Entwicklung einer Bibliothek zur Darstellung dreidimensionaler Szenen

Diplomarbeit
zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Thomas Kranz

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter: Dipl.-Inf. Martin Schumann
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2011
Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ja  Nein

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.  □  □

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.  □  □

.................................................................
(Ort, Datum) .................................................

.................................................................
(Unterschrift)
DIESE ARBEIT LIEGT IN ENGLISCHER SPRACHE VOR.

THE FOLLOWING THESIS IS WRITTEN IN ENGLISH.
Acknowledgements

This work is dedicated to my family and to Katharina.

I want to thank Prof. Dr. Stefan Müller for his flexibility, support and constructive criticism over the course of this work.

Also, my gratitude and respect go out to David H. Eberly, Oliver Mattausch and Morgan McGuire, for sharing their professional experience and insight - even while being on vacation.
Contents

1 Introduction ........................................ 1
   1.1 Motivation ...................................... 2
   1.2 Structure ...................................... 3

2 Current Technology .................................. 4
   2.1 Graphics Hardware ................................ 4
   2.2 The OpenGL 3 Core Profile ....................... 5
   2.3 Scene Management Approaches .................... 6

3 MiReLi: A 3D Rendering Library .................... 10
   3.1 Requirements .................................... 10
   3.2 Constraints .................................... 13
   3.3 System Overview ................................ 14
   3.4 The Render Manager ............................... 16
      3.4.1 Configuration ................................. 16
      3.4.2 State Management ............................ 19
      3.4.3 Transformations .............................. 21
      3.4.4 Deferred Lighting ............................ 23
      3.4.5 Post-Processing .............................. 27
   3.5 The Scene Manager ................................ 28
      3.5.1 Loading Scenes ............................... 28
      3.5.2 Light Source Buffers ......................... 32
      3.5.3 Interaction of SceneManager and RenderManager 35

4 Scene Management .................................. 36

5 Shaders and Materials ............................... 41
   5.1 Shaders ........................................ 42
   5.2 Materials ...................................... 46

6 Results ............................................. 51
   6.1 Rendering Performance ............................ 51
   6.2 Visual Quality .................................. 55
   6.3 Integrating MiReLi ............................... 58

7 Conclusion and Future Work ......................... 59


1 Introduction

Computer graphics has been an ever growing discipline of computer science for decades. Especially one subset of computer graphics, namely real-time rendering, has found countless uses in a multitude of applications - be it computer aided design, video and educational games or scientific visualization. In recent years there have been profound changes, both in regard to the hardware and programming interfaces, which empower developers to create increasingly faster rendering software and produce visually more compelling imagery.

Although similar to previous generations, the architecture of graphics processing units (GPU) has shifted towards potentially more efficient mechanisms and paradigms. Abstraction layers like OpenGL[31] and Direct3D[12] reflect such changes with continuous revisions to their specifications and adapt to evolving hardware features. From time to time, such alterations allow for a revision of algorithms and practices employed in the development of graphics applications.

Real-time rendering has always involved management of the objects in the scene which is to be drawn. Over the years, many algorithms have emerged that enable software to fulfill different tasks. Some aim at runtime performance, others want to provide comfortable user interaction. Most of the time, however, the principal goal is to achieve a good compromise between speed and quality - a trade-off which is and has been a frequent companion of real-time rendering. A very popular notion of scene management has been implemented in what is frequently called a scene graph. The meaning of this word is, however, anything but unique and mostly subjective. Eberly points out that for him a scene graph “is just an abstract graph of objects”, suggesting no concrete implementation[13]. One approach for the realization of a scene graph, and therefore one possible meaning of the word, has been a tree-like structure which contains nodes representing the objects in the scene. Additionally, there may be several other types of nodes augmenting the properties of objects with transformations, material properties, geometry, and basically any kind of information that may be needed[9],[11],[14],[13]. Eberly calls this data structure an uber scene graph, indicating the desire to incorporate as much data as possible in a single data structure[13].

The problem is that most ideas of this approach were conceived at times when the facilities of graphics hardware differed substantially from what it offers today. There are a great many methods which do not necessarily mandate the presence of a single tree-like structure. Another general flaw in this design is that the implementation grows to proportions where it becomes hard to maintain and optimize. Another observation that can be made is
that current abstraction layers have made options available to aggregate render state into dedicated data structures, thus reducing the state management overhead and simplifying the implementation of renderers. The amount of processing that needs to be done to render the scene contents is constantly reduced, which further alleviates many tasks previously conducted on the application side.

1.1 Motivation

The major incentives for this work have been rapidly evolving graphics hardware and abstraction layers, new concepts in graphics programming, and a general interest in designing a rendering software. Another driving factor was that the evolution of OpenGL and the implications of this process have not yet been fully adopted in academia, and so keeping up with state-of-the-art programming techniques has become an individual responsibility.

The possibilities that current GPUs offer and the advent of major revisions to the OpenGL specification sparked great interest in how a renderer could be implemented today. Furthermore, its integration into a framework which augments the system with facilities to author and dynamically load scene descriptions was subject to curiosity. As a result, there was strong desire for creating a reusable, extensible, efficient, and modern piece of software which would be capable of handling these tasks. Since some of the traditional approaches in real-time rendering have or will in the future become more or less obsolete, questions about alternatives and methods of combining them into a complete system arose.

This work intends to explore ways to create such a software, using approaches that map well to current technology and abstraction layers. The goal is to conceive a library based on OpenGL, which provides a sufficient subset of functionality that is common in current rendering engines and provides a solid, easy-to-use basis for application development. Methods of scene management will be discussed and criticism of the aforementioned *uber scene graph* will be established. The discussion is based on references in the literature, personal correspondence, and experiences from developers working in the graphics industry. Also, a number of recent rendering effects will be inspected and their potential for integration into the system will be evaluated. On the side of functionality, the library is to provide the means necessary to

- allow simple and safe integration into an application
- import and organize resources and assets
- define the contents of a scene
- exploit current hardware features and graphics algorithms
Although it is considered impossible to raise the resulting quality to the level of industry-strength products, it is supposed to establish a solid basis for further development and research.

1.2 Structure

The following chapters will explore in detail the architecture of the core system, the scene management and material concept. Chapter 2 offers a brief review of current graphics hardware and abstraction layers, as well as a more detailed look at some current systems. It will also examine some criticism of the above mentioned, traditional scene graph approach.

Chapter 3 introduces the core system, especially design choices for core classes, and shows the interconnections of the different components. Chapter 4 dissects the scene management system, including a discussion of ways to improve performance with different, current algorithms. Chapter 5 discusses the shader and material system in detail. Chapter 6 evaluates the performance characteristics and visual quality of the renderer. Chapter 7 summarizes the efforts made in this work, offers a critical review of what has been achieved, and draws a final conclusions. It also provides an outlook on what will come next in the development of the library.

The reader is expected to be familiar with OpenGL although at times, important concepts will be introduces to explain a certain feature.
2 Current Technology

This chapter gives a brief overview of the current state of computer graphics, especially graphics hardware and common techniques in graphics programming. It will also cover a few concrete examples of recent real-world contributions to real-time rendering and a discussion of current trends in scene management techniques.

2.1 Graphics Hardware

Over the past years graphics hardware has evolved from providing a complex, partly configurable fixed-function pipeline to exposing parts of it as programmable stages, which allow for a vast number of algorithms to be processed directly on the GPU. Programs executed during these stages are commonly called shaders. Earlier GPUs already exposed some support for shader programming. For instance, the NVidia GeForce 3 architecture[21] provided a programmable vertex and fragment stage. Still, the possibilities and functionality were limited. In current Direct3D 10 capable hardware, such as the G80 and later architectures[26], dedicated pipelines for vertex and fragment processing have been replaced by what is known as the unified shader architecture. This approach allows for different types of shader programs to execute on every shader processor. With this hardware generation, there are three distinct stages of the pipeline which are fully programmable: the vertex, geometry and fragment shader. Conceptually, these shaders share a common programming model, known as the common-shader core, meaning that they can be programmed very similarly - with certain differences depending on the specific shader stage[9]. For instance, the OpenGL Shading Language (GLSL)[20] provides built-in variables which are only accessible from, for example, the fragment shader. The unified shader architecture has been designed to adequately fit the common-shader core.

The basic approach of rendering is still the same: the user defines a number of primitives, sets the appropriate render state and submits the geometry to the GPU using dedicated draw calls. The major difference between current and former GPUs and abstractions layers is the buffer centric view, and the complete absence of immediate mode constructs. If possible, an application will store most data used in the rendering process directly in graphics memory. The rendering process is exclusively steered by shaders which can be granted access to the contents of said memory. A number of techniques need support for off-screen rendering to dedicated areas of memory, which in the case of OpenGL is provided through framebuffer objects (FBO). Image-space effects can reuse such off-screen buffers inside dedicated shaders and alter their contents to produce the final image.
To be able to use these features, multiple C-like shading languages exist in addition to the abstraction layers. As mentioned above, GLSL is the representative used in conjunction with OpenGL. Others include the CG Language by NVIDIA and the High Level Shading Language by Microsoft\cite{16,23}. They expose functions which can be efficiently evaluated on the GPU and facilities to draw not only to the default framebuffer but to specifically selected off-screen buffers. This enables exporting different kinds of data from within a shader in a user-defined fashion and is the basis for a wide variety of effects.

2.2 The OpenGL 3 Core Profile

A few years ago, the OpenGL architecture review board (ARB) decided to release an overhauled version of the specification which was to carry the version number 3.0 in conjunction with an updated version of GLSL carrying the version 1.30. As had been practice before, a number of extensions were due to promotion to the core to reflect current hardware features, but the real difference between this and previous releases was a deprecation model which marked a large part of the former specification to be subject to future removal. However, the deprecated features remained functional in all implementations. In OpenGL 3.1 they actually vanished. Versions 3.2 introduced a separation of the specification into two variants: a core and a compatibility profile. Core profile specifications only contain a mandatory, forward-compatible subset of their compatibility profile counterparts, including GLSL versions 1.30 and later, whereas the compatibility profile is backwards-compatible with all former OpenGL versions - however, implementing the latter is optional. Major hardware vendors have decided to provide drivers offering both profiles to guarantee the functionality of legacy software.

When designing and implementing a new product with clearly specified hardware constraints, using the compatibility profile and its functions is not necessary. More accurately, due to increasingly unified concepts and a reduced state subset, using the core profile may result in less, more maintainable code and additionally eases future transitions to more recent major revisions of OpenGL. It needs to be stressed, however, that software based on the core profile does not necessarily perform better.

The library conceived during the development of this work is based on the OpenGL 3.3 core profile and GLSL 3.30, since it is written entirely from the ground up and is intentionally targeting only capable graphics accelerators.
The reason is that most of the mechanisms require more features and processing power than legacy hardware exposes. Another premise is that all functions provided by core profiles are mandated to work in conforming implementations across hardware from many different vendors, eradicating the necessity to query supported extensions and to implement fall back render paths should any of them not be supported.

2.3 Scene Management Approaches

As already stated in the introduction this section will discuss what Eberly calls *uber scene graph* approach of handling three dimensional scenes. Although having been popular for a long period of time, scene graphs in the above sense revolved around a single, principal idea: representing *any type of scene* using a single, *all-encompassing* data structure. Henceforth “scene graph” will be used to denote the above mentioned data structure unless indicated otherwise.

In principle, the theory behind many of the individual concepts is well intended. This includes hierarchical representations, managing shared geometry, and render state, determining occlusion, selecting different levels of detail, and so on. For instance, rejecting large parts of a scene in a hierarchical fashion can be multiple times faster than a brute force approach - depending on the number of objects to test. Eberly states that in general one of the most important goals when designing a rendering system has been to improve its performance; and this has also been an important aspect of scene graphs.

Still, choosing a data structure is not the only thought which is necessary today to achieve good results. Bar-Zeev and Eberly[13, 11] mention the vastly different facilities of graphics hardware which was available when many of the earlier scene graph systems were conceived. There were only very limited graphics resources and the processing speed was multiple orders of magnitude below current GPUs. There were also very limited possibilities for programmable shading, let alone sophisticated image-space effects. Also, the main concern when designing such a system several years ago was reducing the set of geometry. Although possible, Eberly merely gives an introduction of how programmable shaders can be integrated with a scene graph but does not mention a completely shader based rendering pipeline[14]. What was published in 2004 is now described by the author as “outdated”[13].

There were also a number of problems with established scene graphs, where optimizing for one purpose may render other tasks sub-optimal. Zeev mentions a conflict, where sorting the scene contents by state increases the amount of processed geometry[11].

6
Wilson and Eberly point out that incorporating skinned animation into scene graphs is possible, but restrict their suggestions to only concatenating the transformations involved [13, 33]. Eberly also brings up portal systems, either as a stand-alone solution or integrated with a scene graph, which have traditionally been used to quickly reject occluded parts in a scene. In earlier years these have been essential to achieve good rendering performance, and although portals are still used in contemporary systems, Eberly remarks that today it sometime “is faster just to let the hardware do the draw - the CPU costs for portals is the bottleneck” [13]. If aforementioned portal system was an integral part of the rendering process, the implementor of the scene graph system would need to optimize for cases where it limits the rendering performance. This is substantiated by tests conducted prior to writing this thesis, where a low-budget mobile GPU was used to render three million triangles with a simple fragment shader at approximately 30 frames per second without any scene management involved; rendering the same amount of polygons only to a depth buffer will even be multiple times faster on current hardware [25]. This indicates that the cost of processing primitives has declined and trying to save every piece of unnecessary geometry has lost its importance.

Furthermore, with current algorithms additional geometry is sometimes drawn on purpose and may still increase performance, is is the case with hardware occlusion queries which will be discussed later. On the other hand, even scenes with very low geometric complexity may perform unacceptably, when using a number of shaders on objects or as image-space effects which incur tremendous processing costs - a problem which is only present with current GPUs. Since the fragment stage is now fully programmable, developing efficient fragment shaders, and using Early Z [28, 25] culling if necessary, has become as important as managing scene geometry. Even the efforts of a thoughtfully designed scene management system can be useless if shader complexity is too high for the executing hardware.

Another instance where integration into a scene graph, or any hierarchy for that matter, is not necessary today is a terrain rendering system. In [13], terrain pages were treated as nodes in the hierarchy and were processed by the CPU before rendering. Current, terrain algorithms like geometry clipmaps do only rely on very little application intervention and do most of the work directly on the GPU [10]. This is a good example of a subsystem that can be treated independently from the rest of the scene geometry. Also, processing the complete terrain first may be beneficial for determining the visibility of other objects in the scene, since it generally acts as a large occluder in the scene [32]. This may, however, be less efficient if it mandates traversing a hierarchy to be able to render the terrain.
A common problem that has been present in real-time graphics is dynamic lighting. Early hardware only offered a certain number of active dynamic light sources in a scene. To counter this limitation, systems tried to simulate more light sources by altering the properties of the available quantity to light each object in the scene separately. This is clearly a scene management task. Although this could solve the problem to a certain extent, dynamic lighting with many light sources was not feasible on earlier hardware and scene graphs would rely on different combinations of dynamic and pre-computed lighting information. Today there are at least two common approaches to simulating hundreds of small local light sources at high frame rates. One in particular, the light pre-pass, will be discussed later and shows that only a limited involvement of the scene management system is necessary to produce convincing lighting effects.

As an example of the different notions of scene management, the following to examples show two current possible solutions.

To overcome the possibly detrimental implications of a single scene management system, the open-source rendering engine Ogre3D[3] provides a selection of different scene managers to accommodate different use cases[7]. Ogre3D is still a huge scene graph system but through this selection of different underlying data structures, it is more flexible than traditional systems. By default, the engine uses an octree, but there are also managers dedicated to processing large outdoor environments. One complaint one could establish is that an application is bound to using the specified scene manager regardless of the organization of the scene. As a consequence, using a terrain manager will organize other types of objects using the same data structure employed for the landscapes; which may not be optimal depending on the spatial organization of said objects. On the other hand, an octree may not perform well when trying to simulate a large terrain database. This reflects the potential trade-offs a developer has to accept when mandated to use a homogeneous data structure.

An approach which wants to completely move traditional CPU based scene management tasks to the GPU is the March of the Froblins technology demonstration presented by Shopf et al.[32]. View-frustum and occlusion culling for instance, animated characters is done entirely using shaders. Contrary to this work, occlusion culling is done with a method leveraging the hierarchical z-buffer, which has already been proposed in 1993 by Greene et al.[17], but was not feasible due to lacking hardware support until recently, and an octree. The authors mentions that since computing animations is done entirely on the GPU, managing the characters from the application “would require a costly readback operation”.

8
Aside from determining the visibility, the selection of the level of detail is done based on the distance from the camera and stream-out operations to produce buffers holding the instances of a specific level.

As a conclusion, one will agree with Bar-Zeev who stated that “it is probably impossible to find a single perfect organization […].” for any type of scene[11]. The logical consequence is simple and formulated clearly by Gregory, who gives a selection of different types of video games and from which data structure they may benefit[18]: “Which data structure to select for your game will depend upon the nature of the scenes you expect to be rendering. To make the choice wisely, you must have a clear understanding of what is required […] for your particular game.” Gregory ends with suggesting that "your choice of scene graph\(^1\) should be based on hard data obtained by actually measuring the performance of your rendering engine", clearly ruling out a single data structure for any purpose.

Rendering systems have to be able to change and adapt to evolving hardware. The above considerations indicate that sticking to one single approach for years may not be a wise choice, and revisiting at least parts of a system may be advisable to keep up with contemporary approaches. Although a software can not be rewritten after every project it is involved in, it can and should be designed so that changes to the architecture reflect current trends in real-time graphics and still remain functional for a longer period of time, suggesting a choice of concepts which can be integrated in modular fashion and optimized and exchanged independently.

Regarding the above observations, it has been decided to base this work on a variety of techniques to speed up the rendering process. Using an octree is beneficial for a variety of scenes and it also matches the occlusion culling algorithm the library implements. View-frustum culling and a discrete level of detail approach help to further reduce the triangle count. To keep the cost of simulating lighting reasonable while maintaining a large number of dynamic light sources, the aforementioned light pre-pass is employed. State sorting and tracking of different state variables further reduce the costs by assuring that only necessary changes are conducted. The following chapters explain each of the mechanisms in greater detail and observe the resulting properties of the system.

\(^1\)Like Eberly, Gregory is referring to a scene graph as an abstract way of scene organization, meaning "graph like data structures" with no clear indication as to how such a structure would be implemented.
3 MiReLi: A 3D Rendering Library

This part explores in great detail the requirements, concepts and concrete implementation of the Minimal Rendering Library, in short MiReLi. Several sections discuss the requirements, constraints, design and implementation of the library in detail. Especially the two most important components will receive attention.

3.1 Requirements

This sections formulates concrete requirements and constraints, which have to be fulfilled by MiReLi. The following enumeration is divided into several categories, each of which address distinct parts of the system. Each category states a list requirements and gives brief information about how compliance is achieved.

1. General Requirements

- Cross-Platform Support
  MiReLi is supposed to be usable on multiple platforms without much consideration. To ensure this, dependencies have been selected which inherently support cross-platform development.

  In general, the system itself is based on standard C++, C++ Tech Report 1 (TR1) and the Boost\[1\] library. Dependencies may be written in C, but this has no consequences for portability.

  OpenGL is chosen as the graphics hardware abstraction layer. This enables MiReLi to leverage current hardware features across multiple platforms, as long as capable drivers are installed on the executing system. To further reduce platform dependence, applications are required to handle OpenGL context creation on the target platform. As soon as a valid context is present, all features of the library may be used.

  \textbf{Note:} The technology demonstrations are based on Qt 4.7\[4\] which provides means to create and configure a context, a default framebuffer, and windows to display the buffer contents.

  Most of the resources supported by the library, like scene and material descriptions, are based on the XML format. To read, validate, and write these resources, TinyXML\[5\] and libxml2\[6\] are used.
Importing images of a wide variety of types is handled by the DevIL\cite{2} image manipulation library.

- **Usability**
  Including the library into an application is supposed to work straight-forwardly. The object oriented interface, automated resource handling, and flexible XML based script support guarantee easy integration and offer the possibility to edit properties of scenes and materials without any changes to existing code.

- **Safety and Reliability**
  The library aims at being as fault tolerant as possible. Each script comes with distinct specification. Runtime validation based on RelaxNG ensures that invalid scripts, i.e. files which do not conform to the specification, are not loaded and the developer is notified accordingly. Another important part is memory management. The library takes care of all allocations and deallocations. By keeping memory requests as local as possible and by using smart pointers memory leaks are avoided. To guarantee the functionality of certain parts of the system, some shader source codes, which will be denoted the immutable shader library, are directly compiled into the executable and properly assigned at runtime. This is to ensure that a user error does not lead to runtime memory corruption.

- **Efficiency**
  The system relies on proven containers and algorithms from the C++ standard library and tries to off-load as much computational work as possible to the GPU.

2. Rendering Requirements

- **A Configurable Rendering Process**
  MiReLi uses a very modular system. This means that almost all features can be toggled at runtime or manifested in a configuration file. This way, the behavior of the renderer can be tailored to the capabilities of the executing platform.
• **Flexible and Efficient Scene Management**
  The library relies on a multitude of algorithms to provide stable and high frame rates. This includes fast visibility determination, automatic level of detail selection, geometry instancing, and state sorting. Currently, the main data structure to achieve this is the *octree*, which provides spatial information about the scene contents, allowing for early rejection of geometry and implementational simplicity. No visibility pre-processing is needed, since the library relies on runtime visibility determination using view-frustum and occlusion culling and discrete level-of-detail selection.

• **Fully Shader-Based Geometry and Material System**
  MiReLi's renderer uses OpenGL as its backend and only uses functions defined in the core profile specification. Since large parts of the fixed-function pipeline have been removed in current OpenGL implementations, the only means of processing geometry are shaders. Imported geometric data and attributes are held in video memory and can be accessed from within a shader through predefined interfaces. To offer adequate support for a variety of current rendering techniques, MiReLi comes with a XML scriptable material system which allows easy integration of potentially arbitrary shader code into the rendering process. Incorporating data, such as shader constants or multiple textures, is simple and straightforward.

• **Support for Image-Based Effects**
  In recent years, a number of image-based effects have entered real-time computer graphics. One prominent example is deferred shading, which can bring notable performance gains in scene with a large number of light sources, but there are many others which can be applied. MiReLi divides the rendering process into several parts. One of the last steps applies user assigned shaders on the current contents of the framebuffer to produce the final image - either in a single step or in a multi-pass fashion. Also, the setup of this buffer can be achieved by assigning a shader tailored to the needs of the developer.
• **Support for Dynamic Lighting and Shadows**

Real-time rendering has explored countless ways of simulating the interaction of light and material in three-dimensional scenes. As the power of graphics hardware continues to grow, dynamically illuminated scenes with vast number of light source are becoming the state of the art. In combination with image-based techniques and physically based models, the realism increases steadily. A fairly recent technique has been described by Engel which, in contrast to deferred shading, enables deferred lighting computations and multiple materials. Dynamic shadows are supported through configurable and automatic shadow map generation.

### 3.2 Constraints

Due to the time frame in which this work has been conceived, there are limitations which have to be taken into consideration. They apply to the library and all technology demonstrations and include, but are not restricted to, the following:

- **Currently the library handles static geometry.** However, every scene object may contain a object-to-world transformation which can be altered. This way, the orientation and position may be changed by the application at runtime.

- **Not all parts of the OpenGL Shading language are mapped to the material system.** For instance, user defined matrices are currently not supported. For the test cases, the predefined matrices are sufficient.

- **Only a subset of the OpenGL specification is exposed.**

- **Memory management does not include dynamic unloading of resources.** This means that all scenes must fit into the memory of the executing graphics hardware. Here it is assumed that the size of the machine’s RAM is equal to or greater than the size of video memory. Still, by using lazy loading during initialization, only actually requested resources are loaded into system.

- **MiReLi has not yet been fit for compilation and use as a dynamic library on Windows systems.** Only static linking is currently supported.
3.3 System Overview

This section lays out the general structure of MiReLi and explains the connection and function of the five main sub-systems. Each one is represented by a class denoted as follows:

- EngineManager
- RenderManager
- SceneManager
- ResourceManager
- EngineLog

The following diagram visualizes the interconnections of the above components:

![Diagram of MiReLi's Main Components](image)

**Figure 1:** MiReLi’s Main Components
The **EngineManager** is responsible for the initialization and shutdown of the other managers. It also provides functions which return references to every nested component. This way, dependencies holding a reference to the EngineManager can communicate with one another. Convenience functions, which notify the manager’s dependencies, can be invoked by an application. For instance, if the size of the rendering window changes, the application can invoke `EngineManager::windowResizeNotify()` to request the `RenderManager` to resize its current viewport and off-screen buffers. Another example would be logging. The manager provides a function `EngineManager::message()` delegating text to the `EngineLog`.

To control the rendering process and associated render state, the **RenderManager** comes into play. This class manages higher-level information about a subset of the OpenGL pipeline state, for example the active texture unit. Other components may set and query this information which helps to reduce redundant state changes. Rendering itself is modularized depending on the configuration of the RenderManager, separating tasks which need to be completed every frame from optional steps. Also, the manager maintains a matrix stack. Section 3.4 examines this component and the rendering process in more detail.

Management of all objects in the current scene, like meshes and light sources, is handled by the **SceneManager**. Before the RenderManager can request drawing, certain steps need to be taken to improve performance. This includes view-frustum and occlusion culling, level of detail selection, and state sorting. Since this process is quite involved depending on the render configuration, the manager exposes wrapper functions which encapsulate these sub-tasks. Additionally, there are functions for updating the light source buffers and loading and unloading of scenes. The SceneManager is the topic of section 3.5.

Resources need to be imported into the system at runtime in order to be able to render a scene. This is the responsibility of the **ResourceManager**. It basically acts as a cache for use by the Render- and SceneManager or an application and employs a lazy approach to loading files. During initialization, the manager will only determine the paths of files in the specified resource directories. Since every resource is uniquely identified by a name, files and identifier can be directly associated. If a resource is requested by another component or an application, the manager will check if it is already in the system, in which case a pointer will be returned, or load the associated file. Resources which are described using dedicated scripts are processed by specific class to reduce complexity and increase maintainability.
The *EngineLog* is mostly used to display messages in an organized fashion. It is employed throughout the library to show system information, which it automatically gathers on initialization and which can be queried at runtime, debugging information as well as warnings and errors. Aside from passing text to the standard output, it also writes to a log file at a specified location.

Now follows a more detailed description of the two major components responsible for steering rendering and managing scene contents.

### 3.4 The Render Manager

This class is essential for configuring and steering the rendering process. Some of the most important tasks the RenderManager is responsible for can be summarized to:

- trigger and control the rendering process
- change a subset of the OpenGL state without incurring redundant calls
- manage and provide access to transformations
- handle deferred lighting, including all light source related data
- handle post-processing effects

To deliver a more detailed picture of the manager’s purpose, these points will subsequently be explored in order.

#### 3.4.1 Configuration

Almost all major operations conducted by the RenderManager can be configured by the user. This includes the execution itself and also the concrete shading method used during the application of a certain step. For instance, post-processing effects can be executed in multiple passes if the user decides to add more than one shader to the render configuration. Another example is the selection of the lighting solution for each type of light source. Other steps are not only dependent on the configuration, but also on their existence in or absence from the scene description - therefore the RenderManager will double-check if a skybox is present *and* if it is supposed to be rendered. The following diagram visualizes the process:
Figure 2: Flow of the Rendering Process
As can be seen, there are five conditions that are evaluated per frame. Still, should an operation be executed, then there are usually more options for the user to influence its details. A good example is the selection of different resolutions for the shadow maps and the selection of shadow-casting light sources. Changes to the configuration can be made at runtime.

The implementation is straight-forward: A structure called RenderConfig is initialized with values from the user provided configuration file. At runtime, alterations can be made through corresponding function calls. The layout of the structure is shown below:

```cpp
struct RenderConfig {
    bool render_shadows;
    bool render_skybox;
    bool occlusion_culling;
    bool use_chc;

    std::string directional_light_shader;
    std::string point_light_shader;
    std::string spot_light_shader;
    std::vector<std::string> post_effect_shaders;

    LODRange lod_range;
};
```

Listing 1: The RenderConfig Structure

In addition to the boolean variables controlling the rendering flow, the configuration also maintains a list of post-processing shaders which are run after rendering and lighting has been completed. The variable lod_range of type LODRange is simply a structure holding two floating point values which indicate the squared distances used for the selection of the correct level of detail. This is used by the SceneManager during its culling and sorting procedure. The inner workings of scene management will be the topic of sections 3.5 and 4. A number of shaders is used during the lighting step and post-processing. Each type of light source has its own shader which can also be dynamically adjusted at runtime. This way the developer has complete control of the lighting computation. How the lighting step works will be explained in more detail below.
3.4.2 State Management

The maintenance of OpenGL state is implemented using another structure called RenderState:

```c
struct RenderState {
    GLboolean blend;
    GLboolean color_writes;
    GLboolean depth_test;
    GLboolean depth_writes;
    GLboolean face_cull;
    GLenum cull_face;
    GLenum depth_func;
    GLenum render_mode;
    // uniform buffers
    GLint max_ubuffer_bindings;
    // texture state
    GLenum active_texture;
    std::map<GLenum, Texture2DState> texture_2d_state;
};
```

Listing 2: The RenderState Structure

Though only a small subset of all OpenGL state variables is mapped, the most common cases can be managed effectively. As a representative example, one of the most important operations is the selection of the active texture unit and binding a specific texture object to it. Since only a single texture unit is considered active at a time, storing this particular value is unproblematic. The same is valid for the currently bound texture. The dictionary `texture_2d_state` therefore holds values indicating to which texture unit a specific texture object is bound.
The manager provides the following functions to change said state variables:

```cpp
void RenderManager::bindTexture2D(GLenum unit,
                                    GLuint texture)
{
    if(gl_state_.active_texture != unit)
    {
        gl_state_.active_texture = unit;
        glActiveTexture(unit);
    }

    GLenum active_texture = gl_state_.active_texture;
    // check if the target unit has already been
    // registered
    if(gl_state_.texture_2d_state.find(active_texture)
       != gl_state_.texture_2d_state.end())
    {
        // check if the currently assigned
        // texture object differs
        if(gl_state_.texture_2d_state[active_texture]
            != texture)
        {
            gl_state_.texture_2d_state[active_texture]
            = texture;
            glBindTexture(GL_TEXTURE_2D, texture);
        }
    }
    // otherwise it is safe to register and bind
    // without redundancy
    else
    {
        gl_state_.texture_2d_state[active_texture]
            = texture;
        glBindTexture(GL_TEXTURE_2D, texture);
    }
}
```

**Listing 3:** Setters for Active Texture and Binding a 2D Texture

It is obvious that the state changes will only happen if the targeted variable will be assigned a different value. This way, at least for the maintained state subset, redundant changes are completely avoided, potentially increasing the rendering speed. The same principle applies to the remaining variables.
3.4.3 Transformations

Managing and providing access to a set of transformations is crucial for applications and shader execution. Most of the data is provided as matrices, vectors or single floating point values. Among the most important are the model-view matrix and the projection matrix. To be able to use these within a shader, they have to be transferred to the program’s uniform memory, also called constant memory. Since many or all of these do not change more than once per frame, retransferring them to every active program is wasteful.

To counter this, current OpenGL specifications have introduced the notion of uniform buffers. Since this concept is used on various occasions, a brief explanation will now be given. For further information, especially usage constraints, one may refer to the OpenGL and GLSL specifications.

A uniform buffer is a regular buffer object\(^2\). Each buffer can be bound to a unique binding point and be used to source data from within all shaders accessing the buffer. The following code example shows generation and binding of an empty uniform buffer:

```c
GLuint uniform_buffer_object;

glGenBuffers(1, &uniform_buffer_object);
glBindBuffer(GL_UNIFORM_BUFFER, uniform_buffer_object);
glBufferData(GL_UNIFORM_BUFFER, 0, NULL,
             GL_DYNAMIC_DRAW);

GLint max_block_bindings;

glGetIntegerv(GL_MAX_UNIFORM_BUFFER_BINDINGS,
             &max_block_bindings);

glBindBufferBase(GL_UNIFORM_BUFFER,
                 max_block_bindings - 1,
                 uniform_buffer_object);
```

**Listing 4:** Binding a Uniform Block to a Unique Index

---

\(^2\)The concrete type of a buffer object is determined when it is first bound to a specific target like GL_UNIFORM_BUFFER.\[31\]
After generation of the buffer object, it is bound to the correct target and initialized with size zero and no data. Then the whole\(^3\) buffer is bound to a unique binding point. These are available in limited, implementation dependent quantity. In this example, the last possible value is chosen\(^4\).

In order to access a uniform buffer from within a shader, the GLSL specification defines uniform blocks. They are uniquely identified by a block name which can be connected to any uniform buffer with a single function call. For now it is sufficient to say that using uniform buffers saves a number of function calls as it allows to share data among several shader programs by simply updating the buffer instead of transferring values with separate function calls. As an example one may look at the following GLSL code snippet which is used throughout the shaders coming with MiReLi:

```glsl
layout(std140) uniform p_Transform
{
    mat4 p_ModelViewMatrix;
    mat4 p_ProjectionMatrix;
};
```

**Listing 5: Predefined Uniform Block Holding Shared Transformations**

The uniform block `p_Transform` is predefined and automatically maintained by MiReLi. Any shader program declaring it may access its data. Still, since the matrices generally change only once per frame, the update has to be done with the same frequency. The resolution and automated binding of shader inputs, including uniform buffers, is described in detail in chapter 5.

Still, one thing to notice is the `layout` qualifier. In conjunction with uniform blocks, it is a request to the OpenGL implementation to organize data which is accessed from the block in a certain way. The `std140` layout guarantees that the data is organized exactly as declared in the uniform block, across all programs that may use the block. This implies that certain optimizations are not imposed on the data - unused members for instance are not optimized out to keep everything consistent. The OpenGL specification defines two additional layouts which will not be covered here, as they are not used in the MiReLi shader library.

---

\(^3\)It is also possible to bind only parts of a uniform buffer to a binding point. This is shown in section 3.5.

\(^4\)OpenGL implementations define a maximum amount of buffer binding points in the range \((0, \text{GL\_MAX\_UNIFORM\_BUFFER\_BINDINGS}[3])\).
Aside from providing transformation data to shaders, applications can access the matrices directly from the matrix stack. Earlier OpenGL specifications defined multiple stacks for different modes which helped to manage matrices without losing any data. However, this functionality has been removed from the OpenGL core profile. This concept has been implemented in MiReLi as well and, as with state changes, there are dedicated functions to modify and query a specific stack. The equivalent of the removed OpenGL call glLoadMatrix, which simply overwrites the top of the stack with the given matrix, is implemented as follows:

```c
void RenderManager::loadMatrix(MatrixMode mode,
                                const Matrix4 &matrix)
{
    switch(mode)
    {
        case MODELVIEW:
            modelview_stack_->loadMatrix(matrix);
            break;
        case PROJECTION:
            projection_stack_->loadMatrix(matrix);
    }
}
```

Listing 6: Function to Load a Matrix Onto a Specific Matrix Stack

In contrast to older OpenGL versions, there is no notion of a matrix mode which has to be set prior to modifying the corresponding stack. In MiReLi, the selection is made based on a value from the enumeration MatrixMode. At the time of writing, MiReLi maintains only two stacks which hold matrices uploaded to the above mentioned uniform buffer. A concrete usage example will be shown in the next section describing the SceneManager.

### 3.4.4 Deferred Lighting

As a reminder, post-processing and lighting effects are applied by the RenderManager. Since MiReLi implements a deferred lighting pipeline, the lighting step is intentionally separated from the rendering step, which is in large parts handled by the SceneManager. Deferred lighting involves the maintenance of a number of off-screen buffers which certain properties of the scene and its objects are rendered to in multiple passes. In the setup described by Engel[15] this includes a buffer storing normals and depth, and an accumulation buffer for light contributions, called the light buffer. The general algorithm is as follows:
1. in a first geometry pass render normals and depth to a buffer
2. iterate over all light sources, compute the per-pixel lighting contributions to the scene and additively blend the results into the light accumulation buffer
3. in a second geometry pass render the scene with materials and reconstruct the lighting equation using the light buffer

Although rendering multiple geometry passes seems costly, one has to bear in mind that the costs for the first pass a relatively low and newer hardware handles it very well, even in scenes with a high polygon count. An advantage of the light pre-pass over traditional deferred shading are the lower bandwidth requirements and the possibility to render multiple materials without influencing the lighting computation. However, the light pre-pass shares one flaw with deferred shading: transparent objects cannot be handled like opaque geometry. The most simple approach to solving this, and in fact the way MiReLi tackles the problem, is to not simulate lighting on transparent geometry at all.

To support off-screen rendering, OpenGL provides a mechanism called framebuffer objects (FBO) [31]. Like the window system provided framebuffer, a FBO has a certain set of state and a mandatory or optional set of buffers, also called attachments or render targets, which may be used to render color, depth and stencil values. Just like other buffer objects, a FBO needs to be bound to a specific target in order to become active, in which case all drawing operations potentially modify only the attachments of the active FBO. Which render target to draw to can be easily decided from within a shader. MiReLi provides a class encapsulating all necessary facilities of framebuffer objects. Getting back to the deferred lighting, let us assume that we need to render a color buffer containing normal information, and a depth buffer attached to a framebuffer object.

---

5 Whether an attachment is mandatory depends on the state of the FBO. For instance, if an FBO is set to have one or more draw buffers, but has no color attachments, then the object is considered invalid and cannot be used.
The following fragment shader, which is part of MiReLi’s immutable shader library, renders the above mentioned properties to the corresponding framebuffer:

```glsl
#version 330
layout(location = 0) out vec4 Normal;
layout(location = 1) out vec4 Position;

in float e_DepthLinear;
in vec3 e_Normal;
in vec4 e_Position;

void main()
{
    Normal = vec4(normalize(e_Normal), e_DepthLinear);
    Position = e_Position;
}
```

Listing 7: Rendering Normals to an Explicit Render Target

The first observation to make is the layout qualifier. Current GLSL versions define different layouts for different shader stages. In the case of the fragment language, and given an active framebuffer object with a number of color attachments, the layout qualifier can be used to determine which render target to write to. The parameter `location` indicates the index of one of the FBOs color attachments, starting from 0 to the total number of attachments minus one. It is mandatory to declare a variable with the interface `out` when explicitly defining its location. After this, writing to the respective buffers is done as usual by a simple assignment. The depth buffer is filled automatically using the fixed-function pipeline.\(^6\)

Now the lighting simulation can take place. As shown above, there is one distinct shader per light source. One advantage of deferred lighting is that calculations are only done for parts of the scene which are visible and covered by the fragments generated for the light source geometry. This means that the polygonal mesh representing the area of interest has to be transformed into screen space and be shaded based on the type of the light source. For directional lights, a screen-filling quad is drawn, spot lights are represented by a cone, point lights by a sphere. Using the information from the normals and depth buffer, the light contributions of each fragment can be computed and additively blended into the light buffer using a simplified model.

\(^6\)Note that MiReLi currently uses an additional buffer storing eye-space positions. This is generally not necessary, since the position can be reconstructed from a fragments screen coordinates and the corresponding depth value.

\(^7\)Note that this will also result in a non-linear depth buffer which is useful for a number of tasks. Still, a wide range of effects benefit from linearized depth values which MiReLi encodes in the alpha channel of the normal buffer.
There are optimizations that apply to regular deferred shading\cite{19} and the light pre-pass alike. One that MiReLi includes is automatic detection of the correct polygon winding to ensure that light sources are correctly rendered if the camera is inside the volume, i.e. inside the cone of a spot and the sphere of a point light.

The resulting normal and light accumulation buffers for one of the test scenes is visualized below:

\textbf{Figure 3:} Visualization of the Normal and Light Accumulation Buffers
After all buffers have been laid out, the full material pass is rendered. A simple example of reconstructing diffuse lighting within a shader, consider the following fragment shader:

```glsl
#version 330

uniform sampler2D p_LightBuffer;
const vec4 Color = vec4(0.5, 0.5, 0.8, 1.0);
in vec3 e_TexCoord;
out vec4 p_FragColor;

void main()
{
  vec2 TexCoord = e_TexCoord.xy / e_TexCoord.z;
  vec4 DiffuseContrib = texture(p_LightBuffer, TexCoord);
  p_FragColor = Color * DiffuseContrib;
}
```

Listing 8: Simple Fragment Shader Reconstructing the Lighting Equation

In this shader computing the color of the a fragment is reduced to modulating a constant color with the diffuse lighting contribution retrieved from the light accumulation buffer. More complicated lighting equations are possible by encoding additional information into the light accumulation buffer.

### 3.4.5 Post-Processing

Following the lighting step, zero or more effects may be applied to the image currently stored in the back buffer. At present, this is simply an iteration over all post-processing shaders in the configuration which are applied to a screen-filling quad\(^8\). As with any shader, custom and predefined data may be used here as well. Although the set of effects is limited due to the available information at the time of writing, some common effects are still possible and are considered sufficient.

---

\(^8\)Note that for certain effects a full resolution render target is not required. A popular example of this is the full-scene glow effect.
3.5 The Scene Manager

While the RenderManager controls *how* to render the scene, the *Scene Manager* is concerned with *what* will be visible in the final image. It provides a number of functions the RenderManager invokes depending on its configuration. The responsibilities of the SceneManager can be summarized to:

- managing scene contents, including objects in an octree, light sources, and a skybox
- … view-frustum culling \[29\]
- occlusion culling using hardware occlusion queries
- separating opaque from translucent objects
- rendering shadow maps
- managing uniform buffers for light sources
- material state sorting

The following sections describe the most important tasks regarding the setup of the scene contents.

3.5.1 Loading Scenes

Before any rendering can take place, there need to be scene contents to process. As for resources, there is a distinct script type for describing the composition of the virtual world. To explain how it can be organized, one may first take a glance at the following screenshot, which shows a test simple test consisting of one directional light source, a textured plane and a camera, and the immediately following scene script:

![Figure 4: Simple Test Scene Showing a Textured Plane](image)
<!-- scenes are identified by unique names -->
<scene name="Textured Plane">

<!-- define a camera which looks at the origin -->
<camera>
  <position x="0.18" y="4.72" z="6.45" />
  <lookat x="0" y="0" z="0" />
</camera>

<!-- skyboxes are defined by six planes -->
<skybox>
  <positive x="px.png" y="py.png" z="pz.png" />
  <negative x="nx.png" y="ny.png" z="nz.png" />
</skybox>

<!-- a white light source radiating at a 45 degree angle -->
<directionallight name="sunlight">
  <direction x="-0.5" y="-0.7071" z="-0.5" />
  <ambient x="0" y="0" z="0" />
  <diffuse x="1" y="0.9067" z="0.9067" />
  <specular x="1.0" y="1.0" z="1.0" />
</directionallight>

<!-- object names are also unique IDs -->
<object name="plane">

<!-- an object may have a transformation. In this case it has no effect! -->
<transform>
  <translation x="0" y="0" z="0" />
  <rotation angle="0" x="1" y="0" z="0" />
  <scale x="1" y="1" z="1" />
</transform>

<!-- an object may consist of one or more meshes which define its geometry. Each mesh may have its own material and data group. -->
<mesh material="simple_texture">

Listing 9: Scene Script for Textured Plane
The scripts demonstrates a few important concepts and, as with any other script, each node maps to a class or structure in the system. The most crucial scene object is the *camera*, since without providing transformations used to orient and project objects to screen-space, the scene will not be rendered as expected. Such a transformation is derived from the definition of the *camera* node. It represents a virtual viewer with a position and a point in space it is looking at.

**Listing 10:** Scene Script for Textured Plane (contd.)
From this information, an instance of the \textit{Camera} class derives a local coordinate system and calculates the so called \textit{view matrix}, which becomes part of the transform buffer maintained by the RenderManager\footnote{In fact, the user camera is not maintained by the SceneManager, but by the RenderManager, since updating the transform uniform buffer is the latter one’s responsibility. Through a dedicated interface and with the help of the matrix stack, however, changing the active camera and updating said buffer is very simple. See below for further explanation.}. The instance created when reading a script serves as the main camera meant for moving through and viewing the scene. There may also be additional properties defined as attributes of the node defining the \textit{clipping planes} and the horizontal \textit{field of view}.

Skyboxes, which can enhance the realism of the portrayed virtual world, can be easily defined with the \textit{skybox} node. Since such boxes are essentially handled as a cube, there are six planes which need to be textured. Each plane is assumed to be perpendicular to one of the coordinate axes of the world coordinate system in positive and negative direction. The texture used for each individual face is determined by specifying the direction and axis.

Another important type of objects are light sources. Remember that at the time of writing MiReLi supports three types of lights, \textit{directional}, \textit{spot} and \textit{point}. Since the configurable properties differ among types, there exists a specific node for any kind of light source. Since light sources may “cast” shadows they need to maintain a \textit{Camera} object themselves. However, the properties of the camera can be computed from the properties and type of a light source.

Objects which may be visible in the final image are represented by \textit{object} nodes. Each object may have a number of properties, including a unique name, an object-to-world-space \textit{transform}, and at least one \textit{mesh}. Meshes store the actual geometry in the form of indexed data arrays. Each individual mesh may have its own \textit{material} and a specify a \textit{material data} group, which allows multiple materials to be applied to a single object.

Scene descriptions are evaluated by the \textit{SceneLoader}. Depending on the type of the object it encounters while parsing, different actions are taken. A camera is processed and then set as the main camera of the RenderManager, a skybox is also merely set up and added to the SceneManager. For lights the process involves loading all sources and then requesting the scene manager to setup the uniform buffer for each type. Setting up objects is slightly more involved as it includes loading the transformation, every individual mesh, which the transform needs to be applied to, computing the bounding box, and finally adding the object to the octree organizing the scene.
Dynamic loading and unloading is handled automatically by the SceneManager and can be triggered at runtime.

3.5.2 Light Source Buffers

MiReLi handles the lighting step in a simple, iterative manner. It loops over all sources, determines which lights are actually visible by view-frustum culling their geometry, and finally rendering the geometry using the appropriate shader. As shown above, there is a distinct program which is activated for all supported kinds of light sources. However, there needs to be a possibility to access the data associated with every light source of every type and, of course, the normal and depth buffers. This is solved by providing three distinct uniform buffers: \texttt{p\_DirectionalLight}, \texttt{p\_SpotLight} and \texttt{p\_PointLight}, as to allow fast switching between collections of uniforms. Additionally, potential shadow maps and the normal and depth buffer are bound to predefined uniform variables, \texttt{p\_ShadowMap} and \texttt{p\_NormalBuffer}. The below example highlights the interface used in the shader for rendering spotlights included in the library:

\begin{verbatim}
#version 330
layout(std140) uniform p_SpotLight
{
  vec4 LightPosition;
  vec4 LightDirection;
  vec4 LightAmbientColor;
  vec4 LightDiffuseColor;
  vec4 LightSpecularColor;

  // encodes vec4(cuttoff angle, max distance,
  // cast shadows, 0)
  vec4 AngleDistCastNull;

  // scaled and biased light view-projection matrix
  mat4 LightProjectorMatrix;
};

// buffers from first geometry pass
uniform sampler2D p_NormalBuffer;
uniform sampler2D p_DepthBuffer;
uniform sampler2DShadow p_ShadowMap;
out vec4 p_FragColor;
// ... actual shader code
\end{verbatim}

\textbf{Listing 11:} Predefined Spot Light Shader Interface
Setting the appropriate texture units for the buffer interface and the shadow map, if any, can be set one time and then be reused with any invocation of the shader. Still, the uniform block referencing the light source properties needs to be updated per light source with one simple line of code:

```cpp
1  glBindBufferRange(GL_UNIFORM_BUFFER,
2      // buffer binding point
3      gl_state_.max_ubuffer_bindings - 4,
4      // buffer index
5      ubo,
6      // offset and size
7      offset, gl_state_.ubuffer_alignment);
```

**Listing 12: Binding a Specific Range of a Uniform Buffer**

Assuming that the uniform buffer for all light types has been setup properly, the properties of every light source occupy an equally sized portion of its corresponding buffer. This portion can be bound by invoking `glBindBufferRange` which only binds a range of data specified by an offset and size. One important observation is that the offset needs to be a multiple of an implementation-dependent value `GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT`, which represents a concrete number of basic machine units.\(^\text{10}\) As a consequence every light source is assigned size \(n \cdot \text{alignment} \geq 1\) machine units, where the factor is automatically determined by the SceneManager to assure that all data correctly fits into the uniform buffer. When binding a range, the offset starts at 0 and is then incremented for every light source by \(\text{offset} = \text{offset} + \text{size}\). The computations is done like in the following example for the above mentioned spot lights:

Assuming an implementation-dependent offset alignment of

\[
\text{alignment} = 256 \text{ bytes} \quad (1)
\]

a necessary data size for each spot light in the the buffer of

\[
\text{datasize} = 10 \cdot 4 \cdot \text{sizeof(GLfloat)} = 160 \text{ bytes} \quad (2)
\]

it follows that the necessary buffer space per light source is

\[
\text{bufferspace} = \left\lfloor \frac{\text{datasize}}{\text{alignment}} \right\rfloor \cdot \text{alignment} = 256 \text{ bytes} \quad (3)
\]

and the resulting buffer size is simply

\[
\text{buffersize} = \text{numlights} \cdot \text{bufferspace} \quad (4)
\]

\(^{10}\)On most PC platforms this is given in bytes. For instance, on one of the test systems used for this work which uses an OpenGL implementation by NVIDIA, the alignment is defined to be 256 bytes.
This applies to all types of light sources and guarantees properly setup uniform buffers for every platform and implementation. Conforming to the requirements of an OpenGL implementation may result in buffer space which is allocated, but not actively used. In the above calculation, 96 bytes are effectively never accessed. However, this is a small price to pay considering the unified approach and ease of implementation for all light sources. Staying with the example, the following figure visualizes the actual buffer layout for 1 to n spot lights:

<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>64</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Direction</td>
<td>Ambient</td>
<td>Diffuse</td>
<td>Specular</td>
<td>Angle</td>
<td>Projection</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Figure 5: Uniform Buffer Layout for Spot Lights

It is important to note that keeping the semantic order of the variables in the uniform block is absolutely crucial. For example, if one switched the first to variables, the calculations in the shader would very likely produce false results. Furthermore, adding variables to the block which are not backed by sufficient buffer space may result in an application crash! For safety and simplicity, it is best to simply follow the propagated layout.
3.5.3 Interaction of SceneManager and RenderManager

As an example of the connections between the RenderManager and Scene-Manager, consider the following code snippet illustrating the process of updating shadow maps:

```cpp
// push the matrix states for modelview and
// projection matrices
renderer.pushMatrix(RenderManager::MODELVIEW);
renderer.pushMatrix(RenderManager::PROJECTION);

// set the light source’s camera properties
renderer.loadMatrix(RenderManager::MODELVIEW,
    light->getCamera()->getViewMatrix());
renderer.loadMatrix(RenderManager::PROJECTION,
    light->getCamera()->getProjectionMatrix());

// upload new values to uniform buffer
renderer.updateTransformBuffer();

// activate light sources FBO and render
light->activateFramebuffer();
glClear(GL_DEPTH_BUFFER_BIT);
renderSceneSimple(renderer, false, false);
light->deactivateFramebuffer();

// restore original matrix state
renderer.popMatrix(RenderManager::PROJECTION);
renderer.popMatrix(RenderManager::MODELVIEW);
```

Listing 13: A Function of the Scene Manager Manipulating the Render Managers Matrix Stack

To be able to correctly render a shadow map, the current transformation matrices have to be uploaded to the uniform buffer object holding the shared transformations for all shaders. The RenderManager’s matrix stack lets the SceneManager manipulate this buffer by first pushing the corresponding matrices onto the respective stacks and second by updating the transform buffer. After rendering has been done, the original state will be restored and the RenderManager will take care of updating the transform itself.
4 Scene Management

The SceneManager uses different strategies depending on the render configuration. The most simple form is to only invoke view-frustum culling, based on the so called radar approach\[29\], and depth sorting. This is sufficient for simple scenes without too much geometry. The second variant is to include distance based selection of the level of detail similar to \[32\]- provided that multiple levels have been specified for the object in question. The third and most complicated method involves hierarchical occlusion culling, based on the ideas presented by Mattausch et al. in an approach called coherent hierarchical culling (CHC)\[22\]\(^\text{11}\). Before going into greater detail, the following flow chart depicts the steps involved in the scene management process:

![Flow of Scene Management Tasks](image)

**Figure 6: Flow of Scene Management Tasks**

View-frustum culling and depth sorting are always performed, regardless of the occlusion culling configuration. The result is always a specific priority queue, storing opaque objects in front-to-back and translucent objects in back-to-front order. The difference between no or brute force occlusion culling and hierarchical occlusion culling is that in the first-mentioned case, the octree is subject to a simple depth-first traversal, inserting objects which are inside the view-frustum into the respective queues.

\(^{11}\)Note that although this work abbreviates the method to CHC, it actually uses an updated version of the initial proposition which is denoted CHC++. 
It is important to note that culling against the view-frustum is done in a hierarchical manner, potentially saving a large number of tests. Should an internal node of the octree not intersect the frustum then all child nodes, and subsequently all objects in this volume, will be discarded. The latter method traverses the octree in the fashion which is described below.

Level of detail selection is optional, either due to objects providing only one level, or due to selection being disabled, and is performed straight-forwardly using the squared distance of the center of the object’s bounding box from the camera, determined as part of view-frustum culling. Although this is sufficient for most cases, sometimes getting a better estimate is necessary to ensure correct order. Therefore the squared distances to the corners representing the minimal and maximal extents of the bounding box are checked and compared to the the distance to the center; the closest point is then selected.

If simple occlusion culling is enabled, the resulting distance queue for opaque objects will be examined for occluded objects. Culling translucent parts of the scene is not considered important since for one, translucent objects can not be considered occluders, and second, the amount of translucent geometry is expected to be low in contrast to opaque contents which makes them low-priority occludees that may very well introduce rendering costs.

To avoid unnecessary state changes, the last step sorts all objects by material, respectively by shader, as recommended in [27, 25].

To quote Mattausch et al., “visibility algorithms can be roughly categorized into those that work as a pre-processing step and those that work at runtime”. The first step of visibility determination used in MiReLi, view-frustum culling, is obviously a runtime algorithm. Still, this method determines everything inside the view-frustum, including objects that may be occluded by others, making way for overdrawing in scenes with high depth complexity, which “refers to how many times each pixel is overwritten”[9]. Especially in densely occluded scenes with high depth complexity, using a method for rejecting occluded objects is expected to increase rendering performance.

A popular preprocessing technique for rejecting occluded parts of an indoor scene has been and still is portal culling. A description is also available in [9]. In such environments, walls may occlude large parts of the viewer’s surroundings. As the name suggests, portals, which usually refer to doors and windows and internally represented by some bounding volume, which connect cells, usually rooms or hallways delimited by walls, are inserted into the scene and associated with objects.
At runtime, only objects which fall into a logical frustum, determined by projecting a portal’s bounding volume to the screen, are visible and will be rendered. Portal culling can thus be seen as a representative of both view-frustum and, to a certain extent, occlusion culling. Still, it should be mentioned that this method is not a full-fledged occlusion culling technique. As simple example, one may imagine a portal through which two objects are visible, and the first of the two fully occludes the second. Both will be rendered since they are inside the view-frustum, but the latter could have been rejected. However, it has already been pointed out in section 2.3 that with today’s hardware this will not be a problem in the majority of cases.

One traditional method which has been available in hardware for a long time is z-buffering, described for example in [9]. By maintaining a buffer storing depth values of rendered primitives, updating the color buffers is only necessary if later rendered scene contents produce fragments that pass what is called the depth-test in OpenGL. Depending on a specific depth function, fragments which pass will update the depth and color buffers, the ones which fail will be rejected. Aside from providing the benefit of order-independent rendering for opaque geometry, modern GPUs are able to reject fragments before they are processed by the fragment shader which may vastly improve rendering performance when using complex per-pixel shading effects. This is called Early-Z or Z-call and can be enforced by rendering a pre-Z or z-only pass [25]. Although this method may save much processing, it still requires rendering at least parts of the scene which are major occluders to lay down an effective depth buffer. Since, however, current GPUs reach multiple times the speed for z-only passes, this can be an efficient way to reject occluded fragments and is also beneficial for tasks such as shadow map rendering. Still, although it helps for many purposes, relying only on z-buffering is not sufficient to reach high rendering speeds.

As already mentioned earlier, a method which achieves very good performance for densely occluded scenes is CHC. It is based on hardware supported occlusion queries [31]. An application can request the OpenGL implementation to render some object and record the number of samples, which is similar to the number of fragments that are actually generated. If the returned value is greater than zero or some threshold, at least part of the object’s geometry is visible at the time of testing. The general approach is to render very simple objects, such as a bounding box, to quickly determine the visible samples. The application can then decide whether an object is to be drawn.
Aside from the additional rendering, the most severe problems of occlusion queries in terms of performance are the following:

- state changes may be necessary to ensure that the color and depth buffers are not updated by the geometry used for the query
- a query itself incurs a cost which cannot be neglected
- waiting for the results of a query may stall the GPU and the CPU

Preliminary tests showed that disregarding the above observations can actually decrease rendering performance substantially. It is therefore crucial to reduce state changes as well as occlusion queries, and to try to fill the time waiting for query results as good as possible - three major goals of the presented method. Although Mattausch et al. use a bounding volume hierarchy, coherent hierarchical culling can also be applied to the octree used by MiReLi, but it should be noted that a simplified version of the algorithm is used.

Aside from a distance queue, which was already present in MiReLi, the authors of CHC employ the following methods to improve the performance of occlusion culling using occlusion queries:

1. Formerly invisible and visible nodes (and objects) are accumulated in queues to reduce state changes
2. They perform what is denoted multiqueries to determine if a group of node is visible with a single query, based on cost-benefit metric.
3. Using a randomized sampling pattern, temporal alignment and the number of queries for formerly visible nodes is reduced
4. Tight bounding volumes which reduce falsely classified object and thereby also reduce the number of queries

Besides reducing state changes, the first point also helps to take advantage of an important property and the first word in CHC: coherence. Since in many cases visible objects are assumed to stay visible for at least a period of time, issuing an occlusion query may be wasted since the visibility classification may not have changed. To reduce this effect, nodes in the visible queue are only checked if a certain amount of time of frames have passed. The invisible queue, however, needs to be processed every frame since there can be no guarantee that nodes that were not visible in frame $n$ are still invisible in frame $n+1$ and detecting such a change too late may result in suddenly appearing objects.
An important observation is that there may be a number of nodes which stay collectively invisible, and *hierarchical culling* helps to identify such groups; just like in the case of view-frustum culling. In addition, multiqueries can be used to determine if a collection of nodes is invisible with only one occlusion query. Should the amount of samples for \( n \) nodes be zero, the reduction of queries is equal to \( n \). Otherwise the number of queries is \( 1 + n \), one for the multiquery and \( n \) for the individual nodes.

To counter the problem of temporal alignment, which refers to querying a large number or all formerly visible nodes at the same time and can cause severe frame rate drops, the *first* query for each node is offset by a random value. This way temporal alignment and the number of queries per frame is reduced.

Tighter bounding volumes are not implemented in MiReLi but have been proven to have some effect by Mattausch et al.
5 Shaders and Materials

As already stated above, MiReLi is entirely shader based and it therefore makes sense to automate the process of introducing them into the system. All work necessary to obtain a valid shader program is done by the essential Shader class. However, shaders must have access to a number of values to be able produce the desired image. For this purpose, OpenGL specifies a number of interfaces which allow applications to transfer data to the graphics card and to establish a connection between shader programs and this data. This process will be the topic of the first section of this chapter.

While a shader can produce certain results without intentionally altering the appearance of the fragments,\textsuperscript{12} the common case involves some sort of color computation to produce the final image. Although internally any data is interpreted as numbers, historically there has been the notion of an object's material. In the easiest scenario, an object can simply be assigned a number of colors and values which serve different purposes, for instance a diffuse color to simulate the diffuse reflection encoded in a 24-bit RGB value. Older version of OpenGL\textsuperscript{30} offered fixed functionality to set such data and have it applied during lighting or simply, if lighting was turned off, by putting colors out to the framebuffer. The most common way by far, however, is the application of textures in conjunction with certain attributes. Since the advent of programmable shading, more complex lighting models and image-space effects (see section 3) have opened a wide range of possibilities for representing the appearance of objects in a three dimensional scene.

The general problem is that nowadays OpenGL and graphics hardware do not have any concept of materials. It is completely the programmers responsibility to determine the outcome of the shading process. Therefore it is necessary to establish a logical collection of data representing an object's material properties. MiReLi provides a class named Material which will be explained in this chapter's second section.

\textsuperscript{12}Rendering a non-linear depth buffer is a good example where there need not actually be a fragment shader, since the depth values are generated by the fixed-function pipeline without user intervention.
5.1 Shaders

As already mentioned before, shaders are small programs being executed by dedicated processors on the GPU. Creating such a program involves a number of steps which are shown in the following figure:

![Figure 7: Operations Needed to Create a Shader Program](image)

MiReLi’s *Shader* class provides convenience functions encapsulating all necessary commands. Remembering that the ResourceManager automatically loads all resources it can find in the specified search paths, the only user action necessary is providing a valid *shader script* file, stating with the unique name of the program and the shader source code for at least the vertex stage, and optionally the geometry and fragment stages.
For instance, the shader used to process point lights is introduced into the system using the following script:

```
<?xml version="1.0"?>
<shader_program name="point_light">
  <vertex_shader>
    <source filename="point_light.vert"/>
  </vertex_shader>

  <!-- geometry and fragment shaders are optional only shown for demonstration -->

  <!--geometry_shader>
  <source filename="..."/>
  </geometry_shader-->

  <fragment_shader>
    <source filename="point_light.frag"/>
  </fragment_shader>
</shader_program>
```

**Listing 14: A Typical Shader Script**

Although identifying the source files that belong to a shader could be simply done by comparing file names, using a script offers more safety and, more importantly, lets the user define unique identifiers themself. While the script is being read, an instance of the class is being created. Through the respective functions the program is set up properly and the shader is made available on the application level. Should any of the operations fail, the system will output an appropriate error message to the log and the ResourceManager will assign the corresponding null resource.

The final step before the shader can be fully used by MiReLi is querying the declared shader interfaces, specifically attributes, uniform variables, and uniform blocks. This is automatically done after validation\(^\text{13}\) and results in a number of *ShaderInputs*, a class storing the name, location, type and size of each interface declaration. The two latter properties, however, are not provided separately by OpenGL, but in the form of a symbolic constant. For instance, the type and size of a variable declared as a vector of four floating point values, `vec4`, will be represented by the symbolic constant `GL_FLOAT_VEC4`. The *ShaderLoader* automatically deduces the type and size of every active variable by analysing the constant returned by OpenGL. The importance of this information is stressed in the next section.

\(^{13}\)Validating a shader is not strictly necessary. However, it is recommended since it guarantees that the shader program will definitely run\(^{31}\).
Every instance of ShaderInput can be queried by identifier and used to check if a specific interface declaration is active in the program. Also, the block indices of uniform blocks can be resolved by name. Now the shader is ready for use and can be queried from anywhere a reference to the ResourceManager can be obtained.

Providing access to certain information is crucial for the rendering process to function. Former versions of GLSL have exposed certain state for the vertex and/or fragment language through well defined built-in interfaces. One example is the following structure `gl_FogParameters` and the global uniform instance `gl_Fog` which is available:

```cpp
struct gl_FogParameters
{
    vec4 color;
    float density;
    float start;
    float end;
    float scale;
};

uniform gl_FogParameters gl_Fog;
```

**Listing 15: Built-In Structure Declared In GLSL 1.20**

The most useful property of such built-in uniforms was that they could be used across different programs. With the core profile of current OpenGL and GLSL versions, many built-in variables have vanished, since much of the state and data backing the variables have been removed from the implementation. Still, having easy and convenient access to frequently used information is favorable. Rendering engines like Ogre3D have defined additional built-ins alongside those provided by OpenGL to offer usage of data defined by the engine itself. These are not dependent on any concrete version or implementation of the abstraction layer.

MiReLi follows this approach and maintains a number of predefined bindings to specific vertex attributes and uniform data for use inside shaders, which are automatically backed by sufficient and appropriate data. One example which is frequently used throughout the shader library is the uniform block holding transformation matrices called `p_Transform`. All predefined names are distinguished from user defined names by the suffix `p_`.
As already mentioned before, the data to back the buffer is sourced directly from video memory and is automatically updated by the RenderManager each frame. Every predefined attribute or uniform block is statically assigned a specific identifier. For uniform buffers, the following binding points are permanently used:

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Binding Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformations</td>
<td>p_Transform</td>
</tr>
<tr>
<td>Directional Lights</td>
<td>p_DirectionalLight</td>
</tr>
<tr>
<td>Spot Lights</td>
<td>p_SpotLight</td>
</tr>
<tr>
<td>Point Lights</td>
<td>p_PointLight</td>
</tr>
</tbody>
</table>

\[
\text{MAX} = \text{GL\_MAX\_UNIFORM\_BUFFER\_BINDINGS}
\]

**Table 1:** Predefined Uniform Buffers and Binding Point Mappings

Some vertex attributes are also exposed and frequently used, at least in part, in almost every shader in the library:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Positions</td>
<td>p_Vertex</td>
</tr>
<tr>
<td>Vertex Normals</td>
<td>p_Normal</td>
</tr>
<tr>
<td>Texture Coordinates</td>
<td>p_TexCoord</td>
</tr>
</tbody>
</table>

**Table 2:** Predefined Vertex Attribute and Location Mappings

It is important to correctly declare all predefined variables inside a shader source file. If the storage, type, and naming conventions are not followed, the data will not be defined when the shader is executed. The most severe problems may occur when declaring members of a uniform block which are not backed by sufficient buffer space. In that case the application may crash when a shader tries to access the invalid portions of memory.

Although not permanently exposed, additional data can be made available in shaders through *materials* which are the the topic of the following section.

---

\[^{14}\text{Meaning that it will never change throughout the application run time, unless manually changed - which is not recommended.}\]
5.2 Materials

In MiReLi, a material is interpreted as the combination of a shader and one or more groups of additional resources like textures or color values. Before going into more detail about the inner workings of the class, the following diagram highlights the relations between of the involved types:

![Figure 8: The Material Class and Associated Types](image)

Clearly visible is the mandatory, valid reference to a shader. Unless the shader defined for the material is present in the system, the material is not considered valid as well. Should this be the case, an error message is put out and the corresponding null resource, which in this case is simply a material referencing the null shader, will be assigned. A material may contain zero or more DataGroups which have a unique index. These are collections of uniform values and textures, represented by the two structures UniformData and TextureUnit, each of which store specific information. Note that although inside a shader source code types may be declared with the uniform storage qualifier, independently of whether they are single values, vectors or samplers, MiReLi makes an intentional distinction between them samplers and other typed of data to emphasize the different semantics and the slightly different way they are bound by an application. The incentive for using different groups of data is the fact that it allows to reuse the material with different inputs. Otherwise, each instance would have to be defined separately. This concept, though implemented differently, is also used by Ogre3D. Also, there may be a default data group, which is useful for materials that will be applied to a number of different objects sharing the same appearance.
Materials need to be defined within a *material script*. As an example consider the following script file:

```xml
<?xml version="1.0"?>
<material name="textured">
  <shader name="single_texture"/>
  <inputs>
    <texture name="Texture" image="wall.png" unit="0"/>
    <uniform name="Scale" x="2.0" y="1.0"/>
  </inputs>
</material>
```

**Listing 16: Simple Material Script Defining two Shader Inputs**

The above description defines a material named *textured* which references a shader with the identifier *single_texture*. As the name of the latter suggests, it simply applies a texture to a mesh using its texture coordinates. Within the fragment source code, a uniform sampler named *Texture* and a uniform two-dimensional vector of floats, *Scale*, are declared.

Provided that the program works correctly and the variables are active, remember that there will be a corresponding number of ShaderInputs stored by an instance of the class *Shader*. When a material script is read, MiReLi will try to match the defined inputs to the information of the ShaderInputs. Should there be any discrepancy, be it that the variable is inactive, the types do not match, or a specified image cannot be found in the system, a warning or error message will be put out to to log. For every valid input, the specified data will be associated with the uniform location that has been determined by OpenGL. In the above example there will the active uniform sampler *Texture* which may, for instance, have uniform location 0. It is now necessary to associate this unique index with the texture holding the data from the image *wall.png* which is expected to be bound to a specific texture unit identified by *unit*. Assuming that the image has been loaded successfully and the corresponding texture has been created, the *MaterialLoader* will call the following function of the Material class to set up the above mentioned associations:
void Material::addTextureUnit(unsigned int data_group,
    const std::string& name,
    GLint loc,
    GLenum tex_unit,
    Texture2 *texture)
{
    if(data_group < groups_.size())
    {
        assert(texture != NULL);

        TextureUnit texunit;
        texunit.name = name;
        texunit.loc = loc;
        texunit.image = texture;
        texunit.texture_unit = tex_unit;
        groups_[data_group].texunits.
            push_back(texunit);

        // bind the sampler’s location to texture
        // unit ‘texunit’
        shader_->activate();
        glUniform1i(texunit.loc,
            textureUnitToInt(texunit));
        shader_->deactivate();
    }
}

Listing 17: Adding a Texture Unit to a Material

Non-sampler uniform variables are treated somewhat differently. Recall that at the time of writing only single values and vectors can be defined inside a material script. The process of validating the existence of the variables defined in the script and retrieving the uniform location is also applied here, but it is not possible to associate data with uniforms using a single function call - like in the case of samplers. Determining which instance of glUniform* to choose is deduced from the size and type retrieved from the corresponding ShaderInput instance. In the example shown above, the uniform two-dimensional vector Scale is supposed to be assigned two values, x and y. The number of arguments suggests that the mapped GLSL type is a vec2. It can now be checked if the number of arguments matches the size stored in the associated instance of ShaderInput.
In case it deviates, there are two possible solutions:

1. ignore additional arguments
2. replace missing arguments with 0

In any case a warning message will be put out to the log stating which definition was inconsistent and which choice was made to resolve the problem. Should everything match, the MaterialLoader invokes `Material::addUniformData()` transferring all necessary data to the material’s active data group. At run time, the information is used to call the appropriate `glUniform*` call.

Updating the shader’s uniform variables for the specified data group is done straightforwardly using the following function:

```cpp
void Material::update(unsigned int group, 
                     RenderManager &renderer)
{
    if(!groups_.empty())
    {
        UniformDataCVec &uniforms =
            data_groups_[group].uniforms;
        for(size_t i = 0; i < uniforms.size(); ++i)
            updateUniform(uniforms[i]);

        TexUnitCVec &texunits =
            data_groups_[group].texunits;
        for(size_t i = 0; i < texunits.size(); ++i)
        {
            const TextureUnit &unit = texunits[i];
            renderer.bindTexture2D( unit.texture_unit, 
                unit.image->getGLObject());
        }
    }
}
```

**Listing 18:** Updating the Shader of a Material
As can be seen, updating merely involves iterating over the contents of data group specified by the first argument. The function `Material::updateUniform()` does exactly what has been described above; it calls the appropriate variant of `glUniform*` and takes a structure named `UniformData` containing all information as an argument. The update of texture units is handled by the RenderManager, since it provides functionality to avoid redundant state changes when binding a texture to a texture unit. After this step, the rendering of the object sharing the material and the specified data group may commence.
6 Results

This section is divided into two parts. The first examines the characteristics of the library in regard to rendering performance using the presented scene management techniques, so for each scene there exists a table comparing statistics gathered at runtime. The second part portrays the visual quality which can currently be achieved by showing screenshots of different scenes simulating various effects.

The test system was equipped with a NVIDIA GeForce 8600M GS mobile GPU at stock clock and memory speeds, and an Intel Core 2 Duo CPU running at 2GHz. Ubuntu 11.04 with kernel 2.6.38-11 and the latest GPU drivers installed served as the Linux distribution for testing. All culling tests were conducted at a resolution of 800x600 pixels.

6.1 Rendering Performance

The first test shows a small scene with low to moderate depth complexity and a triangle count of \( \sim 2.73 \) million.

\[ \text{Figure 9: Scene with Low to Moderate Depth Complexity and Triangle Count} \]
Several runs suggest a small performance benefit over view-frustum culling when using CHC. On average, using occlusion culling gained the application additional 6 FPS, although the number of polygons was significantly higher with view-frustum culling. The average number of queries was as low as expected and did not exceed 17.

In the second scene, intended to simulate an urban setting, the depth complexity is considerably higher. Also, the triangle count is raised to ~5 million.

![Urban Scene with High Depth Complexity and Polygon Count](image)

**Figure 10:** Urban Scene with High Depth Complexity and Polygon Count

Again, using CHC performs better than view-frustum culling. However, the margin is limited to 10 FPS on average. An average number of 160 queries is again acceptable. From a certain polygon count, the frame times of both approaches appear to converge leading to a marginal improvement over CHC. The following example substantiates this suspicion.
As an indoor scene with moderate to high depth complexity and a triangle count 570,000, the Sponza Atrium was used. Here the advantage of CHC is clearly visible as it gains 25 FPS with an average query count of 61.

**Figure 11:** Occlusion Culling Test using with the Sponza Atrium

This indicates that with a lower polygon count, additional geometry shows a greater impact and more quickly increases the frame time on the target hardware. Further tests on different GPUs may provide entirely different results.

Overall, CHC performed better than view-frustum culling in all scenes but the performance gains were almost negligible in the geometry heavy examples. This is somewhat surprising since the additional cost incurred by CHC was low enough that a more clearly visible benefit could have been expected. Further investigation needs to be conducted to verify the effectiveness of the implementation on different hardware. Brute force occlusion culling was, as is to be expected, inferior to both CHC and view-frustum culling in all tests.
For very simple scenes, the overhead of any of the occlusion culling methods was too high to gain additional performance. In such cases, simple view-frustum culling was preferable.

To get an idea of how well the light pre-pass performs, consider the following scene:

![Image: Figure 12: 256 Simultaneous Dynamic Point Lights](image)

The above scene shows a single plane being illuminated by 256 dynamic point lights. Simulating this many light sources using forward rendering would decrease the performance considerably. With the light pre-pass, even the low-budget mobile GPU used for testing reached over 30 frames per second at a resolution of 1024x768 pixels.

In general, the number of state changes was acceptably low and there were no redundancies. A real problem that remains for the time being is the number of draw calls and vertex array bindings. Depending on the number of objects in the scene, their number may be too high to achieve good performance.
6.2 Visual Quality

In its current state, MiReLi is useful for a variety of effects. The following examples are all realized using only the predefined interfaces and, if necessary shader constants. Since one of the goals of this work was to enable convincing imagery, this section portrays some of the effects that are currently possible with the library.

The first image shows a simple landscape to which four textures are applied by a texture splatting shader. In addition, a number of trees with alpha blended leaves - using premultiplied alpha - directional lighting and screen-space exponential fog enrich the image.

![Figure 13: Foggy Terrain](image)

To portray the lighting and shadowing capabilities of MiReLi, the following picture shows a simple scene with four differently colored dynamic spot lights using a Phong lighting model, and four cubes casting dynamic shadows on a ground plane. Note that uniform shadowmapping without filtering is applied.
Returning to the example of the Sponza Atrium, the following scene shows the building being illuminated by 26 point lights and one directional light, with a simplified version of Crytek’s *Screen-Space Ambient Occlusion* applied as a post process.

**Figure 14:** Spot Lights Illuminating Four Cubes and a Plane

**Figure 15:** Sponza Atrium with Five Visible Point Lights
Figure 16: Sponza Atrium Showing Screen-Space Ambient Occlusion

Figure 17: Full View of Sponza Atrium with Screen-Space Ambient Occlusion
6.3 Integrating MiReLi

The demonstration program has been compiled as a stand-alone application statically linking to MiReLi. Since the library can be automatically and completely initialized with a single call to `EngineManager::init()`, loading and rendering a scene is reduced to passing the name of a valid configuration file to the function. Implementing camera movement was equally simple, since the Camera class provides convenience functions to easily alter the position and direction of the viewer camera. The only means necessary to render a scene is invoking `RenderManager::render()`. Releasing all of the resources allocated by the library during initialization, both regarding system and video memory, can be accomplished with another single call to `EngineManager::shutdown()`. Memory leaks on from either one could not be observed.

Altering the contents of scripts has proven to be an effective means of influencing the functionality of the library. The appearance of scene objects, post-processing effects and several rendering configurations were successfully tested without any changes to the library code and recompilation.

Validation of script files and the information put out by the library has proven most useful. During the testing phase it has never been unclear if an operation failed due to false user input, or due to a software error. In the case of invalid materials, shaders or scene contents, visual feedback trough null objects and additional analysis of the log contents always lead to a quick solution. Furthermore, the library has proven to be stable in all test runs.
7 Conclusion and Future Work

This work presented the conception and implementation of the Minimal Rendering Library using current GPU features based on the OpenGL 3.3 core profile. An extensive discussion of scene management and rendering approaches has lead to a flexible and modular rendering system and resulted in a easy to use, reusable and extensible software library, which fulfilled all of the formerly stated requirements.

One issue that has not been addressed is the reduction of draw calls and batching in general, which can increase application performance considerably. At present, for every object that is visible, one call to bind each mesh’s vertex array object, and one draw call for every mesh will be issued. Geometry instancing, texture arrays and more sophisticated vertex buffer management will have to be integrated to develop the library further and yield optimal runtime performance.

Considering the current state of occlusion culling, there needs to be a thorough review of the implemented CHC method to be able to achieve performance gains comparable to Mattausch et al. Although a clear advantage could be observed in one test scene, different hardware needs to be employed and more data gathered to be able to really judge the potential of the current implementation of CHC.

MiReLi’s lighting facilities, although sufficient for a range of scenes, need to be extended and further improved applying more tried and true optimizations implemented in existing deferred renderers. Integrating further possibilities to achieve more realistic and sophisticated illumination effects are a primary goal. In general, the set effects which can be rendered varies mostly with the amount of predefined input data to shaders. Identifying additional information and augmenting the library accordingly will increase the number of use cases. Also shader performance will be examined more closely and optimizations will be implemented. Furthermore, high dynamic range lighting is readily available with MiReLi, since it uses 16-bit floating point buffers for rendering. Increasing the quality of the shadowing solution and providing more methods than simple uniform shadow mapping is another interesting task.

The material system, while proving useful enough for this work, needs to be extended to allow further customization of object appearance, for example, information to animate textures. Since MiReLi handles only static geometry at the moment, bringing more “life” into scenes is another effort. This may include particle systems, character animation, animated skyboxes, and so on.
On the side of scene management a most useful new feature would be the integration of a terrain rendering subsystem. Algorithms which can work nicely and integrate well into the library are readily available. Also, providing support for streaming assets into and out of video memory is a necessary feature for a real-world shader centric rendering library. Which approaches to tackle this specific requirement are best suited needs to be determined in future research.

Another very useful project would be to develop a scene editor based on MiReLi to support direct editing of scenes without the intervention of a separate content creation tool. There will also need to be extensive documentation of the library and its mechanism to provide an adequate basis for developers. Realizing that MiReLi is completely shader centric, using it as the basis of a shader development environment can also easily be pictured.

Although easily integrable, the library is not yet ready for use in larger real-world projects. More effort needs to be invested into determining which additional features are expected to be present, would be useful, and could drive adoption by other developers.

To draw a final conclusion it is sufficient to say that MiReLi is already useful for a variety of tasks and it manages meet the expectations stated before. However, the library is far from complete and remains a work in progress.
List of Figures

1. MiReLi's Main Components ........................................... 14
2. Flow of the Rendering Process ...................................... 17
3. Visualization of the Normal and Light Accumulation Buffers .......... 26
4. Simple Test Scene Showing a Textured Plane .......................... 28
5. Uniform Buffer Layout for Spot Lights ................................ 34
6. Flow of Scene Management Tasks .................................... 36
7. Operations Needed to Create a Shader Program ....................... 42
8. The Material Class and Associated Types ............................ 46
9. Scene with Low to Moderate Depth Complexity and Triangle Count .................................................. 51
10. Urban Scene with High Depth Complexity and Polygon Count ........ 52
11. Occlusion Culling Test using with the Sponza Atrium .............. 53
12. 256 Simultaneous Dynamic Point Lights ............................ 54
13. Foggy Terrain ............................................................ 55
14. Spot Lights Illuminating Four Cubes and a Plane .................... 56
15. Sponza Atrium with Five Visible Point Lights ....................... 56
16. Sponza Atrium Showing Screen-Space Ambient Occlusion ........ 57
17. Full View of Sponza Atrium with Screen-Space Ambient Occlusion .................................................. 57

List of Tables

1. Predefined Uniform Buffers and Binding Point Mappings ............. 45
2. Predefined Vertex Attribute and Location Mappings .................. 45

Listings

1. The RenderConfig Structure .......................................... 18
2. The RenderState Structure ............................................ 19
3. Setters for Active Texture and Binding a 2D Texture ................ 20
4. Binding a Uniform Block to a Unique Index .......................... 21
5. Predefined Uniform Block Holding Shared Transformations .......... 22
6. Function to Load a Matrix Onto a Specific Matrix Stack ............ 23
7. Rendering Normals to an Explicit Render Target ...................... 25
8. Simple Fragment Shader Reconstructing the Lighting Equation .... 27
9. Scene Script for Textured Plane ...................................... 29
10. Scene Script for Textured Plane (cont'd.) ........................... 30
11. Predefined Spot Light Shader Interface ................................ 32
12. Binding a Specific Range of a Uniform Buffer ...................... 33
13. A Function of the Scene Manager Manipulating the Render Managers Matrix Stack .......................................... 35
References


[33] Wilson, K. Game Object Structure: Scene Graphs. [http://gamearchitect.net/Articles/GameObjects2.html](http://gamearchitect.net/Articles/GameObjects2.html) (last checked 02.05.2011).