use of a technique

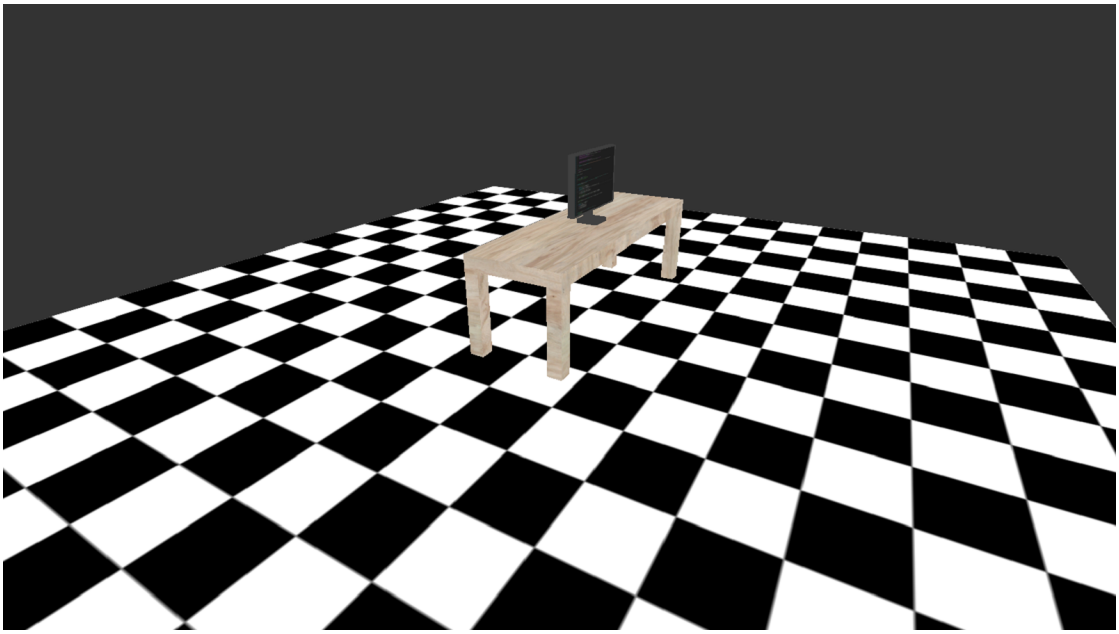

Figure 4.1: The studyroom scene with a cuboid as the floor and two more complex entities loaded from files.

described in Chapter 9.3, where only two points are needed to determine the positions of all eight vertices of a cuboid. The calculated vertex points are then connected by multiple triangles to create a solid mesh.

The other option to obtain more complex entities is by loading them from a file of either the `.obj` file format or `.dae` file format. For model loading the Open Asset Import Library [Kim18] was integrated into the framework. This library provides means to import various 3D model formats in a uniform manner.

The calculation of the outer bounding box is the same for both entity types. During the creation of the entity, a minimum coordinate and a maximum coordinate is found for each axis to determine a minimum point and a maximum point. These two points define a cuboid, called the bounding box, surrounding the whole entity. The use for the bounding box of an entity is explained in Chapter 4.4.

4.2 Light Sources

The definition of different light sources is vital for illuminating a scene.

There are four types of light sources implemented in the framework: ambient lights, directional lights, point lights, and spot lights. As they are scene objects, they all have a unique name and provide a parameter map. Furthermore, each light source emits light of a certain color at a defined intensity.

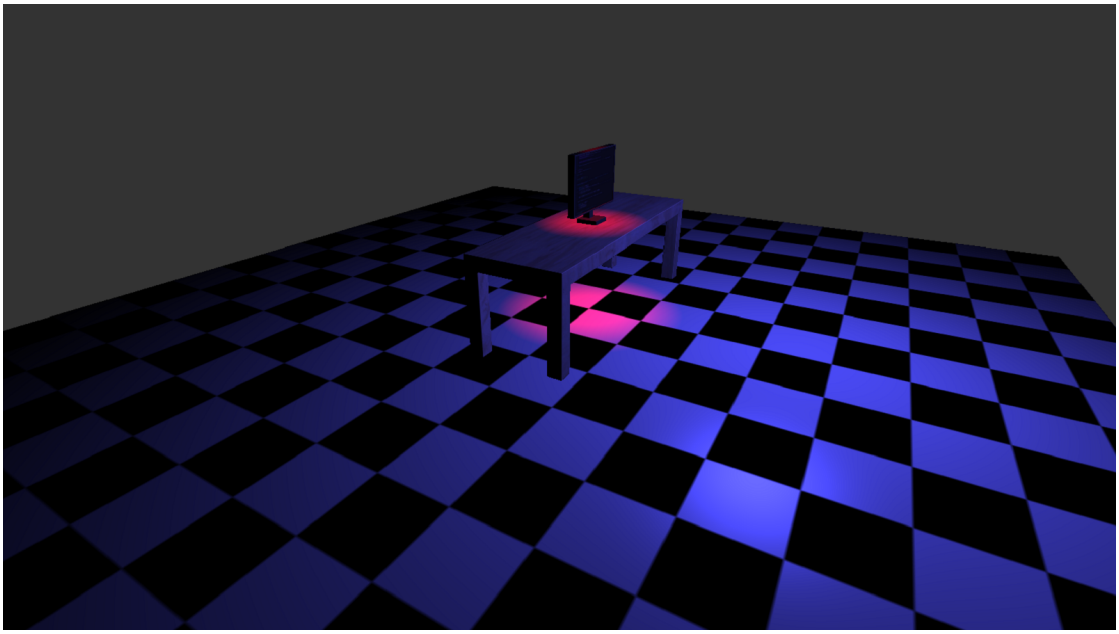

Figure 4.2: The studyroom scene with one blue point light and a red spot light.

Ambient Light. Ambient light describes an omni-directional light that does not originate from any specific position. It illuminates all entities equally from all directions with a constant intensity.

Directional Light. The light of a directional light does not have a specific origin and illuminates all entities with a constant intensity. It differs from ambient light by emitting parallel light rays in a certain direction instead of being omni-directional.

Point Light. A point light emits its light rays in all directions originating from one single point. Its intensity reduces with the distance to the illuminated entity. This attenuation is characterized by the constant, linear, and quadratic coefficients.

Spot Light. A spot light is an extension of a point light. In addition to originating from a certain origin, it also defines a direction and an aperture. Therefore, it emits light rays in a cone-shaped form.

4.3 Camera

The camera defines which parts of the scenes are visible from a certain position and direction. The camera can be moved or tilted freely within the scene using the keyboard and the mouse (Appendix A.4).

Whereas the position is defined by a single three-dimensional vector, the direction is described by two angles called yaw and pitch, as shown in Figure 4.3.

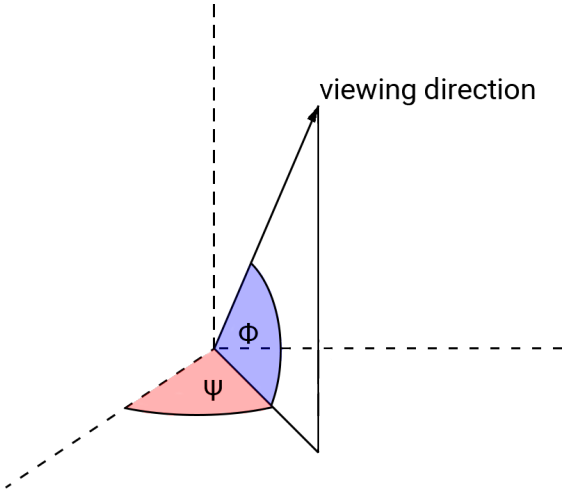

Figure 4.3: Yaw (Ψ) defines a tilting angle going from right to left. Pitch (Φ) describes the tilting angle going up and down.

By limiting pitch to a 85° angle in both directions starting from the horizontal, the camera can be prevented from rolling over and ending up upside down.

The camera is updated for each frame. Before the update, the camera is sent input information — whether the camera should be moved or tilted — from the Controller (Chapter 3.2).

```
//update tilting
_yaw -= tiltingOffset.x * deltaTime;
_pitch += tiltingOffset.y * deltaTime;

while (_yaw > 360.0f) {
    _yaw -= 360.0f;
}
while (_yaw < 000.0f) {
    _yaw += 360.0f;
}
if (_pitch > 85.0f) {
    _pitch = 85.0f;
}
if (_pitch < -85.0f) {
    _pitch = -85.0f;
}

_front.z = (-1) * cos(_yaw) * cos(_pitch);
_front.x = (-1) * sin(_yaw) * cos(_pitch);
_front.y = sin(_pitch);

_front = normalize(_front);
```

```
_right = normalize(cross(_front, _worldUp));
_up = normalize(cross(_right, _front));

//update movement
if (direction == FORWARD) {
    _position += _front * movingSpeed * deltaTime;
}
if (direction == BACKWARD) {
    _position -= _front * movingSpeed * deltaTime;
}
if (direction == RIGHT) {
    _position += _right * movingSpeed * deltaTime;
}
if (direction == LEFT) {
    _position -= _right * movingSpeed * deltaTime;
}
if (direction == UP) {
    _position += _worldUp * movingSpeed * deltaTime;
}
if (direction == DOWN) {
    _position -= _worldUp * movingSpeed * deltaTime;
}
```

Listing 4.1: (Simplified) update procedures for a camera.

The update procedure per frame for the camera can be seen in Listing 4.1. At first, yaw and pitch are adjusted taking incoming tilting information into account. Afterwards, the recalculation of the viewing direction using yaw and pitch is being processed.

To help with further calculations, two other directions, one pointing to the right and one pointing up, are also calculated. The limitation of pitch to 85° prevents the viewing direction from pointing in the same/opposite direction as the constant world-up vector `(0, 1, 0)`. Otherwise the two other calculated directions might result in zero vectors and hence be invalid.

Lastly, the new position of the camera is being determined by translating it in the requested direction along the recalculated directions.

4.4 View Frustum

Connected to each camera is a view frustum that describes the region visible to this specific camera. It is described by six planes. Each plane is defined by a position and a normal vector which is always pointing inwards into the view frustum. Any point is considered inside of a plane, if the dot product of the normal vector and a vector between the plane's position and the tested point is greater or equal to 0. Because the view frustum is dependent on the camera, its planes are always updated as part of the camera update.

The view frustum is an essential part of view frustum culling: a technique to reduce rendering calculations by checking if an entity is within the camera's view. To save

calculation costs, only the entities bounding box vertices are consulted for culling. If all eight vertices of the bounding box are outside of any of the six planes describing the view frustum, it can simply be disregarded from any further rendering calculations for the current frame.

4.5 Scene Class

The scene class acts as the container for all scene objects — entities, light sources, and cameras — and takes responsibility for their administration.

It provides methods to add or remove any scene object and takes responsibility for giving each scene object an unique name by creating one if needed.

Also, a mechanism to help draw all entities within the camera's view is provided. All entities are tested if they are inside of the view frustum and afterwards passed through a shader program (Chapter 5.1) to be drawn.

Furthermore, it supplies an update mechanism to help update scene objects for each frame. At the moment, the mechanism only updates the camera, because entities and light sources are currently static and do not implement an update mechanism themselves.

CHAPTER 5

Shading Module

In general, the rendering of an entity in OpenGL means passing each of its meshes through a shader program to draw it onto one or multiple render targets. A shader program is a sequence of different shaders that execute various calculations which ultimately result in the mesh to be drawn properly. A render target may either be the screen itself or a texture that can be used in subsequent rendering passes.

The shading module is responsible for providing a variety of measures needed to render a scene. It encapsulates various parts of the OpenGL functionality [Gro20] to make them commonly usable for different algorithms.

5.1 Shader

As stated before, a shader program is a linkage of multiple shaders. A shader itself is autonomous from the framework. It is written in C and has access to various functions of the GLSL. There are currently three types of shaders available for use in the framework.

Vertex Shader. A vertex shader performs calculations based on a meshes' vertex positions. It can be used to transform the vertices' coordinates from object space into their designated position in screen space.

Geometry Shader. A geometry shader is capable of creating new geometries. A single point that would not have been visible on the screen could be transformed into a small quad that can be rendered onto the screen.

Fragment Shader. The fragment shader performs calculations on small fragments of polygons in between of vertices. This can be used to calculate the resulting color of a given fragment.

The functionality of shader programs is encapsulated in the `BaseShader` class in the framework. It takes paths to shader code files of different shader types as input and is responsible for loading and linking them to a single shader program.

Shader programs are solely given information stored in the VAO of a mesh as input from OpenGL. However, this information is in most cases not sufficient to do all necessary calculations for rendering. Therefore, a shader program can additionally have multiple uniforms. Uniforms are variables or constants that are set externally and are not modified within the program.

There are various uniforms used commonly by different shader implementations, e.g. the model matrix of an entity or information about light sources. The `WorldShader` class is aiming to encase such variables or constants and simplify their usage within the framework. It is a subclass of the `BaseShader` class and provides multiple methods to set commonly used uniforms to the shader program.

The `ScreenShader` class is meant to be used for two-dimensional rendering and the processing of images. It represents a special case within the framework for it is a combination of a shader program and an entity. Similar to the `WorldShader` class, the `ScreenShader` class is also a subclass of the `BaseShader` class and provides different methods to set various uniforms. However, it also contains a two-dimensional quad representing the screen and provides means to render any image onto the quad and hence onto any render target.

5.2 Framebuffer Object

Instead of rendering a frame directly onto the screen, it is also possible to render it to an image by using a FBO. While the frame rendered to the screen is no longer usable or modifiable, the rendered image in a FBO can be used afterwards in further render processes. Therefore, it is possible to perform post-processing, apply image filters to it or use it for deferred rendering (Chapter 9.4).

The functionality regarding FBOs provided by OpenGL is encapsulated in the `FBO` class in the framework. The class currently offers two types of FBOs: one for rendering colors to a single render target and one for rendering the depth buffer. Furthermore, the `FBO` class provides methods to activate the FBO before rendering and to bind the rendered image to any color attachment afterwards.

5.3 G-Buffer

A G-buffer is in simple terms a FBO with multiple render targets used for deferred rendering (Chapter 9.4). Each target is responsible for extracting different information from the rendered scene. As shown in Figure 5.1, one target could hold information about surface colors of entities in the rendered scene whereas another target contains data about surface normals.

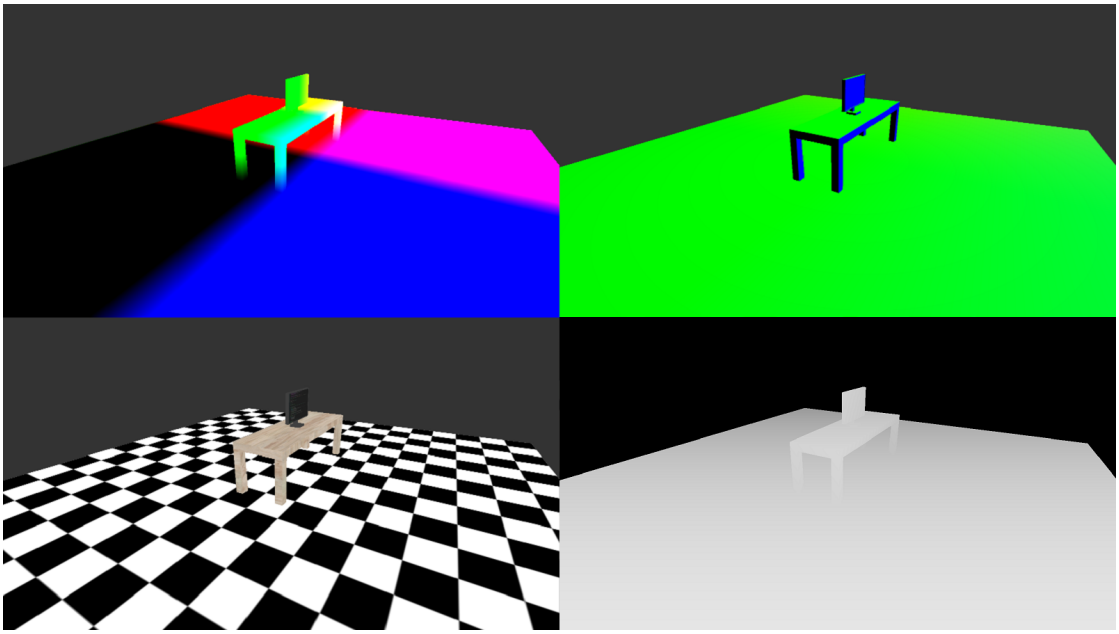

Figure 5.1: Example of resulting images in the G-buffer after its render pass.

The `GBuffer` class accepts responsibility for a whole render pass. It takes shader code files and the number of render targets as input and encases the creation of a FBO and the initialization of a shader program. Additionally, the `GBuffer` class provides functionality to process one render pass and to retrieve the rendered data from the embedded FBO afterwards.

A built-in standard G-buffer implementation provides four images after its render pass: one for world positions, one for surface normals, one for surface colors, and one for the depth buffer (Figure 5.1).

CHAPTER 6

Algorithm Module

A specific technique or procedure, involving several steps to create a frame, and being repeated for each rendering loop is labeled as algorithm within the framework. Algorithms may be quite different from one another, yet have some similarities.

Therefore, it has been a main focus of this work to create a uniform base for different algorithms to be able to compare them. They also need to be configurable at runtime (Chapter 9.2). Furthermore, all algorithms depend on a variety of measures to help render a scene (Chapter 5).

6.1 Integration Of Additional Algorithms

The uniform base for all algorithms in the framework is the `BaseAlgorithm` class and therefore, it must be the super class of any algorithm implementation.

The `BaseAlgorithm` class provides a reference to the framework for each algorithm. It also supplies the following variables:

- **`_name`.** a unique name
- **`_updateRequested`.** flag to determine whether the update mechanism should be executed
- **`_setupRequested`.** flag to determine whether the setup mechanism should be executed
- **`_parameterMap`.** a container for all parameters that are modifiable
- **`_fbo` (optional).** this FBO can be used to render the resulting image
- **`_gbuffer` (optional).** this may hold a reference to a G-buffer

- **`_worldShader` (optional).** this may hold a reference to a 3D-shader program
- **`_screenShader` (optional).** this may hold a reference to a 2D-shader program

To harmonize the handling of algorithms within the framework, each one must implement the following methods:

`process(int outputTextureUnit).` This method is responsible for rendering one frame. It has one input parameter that determines the texture unit the resulting image needs to be bound to for further proceedings in the rendering loop. This method is responsible for executing the update and/or setup mechanism if either one of them has been requested.

`update().` The update method should be used for any updates of the algorithm. It is usually requested if scene objects or parameters in the parameter map change. Therefore, it may be used to update uniforms of a shader or recalculate other parameters. An exception from the update mechanism is the camera for it is assumed to be updated for each frame and therefore does not request an additional update.

`setup().` This method should be used for initial preparations and calculations before using the algorithm within the rendering loop. It is requested every time the algorithm is selected by the user.

`resize().` This method is called whenever the frame size has changed. Due to FBOs being error-prone on resizing, they should be reinitialized here.

`setParameter(string key, string value).` This method is called by the framework whenever a parameter is modified. To avoid timing issues, modified parameters should at first only be set within the parameter map and request an update rather than setting it directly to a shader or similar.

The implementation of an algorithm also needs to be added to the algorithms map within the `setupGLModules` method in the framework class. An example of this can be found in Listing 6.1.

```
shared_ptr<GoochAlgorithm> goochAlgorithm =
    make_shared<GoochAlgorithm>(this, "GoochAlgorithm");
_algorithmsMap[goochAlgorithm->getName()] = goochAlgorithm;
```

Listing 6.1: Integration of additional algorithms in `setupGLModules()` method within `framework.cpp`.

6.2 Exemplary Algorithm Implementations

Currently, there are three implementations of algorithms in the framework: two approaches of Phong Shading [Pho75] and one of Gooch Shading [GGSC98].

Phong Shading describes a rendering technique for calculating the color of a fragment in regard of the surface color, the surface normal-vector, and a reflection model. The reflection model combines ambient, diffuse, and specular reflection to imitate realistic lighting scenarios.

Gooch Shading describes a non-photorealistic rendering technique that uses warm and cold tones to differentiate between surfaces facing towards a light source and those facing away from it. Additionally, object edges are highlighted in black.

The implementations of these algorithms serve solely as examples for possible algorithms and are not optimized in performance or realism.

Phong Shading (Forward Rendering). This may be an example of a simple implementation of any algorithm in the framework using forward rendering. The resulting image is shown in Figure 6.1. As shown in Listing 6.2, its `process` method only consists of the following basic steps:

1. Execute update/setup functionality if requested.
2. Bind the FBO.
3. Clear the Viewport
4. Propagate camera variables to the shader program.
5. Draw the scene.
6. Unbind the FBO and bind the resulting image to the given texture unit.

```
void process(int outputTextureUnit) {
    if (_setupRequested) {
        setup();
    }
    else if (_updateRequested) {
        update();
    }

    _fbo->bindBuffer();

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
    glViewport(
        0,
        0,
```

```
        _framework->getController()->getWindowWidth(),
        _framework->getController()->getWindowHeight()
    );

    Scene* scene = _framework->getScene();

    _worldShader->useShader();
    _worldShader->setCameraPos(
        scene->getCamera()->getCameraPosition()
    );
    _worldShader->setViewProjection(
        scene->getCamera()->getViewProjectionMatrix()
    );

    scene->draw(_worldShader);

    _fbo->unbindBuffer();
    _fbo->bindTexture(outputTextureUnit);
}
```

Listing 6.2: Exemplary implementation of the `process` method for forward rendering.

Phong Shading (Deferred Rendering). This implementation displays the usage of deferred rendering (Chapter 9.4). The G-buffer renders all needed information about the scene into multiple render targets. All shading calculations are then done in screen space within a 2D-shader program. For testing purposes, the resulting images of the G-buffer are also rendered into the final frame. The resulting image is shown in Figure 6.2.

Gooch Shading. This is a rather free interpretation of the gooch shading algorithm, as seen in Figure 6.3. It lacks the creation of black edges and the usage of the warm tone relates to the position of the camera rather than to a light source. The implementation of this algorithm serves as an example for the usage of the parameter map and the configurability of an algorithm during runtime.

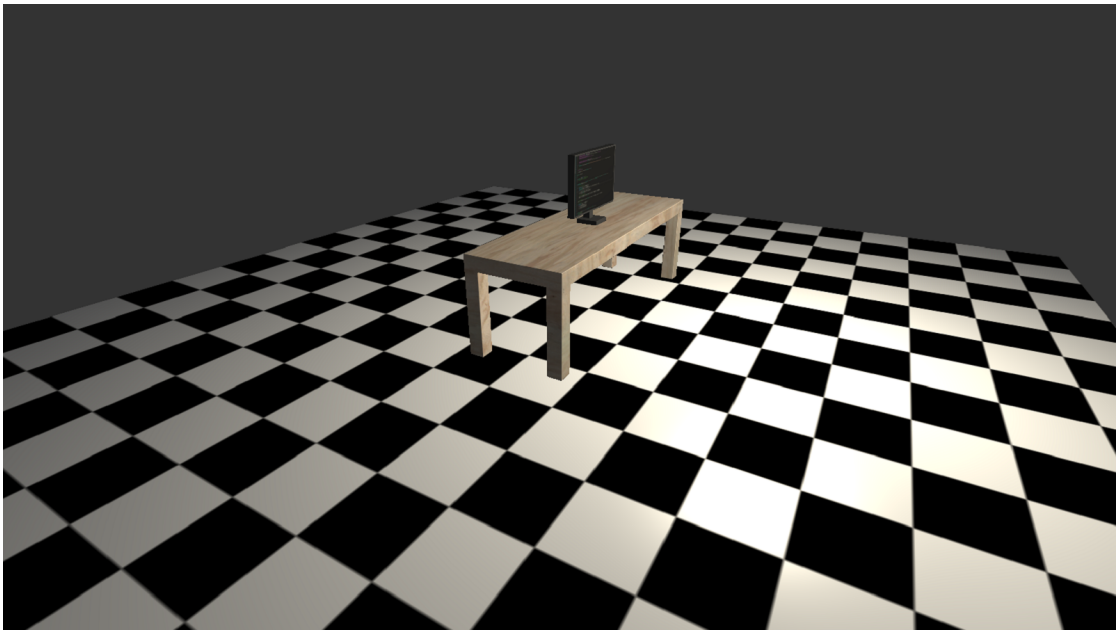

Figure 6.1: Phong Shading (Forward Rendering).

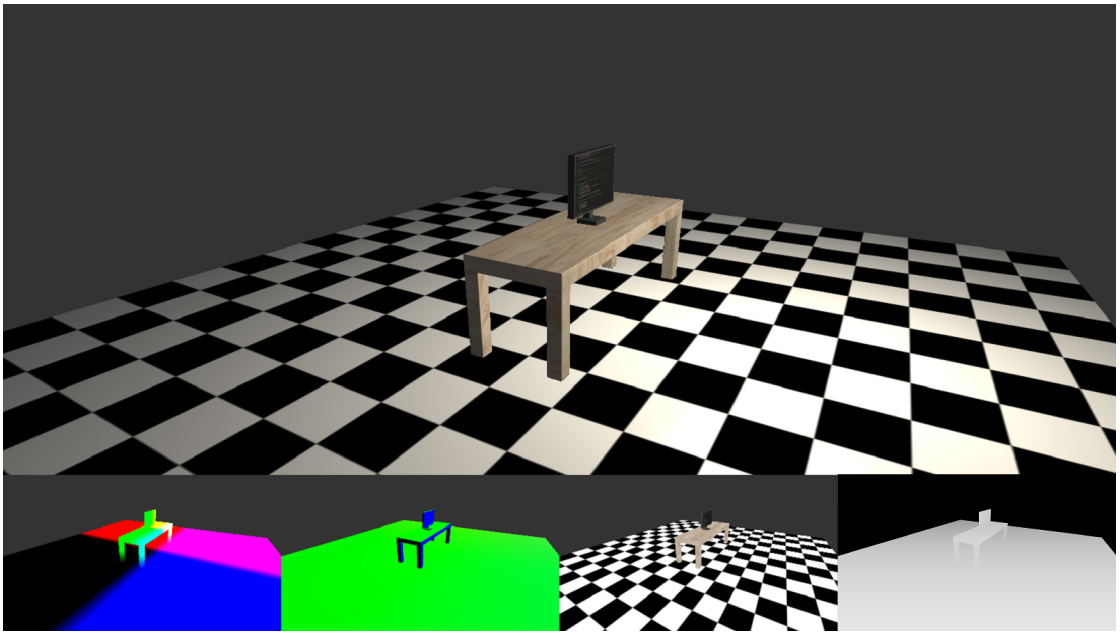

Figure 6.2: Phong Shading (Deferred Rendering).

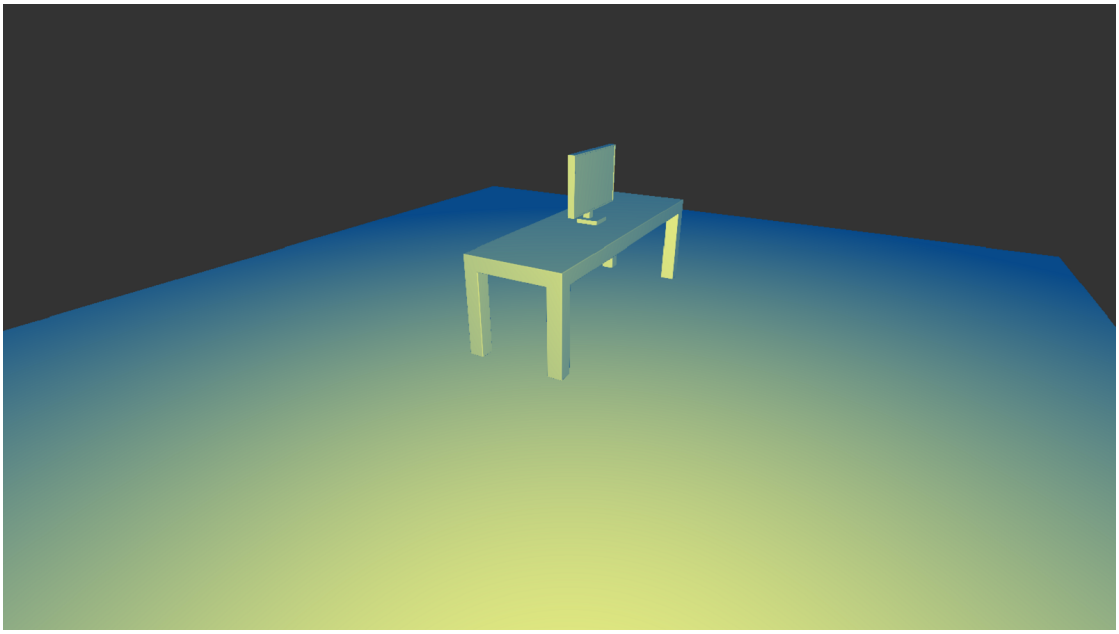

Figure 6.3: Gooch Shading.

CHAPTER 7

Graphical User Interface Module

This module is concerned with all GUI related implementations. The GUI has been created with Qt, a framework for creating cross-platform applications [Com20a].

The Qt functionality is separated into several modules. For this framework, only the following modules needed to be integrated:

- **Qt Core.** the core functionality of Qt
- **Qt GUI.** Qt functionality for creating GUIs
- **Qt Widgets.** extension of the Qt GUI module for C++
- **Qt OpenGL.** extension of the Qt GUI module for OpenGL

7.1 GUI Components

As shown in Figure 7.1, the application consists of one main window. This main window is separated into the following three areas.

Configuration Tabs. The tabs are used to configure and modify various parameters of the framework and its modules (Chapter 9.2). In general, a tab contains a table view for accessing the parameter map of an object. If there are multiple objects of the same kind, they can be switched via selection in a combobox. For some objects, instances can be added or removed per button clicks.

There are currently four configuration tabs:

- **“General”-tab.** contains global settings

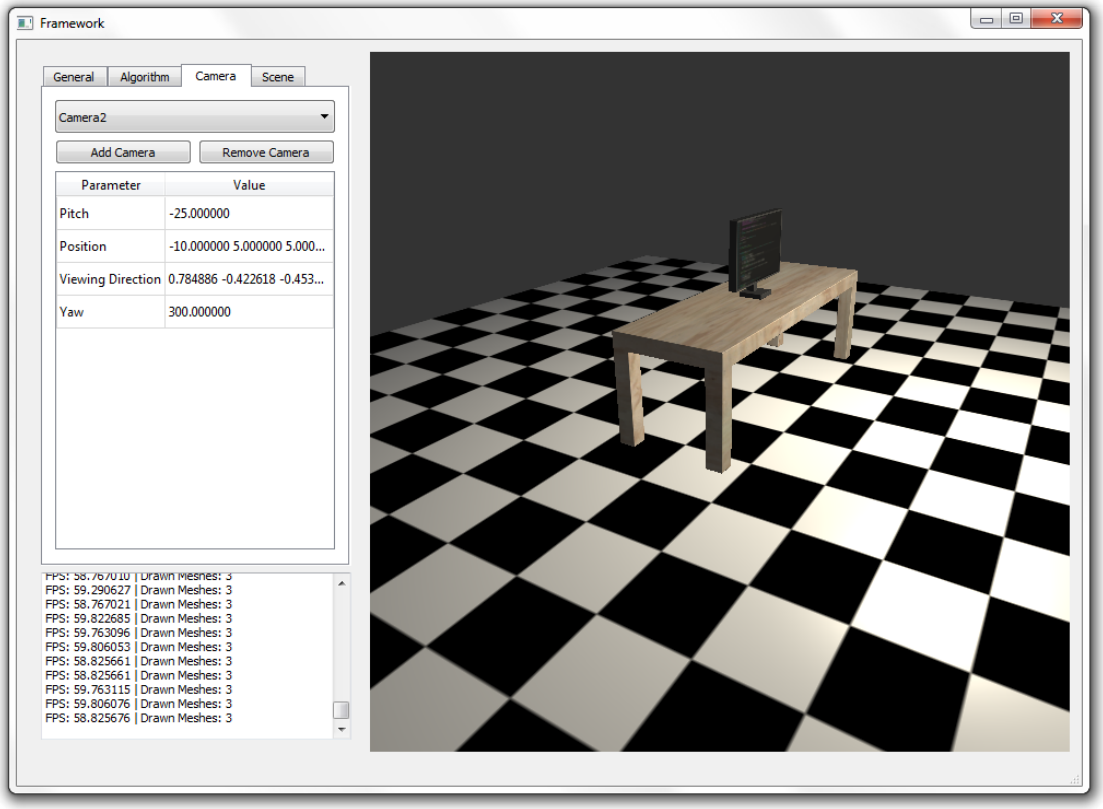

Figure 7.1: The GUI of the framework.

- **"Algorithm"-tab.** algorithms can be switched and configured
- **"Camera"-tab.** provides methods to switch between different cameras. Camera perspectives can also be added or removed.
- **"Scene"-tab.** contains configurations for entities and light sources

OpenGL View. The OpenGL view shows the rendered scene. It is responsible for initializing the OpenGL context and further initiates the initialization of the scene and all available algorithm implementations. Afterwards, a timer is set to repeatedly trigger the rendering loop. To react to mouse inputs properly, the OpenGL view also propagates various mouse events to the controller.

Debug Output. The debug output consists of a textbox that is dynamically filled with log entries. Logs throughout the framework are handled by the `MessageHandler` header class. The `MessageHandler` provides functionality for logging with various severity levels. The logged entries are then formatted and emitted to the debug output.

The GUI components are inherently structured in a tree hierarchy and only the root component — the main window — is accessible via the framework. Therefore, two mechanism were introduced that recursively execute methods in the GUI components.

The injection mechanism injects the framework instance into all components of the GUI. Hence, they have full access to the framework and all connected modules.

The setup mechanism is triggered during the setup of the framework. It is responsible for filling the GUI with needed information about configurable objects throughout the framework.

Every other communication with the GUI components originating from within the framework has to be done over the signal slot mechanism, as described in the upcoming chapter.

7.2 Signal Slot Mechanism

The Qt framework uses event based communication where components can emit and receive signals [Com20b]. A component is also able to react to any signal from another component by executing a slot function, i.e., a callback function. However, a component needs to connect a signal to a slot beforehand for this mechanism to work properly.

In this framework, all GUI components and additionally the controller, acting as the interface for the communication with external components, can emit and receive signals. This mechanism is important for synchronizing parameter changes within the framework with the GUI. Otherwise, there may be outdated values shown in the GUI components.

CHAPTER 8

Results

As addressed in the introduction, the goal of this thesis is the development of an application in order to build a foundation for the comparison of real-time global illumination algorithms. It should be capable of meeting various requirements of different algorithms, would allow switching between implementations of different approaches at runtime and provide the user with a wide range of configuration possibilities.

The application that has been developed aims to meet these requirements. It is separated into several self-contained modules that are accessible over a common connecting link (Chapter 3). Hence, the functionality of modules is available throughout the framework. Commonly used concepts have been supplied to be used in various algorithm implementations (Chapter 5). Therefore, it is possible to extend the framework with new algorithms or new concepts without the reimplementation of already existing functionality.

By creating a uniform base for all algorithms it is possible to unify their handling throughout the application (Chapter 6). Furthermore, the framework is capable to switch between different algorithms at runtime. The uniform base has been designed to have no restrictive requirements to allow the implementation of different algorithms with contrasting preconditions or process flows.

The design of a consistent concept for handling parameters of various objects (Chapter 9.2) in the application brings a great amount of configurability, even at runtime. The scene can be remodeled, camera perspectives can be added, changed, or removed, and algorithm parameters can be adjusted. This concept can furthermore be used for additional algorithms without adaptions in the framework code base.

8.1 Discussion

As stated before, the introduction of parameter maps enhanced the configurability of components of the framework. The `FlexibleValue` class describes a decorator class

for different data types by transforming a value into a string representation. However, the utilization of the `FlexibleValue` itself could have been improved for it is not used in the `getParameter` method or the `setParameter` method. Each object using parameter maps is responsible for casting the formatted string representation back to the data type it needs. Hence, the potential of the `FlexibleValue` being a container for different data types and taking care of all type conversion was not used to its fullest extent. By enforcing the usage of the `FlexibleValue` in the `getParameter` and `setParameter` methods, type conversion can be encapsulated thoroughly in the implementation of the `FlexibleValue`.

A point of criticism may also be the inconsistency of connections for communications between the framework and the GUI components (Chapter 7). The GUI components have full access to the framework whereas the framework either communicates via the signal slot mechanism, the injection mechanism, or the setup mechanism with the GUI components. A better approach might be the enforcement of the signal slot mechanism as the only connection with the framework. This way, the interface for all communications with the GUI components would be the controller which is meant to handle all external inputs.

8.2 Next Steps

There are some extension that are yet to be integrated into the developed application.

For example, to help with the evaluation of an algorithms' interactivity, it might be interesting to implement update mechanisms for entities and light sources (Chapter 4). As of now, the only non-static scene objects are cameras.

Furthermore, the configurability of the scene could be improved by implementing the subsequent loading of additional entities at runtime. It is possible to add or remove light sources at runtime and entities can be disable to exclude them from rendering. However, entities can only be loaded during the initialization of the framework via a scene file.

In the end, the most important next step for this application would be the integration and implementation of various real-time global illumination algorithms. This may reveal weaknesses of the framework, but moreover, some undetected strengths of it might also be disclosed. Furthermore, the framework would fulfill its purpose by revealing the strengths and weaknesses of real-time global illumination algorithms by simplifing their comparison.

CHAPTER 9

Learned Lessons

During my work I was confronted with challenges and concepts that were new to me or needed sophisticated solutions. In the following sections a selection of interesting lessons I learned can be found.

9.1 Circular Dependencies

A circular dependency refers to a mutual dependency of two classes. For example, the framework class uses and depends on the scene class, whereas the scene class also uses functionality of the framework class. To resolve this in C++, there is the need for forward declarations.

```
#include "scene.hpp"

class Framework {
public:
    //constructor, destructor & other public methods
    //would be defined here.
private:
    Scene* _scene;
};
```

Listing 9.1: Example header file without forward declaration.

A simple class declaration for a header file can be found in Listing 9.1. At the beginning of the file are all include statements for declarations of classes that are needed for this class to work. If a class would use another class that was not declared yet, the compilation fails.

This is also the reason why circular dependencies cannot be resolved this way: the compiler would visit a class, find a reference to a mutually dependent class and would

try to compile the referenced class first. However, the referenced class again would have a reference to the initial class so the compiler jumps back to the first visited class and so forth, which would ultimately result in an infinite loop.

```
class Scene;

class Framework {
public:
    //constructor, destructor & other public methods
    //would be defined here.
private:
    Scene* _scene;
};

#include "scene.hpp"
```

Listing 9.2: Example header file with forward declaration.

The modified class using a forward declaration is depicted in Listing 9.2. The include statement was moved to the end of the file and a substitute declaration of the required class is placed before the actual class declaration of this file. This way, the compiler knows that a proper class declaration will follow later during compilation and circular dependencies can be resolved properly.

9.2 Configurability And Parameter Maps

One of the main goals of this work is to provide a great amount of configurability with almost no immutable constants. Therefore, there has been the need for a consistent concept to make parameters modifiable for different objects such as scene objects (Chapter 4) or algorithm implementations (Chapter 6). The concept also has to work at runtime if the user wants to do adjustments via the GUI. Timing issues must be preventable as well as possible. Hence, the owner of the parameter map must have full control whenever a new value for a parameter is set.

To meet these requirements, parameter maps were introduced. Parameter maps store each parameter of a certain object as a key-value pair where the key usually is the name of the parameter as a `string`. However, the values are usually not all of the same data type. This led to the introduction of the `FlexibleValue` class and the `TypeConverter` header class.

`FlexibleValue`. A `FlexibleValue` unifies different data types by storing their type as an enumeration constant and their value as a formatted string representation. It also stores a flag if the value is modifiable. The constructor of the `FlexibleValue` for the `vec2` data type can be found in Listing 9.3.

```
FlexibleValue(vec2 value, bool editable) {
    _type = ValueType::vVEC2;
    _value = TypeConverter::vec22string(value);
```

```
    _editable = editable;
}
```

Listing 9.3: The constructor of the `FlexibleValue` for the `vec2` data type.

`TypeConverter`. The functionality for formatting a value to a string representation is provided by the `TypeConverter`. This header class is also responsible for transforming the string representation back to the initial value.

Currently, both the `FlexibleValue` class and the `TypeConverter` class support the following data types:

1. `boolean`
2. `float`
3. `int`
4. `string`
5. `vec2`
6. `vec3`
7. `vec4`

As stated before, parameter maps should be available to all different kinds of objects. Therefore, there are very few requirements for the usage of parameter maps. An object that uses a parameter map only needs to provide the following three methods:

`string getParameter(string key)`. This method returns the value as formatted `string` for a given key.

`bool setParameter(string key, string value)`. This method should set a value for a given key. The value can be validated before storing it in the map. Additionally, other calculations regarding the new value or requesting further updates may be done here. If the operation was successful, it returns `true`. An implementation of this method can be found in Listing 9.4.

`map<string, FlexibleValue> getParameterMap()`. This returns the map with all parameters that need to be configurable.

```
bool setParameter(string key, string value) {
    if (key == "warmColor") {
        // use TypeConverter for value conversion
        vec3 newColor = TypeConverter().string2vec3(value);

        // validate value
        if (isValidColor(newColor)) {

            // set in parameter map
            _parameterMap[key] =
                new FlexibleValue(newColor, true);

            // request further updates
            requestUpdate();

            return true;
        }
    }

    return false;
}
```

Listing 9.4: Exemplary implementation of the `setParameter` method.

There are currently two different ways parameter maps are used in the framework. In some cases the parameters of an object are already persisted in another way. Therefore, a new parameter map is generated and filled with all parameters of this object ever time it is requested by the `getParameterMap` method. Examples for this would be light sources or the camera.

In contrast, entities and algorithm implementations use permanent parameter maps. The parameters of these objects are only persisted in the map and therefore, the parameter map is generated only once.

9.3 Cuboid Defined By Two Points

A technique that is commonly used within the framework to determine the vertex positions of a cuboid is based on the unit cube vertex positions.

The vertex positions shown in Figure 9.1 could also be considered as weights for an interpolation between a minimum point, being `(0, 0, 0)` in the unit cube, and a maximum point, being `(1, 1, 1)` in the unit cube. The resulting formula for calculating any point of a cuboid is described in Listing 9.5.

```
x = (1 - weight.x) * minimumPoint.x + weight.x * maximumPoint.x
y = (1 - weight.y) * minimumPoint.y + weight.y * maximumPoint.y
z = (1 - weight.z) * minimumPoint.z + weight.z * maximumPoint.z
```

Listing 9.5: Formula for calculating any cuboid vertex position using two points and a weight.

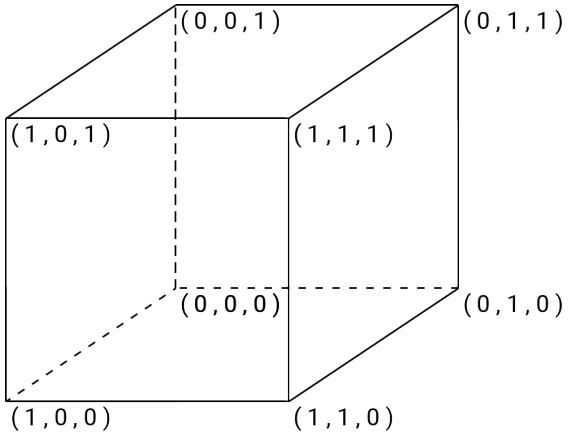

Figure 9.1: A unit cube and its vertex positions.

For example, the point `(0, 1, 1)` of the unit cube could be calculated by using its coordinates as the weight, as shown in Listing 9.6.

```
minimumPoint = (0, 0, 0)
maximumPoint = (1, 1, 1)
weight = (0, 1, 1)

x = (1 - 0) * 0 + 0 * 1
x = 0

y = (1 - 1) * 0 + 1 * 1
y = 1

z = (1 - 1) * 0 + 1 * 1
z = 1
```

Listing 9.6: Interpolation of `(0, 1, 1)`.

This technique is applicable for any two points acting as a minimum point and maximum point, whereas the vertex positions of the unit cube can always be used as weights for the interpolation.

The only vital condition is that the minimum point and the maximum point have to be on opposing ends of the cuboid. A line connecting those two points has to intersect the center point of the cuboid to fulfill this condition, as depicted in Figure 9.2.

9.4 Forward Rendering — Deferred Rendering

Forward Rendering describes a technique using only one render pass to create a frame. All calculations regarding the illumination of the scene are done for each mesh individually. Yet, for many light sources and meshes this results in increased computational costs.

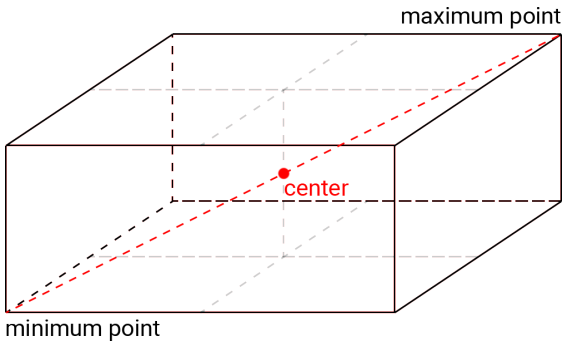

Figure 9.2: The line between minimum point and maximum point intersects the center point of a cuboid.

Some meshes might even be hidden by others, so rendering calculations regarding them might have been unnecessary.

To countermeasure the amount of computational cost, deferred rendering proposes a different approach. It uses at least two render passes to create a frame: one for extracting information from the scene and one for drawing the resulting frame properly.

The first pass renders different information about the scene into images using multiple render targets of a FBO. Each pixel of the resulting images contains data of a fragment belonging to a certain mesh at the corresponding position — e.g., information about surface normals or about surface colors. The data is only available for the meshes visible to the camera, whereas information about hidden surfaces is disregarded. This render pass is usually done by a G-buffer (Chapter 5.3).

The second render pass only uses the resulting images from the first pass to create the frame using proper illumination techniques. All calculations are only done once per pixel and no longer per fragment of multiple entities which can reduce computational costs significantly.

However, transparency presents a problem in deferred rendering. There would be the need for information about the mesh behind a transparent object, but only data about the meshes nearest to the camera is extracted in the first render pass.

APPENDIX A

Appendix

A.1 Polygon Meshes

In 3D rendering, a commonly used data structure to describe complex entities is the polygon mesh. The shape of a polygon mesh is defined by multiple vertices that are connected by numerous polygons (e.g. triangles).

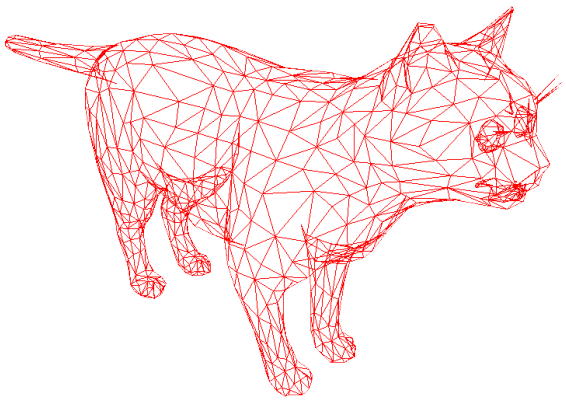

Figure A.1: The polygon mesh of a cat, rendered as wireframe.

Besides the position coordinates, a vertex can also contain additional data needed for rendering. For example, a vertex may also stores a normal vector and texture coordinates. For the rendering of fragments within a polygon, the vertex data of all attached vertices is interpolated.

In OpenGL [Gro20], a VBO holds a defined number of values per vertex and therefore, each buffer can persist a different type of data of a vertex. For example, the position buffer contains three values for each vertex whereas the texture buffer contains only two texture coordinates per vertex.

To simplify the handling of mesh data even further, all VBOs can be collectively stored in a VAO. Whenever the mesh needs to be drawn, only the VAO has to be bound to transfer the mesh data correctly to the shader program.

A.2 Configuration File Syntax

The configuration file contains most global constants which can be altered within the file. It is loaded only once at the start of the application, any further modification within the file are disregarded. The file named "config.txt" is located in the "resources"-folder.

Generally, each line has to start with a certain indicator following some value(s). Values are separated by whitespace characters. Each parameter can only be set once in the configuration file. Comments are indicated by a # (hash) and are skipped.

The following parameters can be set:

Moving Speed. This parameter defines how sensitive the camera reacts to keyboard input.
Syntax: movingspeed [`float` value]

Mouse Sensitivity. This parameter defines how sensitive the camera reacts to mouse movement.
Syntax: mousesensitivity [`float` value]

Near/Far Planes. This parameter defines the distance of the near plane and the far plane to the camera. Entities in front of the near plane or behind the far plane are not considered for rendering.
Syntax: nearfar [`float` near plane distance] [`float` far plane distance]

Path To Scene File. A scene file describes all scene objects and their placement within a scene. Its path needs to be in relation to the application-executable.
Syntax: scenefile [`string` path]

Vertical Field Of View. This parameter defines the cameras' aperture angle in degrees in vertical direction.
Syntax: fieldofview_y [`float` value]

```
# This is an example for a configuration file.
scenefile resources\\scene_studyroom.txt
```

```
movingspeed 5.0
mousesensitivity 0.25
fieldofview_y 60.0
nearfar 0.01  100.0
```

Listing A.1: Exemplary Configuration File

A.3 Scene File Syntax

A scene file is used to define all scene objects and their placement within a scene. It can be set in the configuration file and is only loaded once at the start of the application.

Generally, each line describes one scene object and has to start with a certain indicator following some value(s). Values are separated by whitespace characters. Comments are indicated by a # (hash) and are skipped. Multiple occurences of each scene object type are allowed. A given name might be altered if it is not unique within the scene.

In the following, `vec3` is defined as three float-values, separated by whitespace characters. A `boolean` is either `0` (false) or `1` (true).

The following scene object types can be set:

Mesh Entities. Mesh Entities describe entities loaded from a file. Its path needs to be in relation to the application-executable. If a file contains multiple meshes, they can either be loaded as individual entites or one combined entity.
Syntax: m [`string` name] [`string` path] [`boolean` load as individual entities] [`vec3` translation] [`vec3` rotation] [`float` scaling factor]

Cuboid Entities. Cuboid Entities describe cuboids defined by two points.
Syntax: b [`string` name] [`vec3` minimum point] [`vec3` maximum point] [`vec3` translation] [`vec3` rotation] [`float` scaling factor]

Ambient Light. Ambient Light describes omni-directional light. The color of a light should be in the interval between 0.0 and 1.0 and will otherwise be cut off.
Syntax: a [`string` name] [`vec3` color]

Directional Light. Directional Light describes parallel light rays coming from one direction. The color of a light should be in the interval between 0.0 and 1.0 and will otherwise be cut off.
Syntax: d [`string` name] [`vec3` color] [`vec3` direction]

Point Light. Point Light describes light rays emitting from one point. The color of a light should be in the interval between 0.0 and 1.0 and will otherwise be cut off.
Coefficients for constant, linear, and quadratic attenuation are combined in one vector.
Syntax: p [`string` name] [`vec3` color] [`vec3` position] [`vec3` attentuation]

Spot Light. Spot Light describes light rays emitting from one point in a certain direction and aperture. The color of a light should be in the interval between 0.0 and 1.0 and will otherwise be cut off.
Coefficients for constant, linear, and quadratic attenuation are combined in one vector.
Syntax: s [`string` name] [`vec3` color] [`vec3` position] [`vec3` attentuation] [`vec3` direction] [`float` aperture in degrees]

Camera. A camera is an seeing eye within the scene.
Syntax: c [`string` name] [`vec3` position] [`float` yaw] [`float` pitch]

```
# This is an example for a scene file.
m  Table resources\\table.obj  0  0.0 0.0 0.0  0.0 0.0 0.0  1.0
b  Floor 0.0 -0.1 0.0  5.0 0.0 5.0  0.0 0.0 0.0  0.0 0.0 0.0  1.0
a  AmbientLight 0.2 0.2 0.2
d  DirectionalLight 0.5 0.2 0.2  0.5 -1.0 0.5
p  PointLight 0.96 0.92 0.82  0.0 5.0 -5.0  1.0 0.1 0.1
s  Spot 1.0 0.0 0.0  0.0 5.0 5.0  1.0 0.1 0.1  0.0 -1.0 0.0  50.0
c  Camera  0.0 5.0 10.0  0.0  -10.0
```

Listing A.2: Exemplary Scene File

A.4 Key Bindings And Mouse Controls

- **W.** Move camera forwards.
- **A.** Move camera to the left.
- **S.** Move camera backwards.
- **D.** Move camera to the right.
- **R.** Move camera up.
- **F.** Move camera down.
- **Q.** Tilt camera to the left.
- **E.** Tilt camera to the right.
- **Left Mouse Button + Mouse Moving.** Tilt camera in any direction.

List of Figures

List of Listings

Acronyms

FBO Framebuffer Object

FPS frames per second

GLSL OpenGL Shading Language

GUI graphical user interface

VAO Vertex Array Object

VBO Vertex Buffer Object

Bibliography

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing: A preview. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 207–207, New York, NY, USA, 2011. ACM.

[Com20a] The Qt Company. Qt framework. `https://www.qt.io/`, 2020. [Online; accessed 2020-03-01].

[Com20b] The Qt Company. Signals & slots. `https://doc.qt.io/qt-5/signalsandslots.html`, 2020. [Online; accessed 2020-03-01].

[Gam20] Epic Games. Unreal engine. `https://www.unrealengine.com/en-US/`, 2020. [Online; accessed 2020-11-28].

[GGSC98] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 447–452, New York, NY, USA, 1998. Association for Computing Machinery.

[Gro20] The Khronos Group. Opengl. `https://www.opengl.org/`, 2020. [Online; accessed 2020-03-01].

[Jak10] Wenzel Jakob. Mitsuba renderer. `http://www.mitsuba-renderer.org`, 2010. [Online; accessed 2020-03-01].

[Kap09] Anton Kaplanyan. Light propagation volumes in cryengine 3. *ACM SIGGRAPH Courses*, 7:2, 2009.

[Kim18] Kulling Kim. Open asset import library. `http://www.assimp.org`, 2018. [Online; accessed 2020-03-01].

[MHB+00] Michael Meißner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, VVS '00, pages 81–90, New York, NY, USA, 2000. ACM.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[SL17] Ari Silvennoinen and Jaakko Lehtinen. Real-time global illumination by precomputed local reconstruction from sparse radiance probes. *ACM Trans. Graph.*, 36(6), November 2017.

[SRS14] M. Sugihara, R. Rauwendaal, and M. Salvi. Layered reflective shadow maps for voxel-based indirect illumination. In *Proceedings of High Performance Graphics*, HPG '14, pages 117–125, Goslar Germany, Germany, 2014. Eurographics Association.

[Tec20] Unity Technologies. Unity. `https://unity.com/`, 2020. [Online; accessed 2020-11-28].

[THB+90] Ulf Tiede, Karl Heinz Höhne, Michael Bomans, Andreas Pommert, Martin Riemer, and Gunnar Wiebecke. Investigation of medical 3d-rendering algorithms. *IEEE Computer Graphics and Applications*, 10:41–53, 1990.