Diplomarbeit

Designing a Modern Rendering Engine

ausgeführt am
Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
und
Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
als verantwortlich mitwirkenden Universitätsassistenten

durch

Matthias Bauchinger
A - 3602 Rossatz 152

Datum

Unterschrift

Für meine Großeltern
Elfriede und Heinrich Frischengruber

Abstract

The development of *real-time rendering* applications has become one of the most difficult software engineering areas due to the number and complexity of the needed techniques and algorithms involved. These software projects have in common that they need to structure the data in the main memory, process it and send it to the graphics device for rendering in an efficient way. These recurring and complex algorithms are provided by so called *rendering engines* to allow faster development of real-time rendering applications.

This thesis describes the concepts and design decisions which form the basis for the development of the rendering engine presented in this document. Detailed information is provided on the interface to the graphics device, a novel effect framework and the implemented graph structures allowing efficient data traversal.

Kurzfassung

Die Entwicklung von *real-time rendering* Programmen wurde durch die Anzahl und Komplexität der benötigten Techniken und Algorithmen zu einer der schwierigsten *Software-Engineering* Gebiete. Diese Software-Projekte haben gemein, dass sie die darzustellenden Daten strukturiert im Hauptspeicher anordnen , für den Rendering-Prozess effizient abarbeiten und an die Grafikkarte weiterleiten müssen. Um diese wiederkehrenden, komplexen Algorithmen nicht mit jedem Projekt neu entwickeln zu müssen, werden sogenannte *Rendering Engines* verwendet, die genau diese Aufgaben übernehmen.

In dieser Diplomarbeit werden die Konzepte und Designentscheidungen beschrieben, die der speziell für diese Arbeit entwickelten Rendering Engine zugrunde liegen. Diese Arbeit befasst sich detailliert mit der Schnittstelle zur Grafikkarte, einem neuartigen *effect framework* und den implementierten Graphenstrukturen um die Daten effizient zu traversieren.

Contents

1 Introduction **1**
- 1.1 Real-Time Rendering . . . 1
  - 1.1.1 Graphics Hardware . . . 2
  - 1.1.2 Graphics APIs . . . 2
- 1.2 Graphics Software Frameworks . . . 3
- 1.3 Graphics Effects . . . 4
- 1.4 Goals of this Work . . . 4
- 1.5 Structure of this Document . . . 5

2 Related Work **6**
- 2.1 The Rendering Pipeline . . . 6
  - 2.1.1 The Generation Stage . . . 7
  - 2.1.2 The Traversal Stage . . . 7
  - 2.1.3 The Geometry Stage . . . 8
  - 2.1.4 The Rasterization Stage . . . 9
  - 2.1.5 The Display Stage . . . 9
- 2.2 Programming the GPU . . . 10
  - 2.2.1 Types of Shaders . . . 10
  - 2.2.2 Shader Models . . . 12
  - 2.2.3 High-Level Shading Languages . . . 13
- 2.3 Handling of Shaders and Effects . . . 17
  - 2.3.1 Definitions . . . 17
  - 2.3.2 Requirements . . . 17
  - 2.3.3 Existing Solutions . . . 17
- 2.4 Spatial Data Structures . . . 19
  - 2.4.1 The Octree . . . 19
  - 2.4.2 The kd-tree . . . 20
- 2.5 The Scenegraph . . . 21
  - 2.5.1 The Bounding Volume Hierarchy . . . 22
  - 2.5.2 Data Sharing . . . 22
  - 2.5.3 Classical Scenegraph Frameworks . . . 25
- 2.6 Design Patterns . . . 25
- 2.7 Comparison of Available Rendering Engines . . . 27

3 Designing the Rendering Engine **31**
- 3.1 Why another Rendering Engine? . . . 31

3.2 The Requirements . . . 32
3.3 The Layers of the Engine . . . 33
3.4 The Core Module . . . 37
3.4.1 The Math Framework . . . 37
3.4.2 The I/O submodule . . . 38
3.4.3 The Data Interface . . . 38
3.4.4 The Reflection System . . . 39
3.4.5 The Engine Framework . . . 40
3.4.6 Making the Engine Scriptable . . . 40
3.5 The Graphics Pipeline . . . 41
3.5.1 The Scenegraph . . . 41
3.5.2 The Drawgraph . . . 42
3.5.3 The Effect Framework . . . 42
3.5.4 The Rendering Interface . . . 42
3.6 The Variables Concept . . . 44
3.6.1 What is a Variable? . . . 44
3.6.2 Creating a Variable . . . 44
3.6.3 What are the Benefits of a Variable Container? . . . 46
3.6.4 Variable Manipulators . . . 46
3.6.5 Emitting Variables . . . 47
3.6.6 The Interfaces . . . 47
3.7 The Rendering Interface . . . 49
3.7.1 Details about renderable geometries . . . 49
3.7.2 Vertex- and Fragment Programs . . . 51
3.7.3 Using the Rendering Interface . . . 53
3.7.4 The Interfaces . . . 56
3.8 The Effect Framework . . . 57
3.8.1 Overview . . . 57
3.8.2 Structure of Effects . . . 58
3.8.3 Generating the Vertex- and Fragment Framework Programs . . . 63
3.8.4 The State Table . . . 69
3.8.5 Rendering the Effects . . . 70
3.8.6 Handling of Multipassing . . . 72
3.8.7 Effect Level of Detail . . . 73
3.8.8 Performance Optimizations . . . 74
3.8.9 Post-Processing Effect Framework . . . 76
3.8.10 Implemented Effects . . . 76
3.8.11 Extending the Framework . . . 91
3.8.12 Emitting Effects . . . 92
3.8.13 Comparison with Existing Solutions . . . 92
3.8.14 The Interfaces . . . 93
3.9 The Drawgraph . . . 96
3.9.1 Solving the task . . . 96
3.9.2 The Renderable Class . . . 97

3.9.3 The Drawgraph Interface . . . 98
3.10 The Scenegraph . . . 99
3.10.1 Differences to Conventional Scenegraphs . . . 100
3.10.2 Scenegraph Objects . . . 101
3.10.3 Data Sharing . . . 103
3.10.4 The Scene Database . . . 104
3.10.5 Updating the Scene Database . . . 105
3.10.6 Retrieval of Renderable Objects . . . 110
3.10.7 Scenegraph Examples . . . 111

4 Implementation Details **115**
4.1 The Coding Environment . . . 115
4.2 The OpenGL Driver . . . 116
4.2.1 Resource Handling . . . 116
4.2.2 Variables . . . 116
4.2.3 Rendertargets . . . 117
4.3 Applied Design Patterns . . . 117
4.4 External Frameworks . . . 118
4.5 The Editor . . . 118
4.6 Feature Summary . . . 122

5 Evaluation **123**
5.1 Fulfilling the Requirements . . . 123
5.2 Software Engineering . . . 124

6 Summary and Future Work **126**
6.1 Future Work . . . 126

A C++ Interfaces **128**
A.1 Interfaces of the Variables Concept . . . 128
A.2 Main Rendering Interfaces . . . 133
A.3 Interfaces of the Effect Framework . . . 145
A.4 Interface of the Drawgraph . . . 150

B UML Diagrams **152**

List of Figures **155**

List of Tables **157**

Bibliography **158**

1 Introduction

1.1 Real-Time Rendering

Real-time rendering is the process of displaying three-dimensional data as images on the computer at an interactive display rate. The display rate is measured in *frames per second* (fps), which, for real-time rendering, should be higher than 15 fps [RTR02] to be recognized as *smooth* by the user. From about 72 fps [RTR02] and up, the human eye cannot detect any differences in the display rate.

To achieve interactive frame rates some tradeoffs have to be made. First, the generated image usually is not as *photorealistic* as with global illumination methods like path tracing [Kajiya86] and radiosity [CohenEtAl93]. Figure 1.1 shows a comparison of an image generated using real-time rendering techniques and an image rendered using global illumination methods. Additionally, complex algorithms have to be applied to avoid work which does not contribute to the final image.

Popular areas of application of real-time rendering are computer games, scientific and information visualization and virtual reality.

Figure 1.1: A comparison of a real-time rendered image (∼500 frames per second) on the left and an image generated by a global illumination method on the right (∼50000 seconds per frame). *(Right image by Nick Chapman)*

1.1.1 Graphics Hardware

Since the introduction of the 3Dfx Voodoo 1 graphics accelerator card in 1996 [Eccles00], interactive real-time rendering is available on desktop PCs. Before that, the complete computation of the final image had to be carried out by the CPU. Since then, graphics accelerators (also called *GPUs*) have evolved rapidly in terms of performance, functionality and flexibility. Therefore, more and more work has been shifted from the CPU to the GPU to increase the overall performance of the rendering process. Nowadays, the CPU is mainly used to prepare the data for the GPU by structuring and traversing the models in the main memory. Additionally, physics simulation and artificial intelligence is applied to these models.

The latest GPU at the time of this thesis is NVIDIA's Geforce 8800 [NVIDIA] which is able to compute near photorealistic images in real-time. See Figure 1.2 for a sample image generated by this GPU.

Figure 1.2: This picture shows the photorealistic rendering of a fashion model rendered at interactive frame rates using NVIDIA's Geforce 8800 graphics accelerator. *(Image courtesy of NVIDIA Corp.)*

1.1.2 Graphics APIs

Since most real-time rendering applications need to address almost the same features of the graphics accelerator, several standard programming interfaces (also called *graphics APIs*) were developed to provide portability of the applications allowing to send the

commands to different kinds of graphics devices. The two most prominent graphics APIs are Microsoft's *DirectX* and *OpenGL*.

The first version of the multimedia API DirectX was developed by Microsoft in 1995 and was called *GameSDK*. It provided interface classes which could be used by the programming languages C and C++. At the time of this thesis the latest version is DirectX 9.0 whose interfaces can also be used by *managed languages* like C# and VB.Net.

OpenGL was originally introduced by Silicon Graphics in 1992 and is supported by most operation systems available by now. This makes it the first choice for developing portable graphics applications. Graphics hardware vendors and other graphics related companies have organized themselves as the OpenGL **Architecture Review Board** (ARB) which leads the specification of the OpenGL interface. OpenGL uses the so called *extension concept* for early integration of new features provided by the graphics accelerators.

1.2 Graphics Software Frameworks

Since programming graphics applications has become a very complex task, and often the same code is needed over and over again, graphics software frameworks are available to support the developers. These frameworks (also called *rendering engines*) provide wrappers around the graphics APIs and often needed functionality to allow faster development of graphics applications. Thus, their main task is to support the developers of graphics programs.

Common provided features of rendering engines are as follows:

- Loading and rendering of 3D models.
- Loading and manipulation of different kind of image formats.
- Applying textures, shading and more advanced effects to models.
- Organization of the models for easy execution of different operations on the models.
- Performance optimizations for faster rendering of the models.

Rendering engines are available as commercial (and often very expensive) packages as well as open-source projects. Some of them are presented in Section 2.7. It depends on the following facts which rendering engine fits a specific project:

- The *budget* for the project
- The *programming language* of the project - rendering engines are often available for C/C++, C#, Java and Delphi.
- The *platform* the project is intended to run on. Some engines support both, PCs and gaming consoles. Some engines will run only on Microsoft Windows and others support Linux too.

- The provided *features* of the rendering engine. E.g. for some projects scripting language support is mandatory which is not always provided by the frameworks.

1.3 Graphics Effects

In the context of this thesis, a *graphics effect* is the combination of algorithms needed for rendering a 3D model onto the screen to give the desired visual appearance. Therefore, graphics effects can differ on the following points:

- The geometry, vertex and fragment shaders they use. See Subsection 2.2.1 for details.
- The number of rendering passes they need. See Section 2.3 for details.
- The data input they require. This includes the number and type of textures and lights, as well as the vertex attributes of the geometry.

1.4 Goals of this Work

The management of graphics effects (Section 1.3) has become an important topic and key feature of rendering engines. With the increasing number of effects it is not sufficient anymore to only support them, but also to integrate them into the rendering engine in a clean and extensible way.

The goal of this work and simultaneously its main contribution is to design and implement an *advanced effects framework*. Using this framework it should be easy for further applications to combine several small effects like texture mapping, shading and shadowing in an automated and transparent way and apply them to any 3D model. Additionally, it should be possible to integrate new effects and use the provided framework for rapid prototyping.

Since no existing rendering engine was able to act as code framework for this task, a new rendering engine had to be written to embed the new effects framework. Thereby, the extent of this thesis increased, since common features of available rendering engines were needed too. Specifically, a scenegraph implementation should be provided to allow high-level usage of the effects framework with smooth integration of the implemented effects. A more formal task definition as a list of requirements is provided in Section 3.2.

Since the implemented rendering engine is a relatively big software project, it is also important to define what is **not** the task of this thesis:

- The implemented rendering engine should not become a game engine. Therefore, the integration of physics simulation and artificial intelligence is not necessary.
- The support of animation paths or skeletal animation is not necessary.

- The focus does not lie on the number nor completeness of the implemented effects. Only some basic effects should be provided to test the framework and to get an idea how to integrate additional effects.

1.5 Structure of this Document

This thesis is structured the following way: In Chapter 2 the background for this thesis is presented. It covers the basic layout of graphics applications, programming of the graphics accelerator, graph structures used by graphics applications and a comparison of available graphics frameworks. The main part of this thesis is Chapter 3, which presents the ideas and concepts behind the rendering engine written during this thesis. Afterwards, details on the implementation of this engine are presented in Chapter 4. After checking if the requirements of the engine are fulfilled in Chapter 5, the last chapter contains a summary of the whole thesis along with *open issues* of the implementation. The appendix contains important C++ interfaces and UML diagrams to give the interested reader more details on the class layout.

2 Related Work

2.1 The Rendering Pipeline

The *rendering pipeline* describes the traditional processing steps taken by 3D real-time rendering applications (*RTR-applications*). Each of these applications needs to perform the following tasks:

- Any RTR-application needs data it can visualize. Therefore, it can generate the data or it can load it directly from a storage medium.
- This data has to be organized and prepared for rendering by the program.
- The data has to be transformed to fit into the viewport of the output device.
- Afterwards, the data is converted into pixels and displayed at the output device.

Some of these tasks are performed on the CPU and others have to be executed by a dedicated graphics accelerator (also called *graphics processing unit - GPU*). A typical example for a task which has to be performed on the CPU is the loading of data from a harddisk. On the other side, displaying the generated pixels can only be done by the GPU.

With the increasing power and flexibility of todays graphics cards some of the tasks can be shifted from the CPU to the GPU. Even the generation of data can now be accomplished by the GPU in some cases. The main reason for this increased flexibility of the GPU is its programmability as explained in detail in Section 2.2. Another example of a task which can now by executed on the GPU is *physics simulation* including *collision detection*. Govindaraju et al [GovindarajuEtAl03] presented an algorithm which is able to calculate a set of potentially colliding objects on the GPU using image-space occlusion queries. Another example of physics simulation which lends itself well to a GPU implementation is cloth simulation. Zeller [Zeller06] presented an algorithm for cloth simulation where the cloth is modelled as a set of 3D particles stored in a floating-point texture. The movement of these particles is then calculated only by vertex- and fragment programs running on the GPU.

The tasks mentioned above are directly mapped to the *stages* of the *graphics pipeline* which are described in detail in the following sections. An overview of the stages along with the possible processing-resources can be found in Figure 2.1. All stages are working in parallel. That means, that while e.g. gamma correction is applied to a fragment in the

rasterization stage, new vertices are transformed in the geometry stage. Therefore, the graphics pipeline behaves like a manufacturing assembly and each stage adds something to the previous stage.

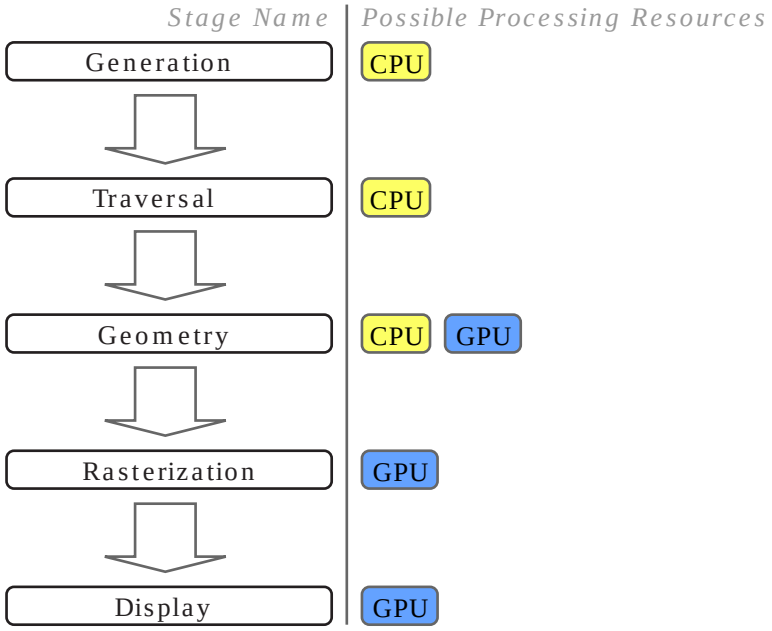

Figure 2.1: The stages of the *graphics pipeline*. For every stage the processing resources are listed which can be used to execute the corresponding stage.

2.1.1 The Generation Stage

The main task of the *generation stage* is to somehow create the data of the objects which have to be stored in the main memory of the computer for later visualization. This can be achieved by generating the data at runtime or by loading the precomputed data from a storage medium. The models along with their data are then organized in the main memory for flexible and performant further processing. This can be as simple as a list of references to the models or they can be organized in a graph as described in Section 2.5.

2.1.2 The Traversal Stage

At the *traversal stage* the data structure of the application is traversed and modified if necessary. During traversal the appropriate graphics commands are sent to the graphics accelerator to generate a visual representation of the model data. As a communication

interface between the CPU and the GPU, most applications use either Microsoft's *Direct3D* [D3D] or SGI's *OpenGL* [OpenGL] *graphics application programming interface* (graphics API).

The combination of the generation- and traversal stage is often referred to as the *application stage*. This stage can also include event handling of the application, artificial intelligence and physics simulation.

2.1.3 The Geometry Stage

At the *geometry stage* the polygons and vertices of a model are processed. The involved processing steps include:

- Transform the vertex positions from object- to perspective- and clip space as illustrated in Figure 2.2
- Calculate per-vertex lighting and shading
- Texture coordinates can be generated and transformed (if necessary)
- Assemble vertices into primitives (e.g. triangles)
- Perform clipping to the primitives
- Perform the perspective division
- Perform back face culling
- Transform polygon coordinates from clip space to device-dependent viewport

Until 1998 PC graphics accelerators were not able to perform the steps listed above in hardware. It was the task of the developer to implement this functionality using the CPU. In 1999 NVIDIA [NVIDIA] released a graphics accelerator called *GeForce 256* which implemented the vertex transformations and lighting calculations in hardware. This new feature was called *Hardware Transform and Lighting* (T&L). Since these hard-wired calculations were performed a lot faster then using the CPU, the polygon throughput of the graphics pipeline was increased. The transformation calculations performed on the position vectors of a model are illustrated in Figure 2.2.

However, the increasing complexity of graphics applications required also more flexibility of the GPUs in terms of possible per-vertex operations. The hard-wired transformations and lighting models were not sufficient any longer. Therefore, the graphics industry introduced the so called *vertex programs* (also called *vertex shaders*). Since then it has been possible to modify vertex attributes like normals and colors in a user defined way. More details on vertex programs can be found in Section 2.2.

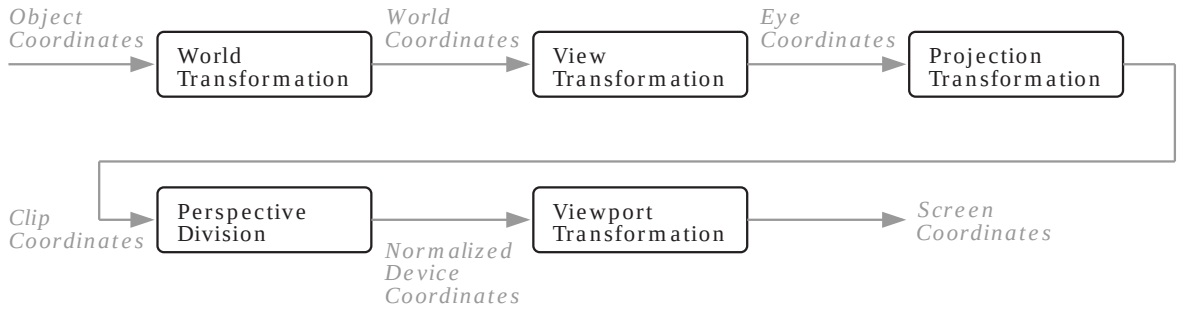

Figure 2.2: Transformation calculations performed on the position vectors

2.1.4 The Rasterization Stage

The task of the *rasterization stage* is to convert two-dimensional primitives (including the vertex colors and similar attributes) into colored pixels stored in a two dimensional array called *framebuffer*. This process is often called *scan conversion* as well.

The performed steps are listed below:

- Triangle setup
- Convert the triangle into fragments by interpolating the vertex attributes of the triangle corners. The interpolated attributes are e.g. the color, texture coordinates and depth values.
- Perform owner, scissor, alpha transparency and stencil tests on the fragment. Using such tests fragments can be discarded before they get displayed.
- Calculate the fragments' final color using lighting, texture mapping and alpha blending.
- Resolve visibility among fragments by testing their depth values against the depth value stored in the Z-buffer [Catmull75].

To increase the flexibility of this pipeline stage, fragment programs were introduced. Using these programs the GPU can be programmed to perform a user-defined set of operations on fragments. Details on this topic can be found in Section 2.2.

2.1.5 The Display Stage

At the *display stage*, *gamma correction* [RTR02] can be applied to the colors of the fragments following Equation 2.1, where V is the voltage input, a and γ (gamma) are constant for each monitor, ϵ is the black level (brightness) setting for the monitor, and I is the generated intensity.

$$I = a(V + \epsilon)^{\gamma} \tag{2.1}$$

This intensity is then converted into electrical voltage output to be used by output devices to display the content of the framebuffer.

Usually, however, gamma correction is *not* applied at the display stage, but all colors (textures, vertex colors, shader inputs) are already stored gamma corrected. This allows the best use of the available floating point precision in the graphics pipeline. See [WimmerRTR04] for details on this topic.

2.2 Programming the GPU

The calculations performed in the *geometry stage* and *rasterization stage* as described in Section 2.1 used to be hard-wired into the graphics accelerators until the release of the NVIDIA Geforce3 graphics device in 2001. This graphics device was able to run small programs modifying vertex and fragment data on its GPU. Such programs are called *vertex-* respectively *pixel shaders* and were introduced with DirectX 8 (they were also available as OpenGL extensions). The task of the shaders (also called programs) is to replace the hard-wired transformation and lighting model of the GPU (also called *fixed function pipeline*) with a programmable one.

2.2.1 Types of Shaders

This section introduces the three types of shaders available at the moment which are illustrated in Figure 2.3.

Vertex Programs

Vertex programs are applied to each vertex of a model and are used to change their attributes. This includes the transformation of vectors or computations of lighting models. The so called *uniform parameters* are additional inputs to the program whose values are constant for each shader invocation. One limitation of vertex programs is that they have only access to data of one single vertex. Fetching the data of another vertex in the model is not possible. Additionally, a vertex program cannot create new vertices.

Examples of use include:

- Lens effects like fish-eye lenses
- Twist and bend of objects
- Movement of cloth or water surfaces

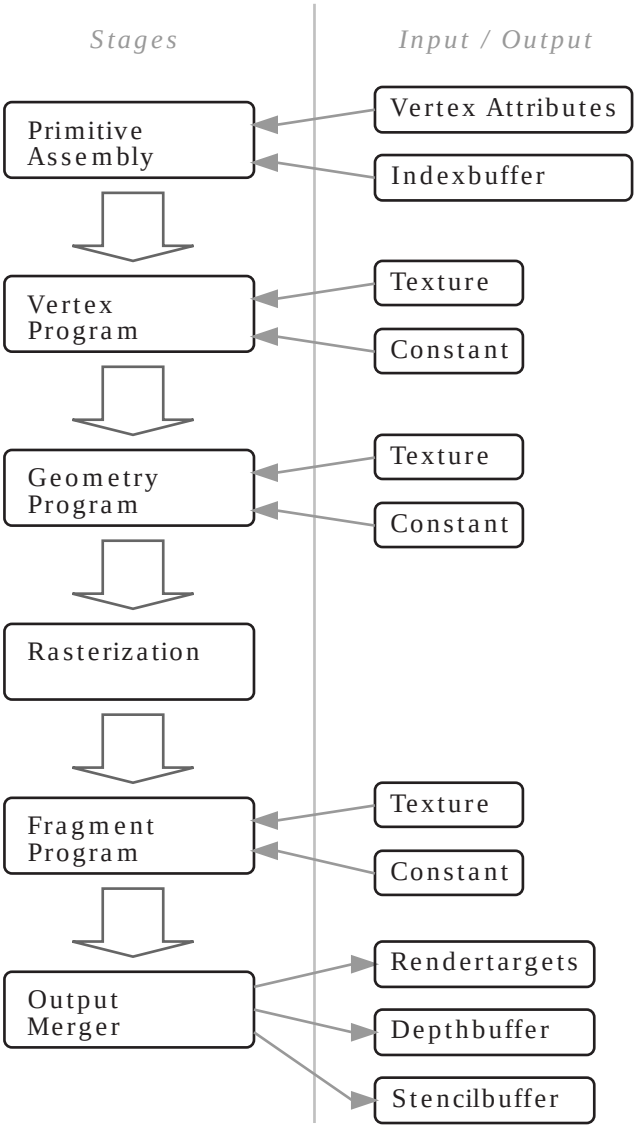

Figure 2.3: The extended version of the graphics pipeline running on the GPU. Parts of the *fixed-function pipeline* have been replaced with programmable stages.

- Texture coordinate generation for the fragment program.

Geometry Programs

Geometry programs were introduced with the Shader Model 4.0 as described in Subsection 2.2.2. A geometry program is invoked for every primitive group like a trianglestrip or a point list, and is able to generate new primitives (including new vertices) and can even convert the type of the primitive. Examples of use include the generation of shadow volumes on the GPU and procedural generation of models.

Fragment Programs

Fragment programs are applied to each fragment generated by the *scan conversion* as described in Subsection 2.1.4. The task of a fragment program is to take the fragment attributes and uniform parameters as input and compute a final color for that fragment which will be written to the render target. The fragment attributes are the interpolated attributes of the associated vertices during the scan conversion.

Examples of use include:

- Calculation of accurate lighting models
- Simulation of multi-layer surface properties
- Post-processing effects like glow and depth-of-field

2.2.2 Shader Models

With every new version of graphics accelerators and graphics APIs like DirectX and OpenGL, the range of available instructions for vertex- and fragment programs is increased. The version of the instruction set is also referred to as *shader model*. The following listing provides remarkable new features introduced with new shader models.

- Shader Model 1.0: The baseline model introduced by the first vertex- and fragment programs
- Shader Model 2.0: Increased maximum number of instructions in vertex- and fragment programs; introduced *branching* instructions; unlimited number of texture instructions; increased number of temporary and constant registers
- Shader Model 3.0: Infinite length of vertex- and fragment programs; full support for subroutines, loops, and branches; texture fetches in vertex programs; simultaneous output to multiple render targets (MRTs)

- Shader Model 4.0: Introduction of geometry shaders allowing to generate primitives on the GPU; unification of shaders which removed differences between pixel and vertex shaders; increased number of texture samplers (128); increased number of temporary registers

2.2.3 High-Level Shading Languages

The computer language used to implement shaders is a low-level programming language very similar to the assembler language used to program the CPU. Therefore, the main task of high level shading languages is to provide an easier-to-use computer language for implementing shaders.

Due to the existence of various graphics device vendors and graphics APIs, different shading languages have been developed. The most relevant high-level shading languages are presented in the following sections.

An example of a DirectX vertex shader is given below:

```
vs.1.1;
dp4 oPos.x, v0, c4;
dp4 oPos.y, v0, c5;
dp4 oPos.z, v0, c6;
dp4 oPos.w, v0, c7;
mov oT0.xy, v7;
mov oD0, v5;
```

OpenGL shading language

The OpenGL shading language [GLSL] is also known as *GLSL* or *glslang* and can only be used by the OpenGL graphics API. GLSL was defined by the *Architectural Review Board*[ARB] during the specification of OpenGL 2.0.

The source code snippet below shows the vertex- and fragment program written in GLSL for a diffuse effect:

```
// ---------------------------------
// Vertex Program
// ---------------------------------
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
  // Transforming the vertex to projection space.
```

```
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  // Transforming the normal to model-view space.
  normal = gl_NormalMatrix * gl_Normal;
  // Transforming the vertex position to model-view space.
  vec4 vertex_in_modelview_space = gl_ModelViewMatrx * gl_Vertex;
  // The vector from the vertex position to the light position
  vertex_to_light_vector = vec3(gl_LightSource[0].position -
    vertex_in_modelview_space);
}

// --------------------------------
// Fragment Program
// --------------------------------
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
  // Defining the material coefficients.
  const vec4 AmbientColor = vec4(0.1, 0.0, 0.0, 1.0);
  const vec4 DiffuseColor = vec4(1.0, 0.0, 0.0, 1.0);

  vec3 normalized_normal = normalize(normal);
  vec3 normalized_vertex_to_light_vector = normalize(vertex_to_light_vector);
  float DiffuseTerm = clamp(dot(normal, vertex_to_light_vector), 0.0, 1.0);
  gl_FragColor = AmbientColor + DiffuseColor * DiffuseTerm;
}
```

DirectX High-Level Shader Language

The High-Level Shader Language [HLSL] was introduced by Microsoft [MS] with DirectX 9.0 [D3D] and is also referred to as *HLSL*. *HLSL* was released before *GLSL* and later extended to match the feature set of the latter. As with *GLSL*, *HLSL* is bound to a single graphics API namely *Direct3D*.

The source code snippet below shows the vertex- and fragment program written in HLSL for a diffuse effect:

```
float4x4 matWorldViewProj;
float4x4 matWorld;
float4 vecLightDir;

struct VS_OUTPUT
{
  float4 Pos : POSITION;
  float3 Light : TEXCOORD0;
```

```
  float3 Norm : TEXCOORD1;
};

// ---------------------------------
// Vertex Program
// ---------------------------------
VS_OUTPUT VS(float4 Pos : POSITION, float3 Normal : NORMAL)
{
  VS_OUTPUT Out = (VS_OUTPUT)0;
  Out.Pos = mul(Pos, matWorldViewProj);
  Out.Light = vecLightDir;
  Out.Norm = normalize(mul(Normal, matWorld));
  return Out;
}

// ---------------------------------
// Fragment Program
// ---------------------------------
float4 PS(float3 Light: TEXCOORD0, float3 Norm : TEXCOORD1) : COLOR
{
  float4 diffuse = {1.0f, 0.0f, 0.0f, 1.0f};
  float4 ambient = {0.1f, 0.0f, 0.0f, 1.0f};
  return ambient + diffuse * saturate(dot(Light, Norm));
}
```

Cg programming language

The *Cg programming language* [MarkEtAl03] (Cg stands for *C for graphics*) was developed by NVIDIA [NVIDIA] and can be used with both major graphics APIs: OpenGL and Direct3D. Cg has the same syntax as HLSL.

The source code snippet below shows a simple vertex- and fragment program written in Cg:

```
struct appdata
{
float4 Position : POSITION;
float3 Normal : TEXCOORD0;
};

struct vfconn
{
float4 HPOS : POSITION;
float3 Normal : TEXCOORD0;
};
```

```
// ---------------------------------
// Vertex Program
// ---------------------------------
vfconn main(appdata IN,
uniform float4x4 WorldViewMatrixIT,
uniform float4x4 WorldViewProjectionMatrix)
{
vfconn OUT;
OUT.HPOS = mul(WorldViewProjectionMatrix, IN.Position);
OUT.Normal = mul(WorldViewMatrixIT, float4(IN.Normal,1)).xyz;
return OUT;
}

// ---------------------------------
// Fragment Program
// ---------------------------------
fragout main(vfconn IN)
{
  fragout OUT;
  float grey = clamp(dot(IN.Normal, float3(0,0,1)), 0, 1);
  OUT.col = float4(grey, grey, grey, 1);
  return OUT;
}
```

A distinctive feature of Cg is that the language allows the usage of *shader interfaces* [Pharr04] as uniform input parameters. These interfaces can then be implemented by *structures* which can be switched at runtime.

The example below defines an interface for all lights and gives a point light implementation of that interface:

```
// The interface for all lights.
interface ILight
{
    float3 illuminate(float3 p, out float3 L);
};

// Point light implementation of the ILight interface.
struct PointLight : ILight
{
    float3 Plight, Clight;
    float3 illuminate(float3 P, out float3 L)
    {
        L = normalize(Plight - P);
        return Clight;
    }
};
```

2.3 Handling of Shaders and Effects

2.3.1 Definitions

Rendering Pass: The sum of all processing commands needed to render a set of geometries with the same textures and GPU programs into one specific rendertarget.

Local Multipassing: The process of rendering a single geometry multiple times with different rendering passes.

Global Multipassing: The process of rendering the geometries of a complete scene multiple times with different rendering passes.

2.3.2 Requirements

One requirement of the users of a rendering engine is to support always the newest rendering effects on the market. Therefore, the time from inventing an effect to the integration into the rendering engine should be short. This enables one to use the engine for rapid prototyping. Another requirement is that the rendering engine should be well designed and have a clean API to be used by other developers. That enforces the developers of the engine to structure the engine in a way that the integration of additional rendering techniques can be done in a clean way.

The rendering engine should not only be able to allow the usage of new vertex- and fragment programs but should also provide a clean and predefined way to implement completely new rendering techniques having global impact on the processing of the whole scene.

2.3.3 Existing Solutions

One approach to organize shaders (see Section 2.2 for more information on shaders) in a rendering engine is the *individual-program* approach [ORorke04] where a collection of shader files exist. Each of these files is either a vertex- or a fragment program used for a single rendering pass for one or more objects. An advantage of this approach is that it is easy to add a new vertex- or fragment program to the list. However, a downside to this approach is that providing just the shader files is usually not enough to implement a new rendering effect. Additionally, management of the individual passes along with their corresponding renderstates is needed, which would result in custom source code for every new rendering effect implemented according to the individual-program approach.

Another approach is to use so called *effect files* whose format is provided by one of the existing *effect frameworks*: either Microsoft's effect framework [MSEF] or NVIDIA's CgFX framework [CgFX]. With this approach all the needed vertex- and fragment programs

of a single rendering effect are placed into a single file and structured into rendering passes. Additionally, effect files allow the specification of the needed renderstates for a pass. Thus, when implementing a new rendering effect which needs local multipassing the user just has to add a new effect file to the collection.

A problem with the presented approaches above is that there will exist a lot of duplicate shader code. The reason for this is that most rendering effects will implement e.g. a common lighting model and some kind of texture mapping. With this duplicate code, maintenance of the shader code can become a lot of work. A different approach which addresses this problem is the so called *Uber Shader* [Hargreaves04] which implements all desired behaviors in a single shader. It is the task of the application to disable elements not currently required. Two techniques exist to implement this behavior:

- Using flow control instructions in the shader code dynamically deciding which elements of the shader have to be calculated
- Using preprocessing of the shader code along with *#ifndef blocks*

A downside to this approach is that it quickly gets complicated to merge all possible rendering techniques into a single shader source code. Also extending this shader can easily break the functionality of another shader element.

The converse approach to Uber Shaders are *Micro Shaders* as presented by Shawn Hargreaves [Hargreaves04]. With Micro Shaders common functionality is splitted into small shader code fragments which are concatenated by the application to build the desired shader code. The problem with this approach is that the source code fragments can get in conflict with each other if they use the same registers or the same name for variables.

A similar apporach to Micro Shaders are the *Abstract Shade Trees* as presented by McGuire et al. [McGuireEtAl06]. With this technique the elements of a shader are expressed as *atoms* which are defined by a declaration, a set of struct/global function definitions, and a body. The body of the atoms contains real shader code which can be written in any available high level shading language. The user of the system can then create a tree out of these atom nodes which is then parsed by an algorithm to generate the complete shader code. The implemented system makes type mismatches in shaders impossible. The limitations of *Abstract Shade Trees* are:

- The created shaders can exceed the instruction and register count limits of the available GPU.
- The implemeted compiler does not feature whole-program optimizations.

The author of this thesis presents an approach similar to *Micro Shaders* which eliminates its disadvantages using the *shader interface* feature of *Cg* as described in Subsection 2.2.3. This new approach as presented in Section 3.8 is also able to handle global multipassing needed for global effects like shadowing, reflection and refraction.

2.4 Spatial Data Structures

This section describes spatial data structures which are used to prevent *unnecessary work* of the GPU. *Unnecessary* in this context means that the work does not contribute to the final image and therefore, can be omitted. The presented techniques have in common that they use some kind of hierarchical structure to store the objects of a scene. This hierarchical structure is then tested against the viewing frustum to find the list of visible scene objects. All objects not visible to the viewing frustum will not be rendered. This results in a performance improvement of the rendering engine.

2.4.1 The Octree

An octree [Samet89] structures the scene data as a tree. Each node of this tree can have up to eight children and represents a box region of three-dimensional space. Each node also has a list of scene objects which lie completely or partly in the box region of the node. Figure 2.4 shows the octree of a scene containing two objects.

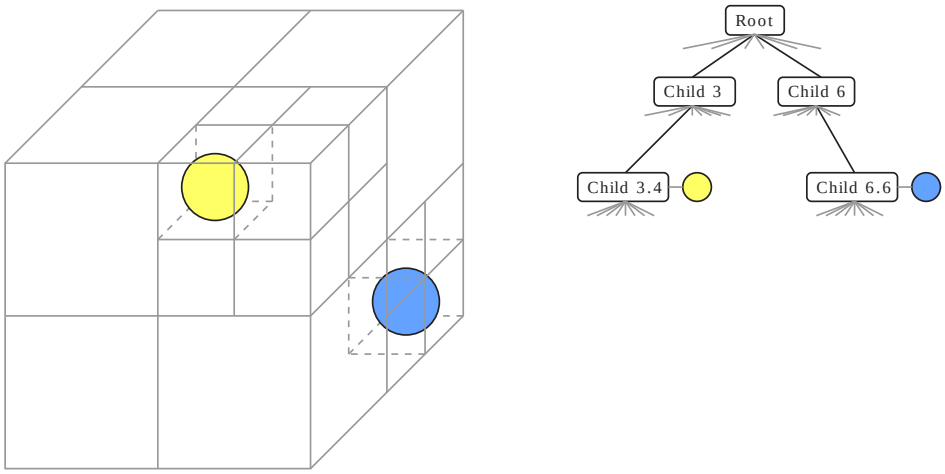

Figure 2.4: The octree of a scene containing two objects. The left image shows the division of space and the right image the internal tree structure.

The octree is built by starting with the root node which represents the bounding box of the whole scene. With each object added to the octree the bounding volume of the object is tested against the node's box. If the bounding volume intersects that box the children of the node are asked for an intersection test. This is done until the children are too small to split further, or the bounding volume of the object is bigger than the box of a child.

Many different implementations of the octree concept exist. With some of them it is allowed to add objects to interior nodes of the octree, with others it is only allowed to

add the objects to the leaf nodes. Additionally, an octree can split the added objects if they intersect more than one box or the octree stores an object-reference for each node which contains that object.

2.4.2 The kd-tree

The kd-tree [Samet89] is similar to the octree presented in the previous chapter except that it is a *binary* tree. Only planes which are perpendicular to one of the coordinate system axes are used as splitting planes. Moving down the tree, the axes used to select the splitting planes are cycled through. As in Figure 2.5 the first plane is perpendicular to the Y axis and the second plane is perpendicular to the Z axis.

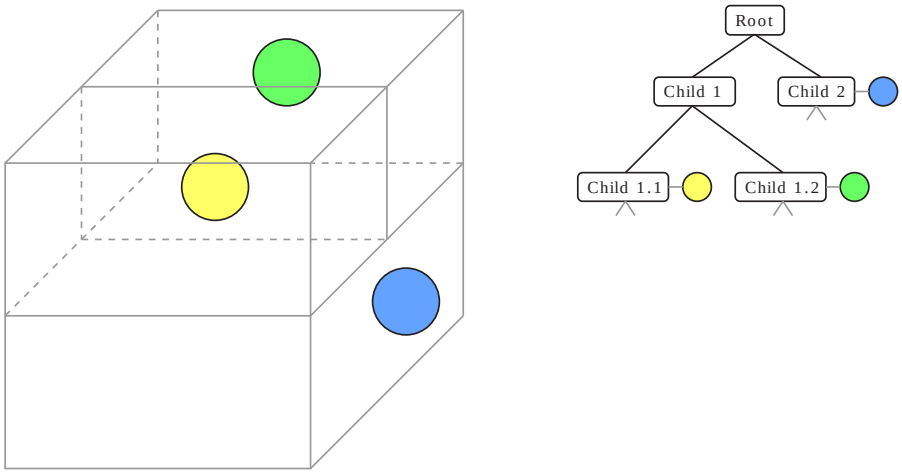

Figure 2.5: The kd-tree of a scene containing three objects. The left image shows the division of space and the right image the internal tree structure.

The kd-tree is built by starting with an empty root node which represents the bounding box of the whole scene. This box is than divided by its two children into a lower and an upper half-box. The next split is done along another axis.

There exist many algorithms to decide where to place the splitting plane and when to stop this process. One of this algorithms is the *surface area heuristic* [MacDo90] which finds the position of the splitting plane by minimizing a cost function which estimates the cost of a ray traversing through the kd-tree.

Two of the termination criterias for the splitting process as used by many algorithms are listed below:

- The process is stopped when the graph depth of the box is higher than or equal to a fixed constant.
- The process is repeated until every splitted box contains only a certain number of objects.

2.5 The Scenegraph

A scenegraph is a graph which organizes the objects of a scene in a hierarchical way. In contrast to octrees and kd-trees, scenegraphs store not only the geometry of objects. Since the scenegraph has to store all relevant data of a scene, it stores also the transformations, material parameters, textures, bounding volumes, lights, cameras and effects of each object in the scene as illustrated in Figure 2.6.

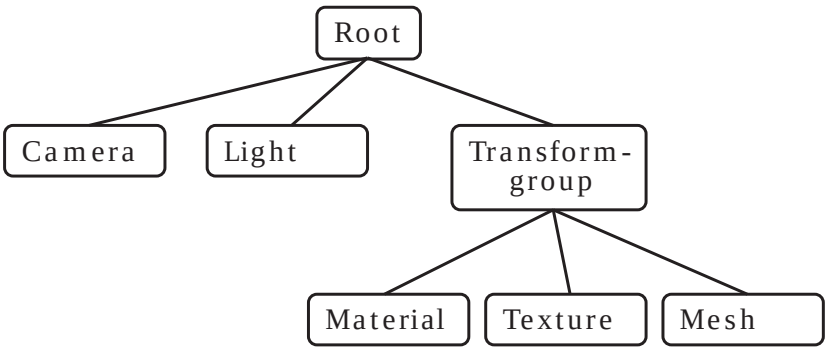

Figure 2.6: An example of a scenegraph containing different kind of node types

To render the scene, each node in the graph has to be traversed. This traversal usually needs some sort of *state* to remember the rendering data a node has changed (e.g. renderstates and transformations). This state object is *carried* to every node to let the current node change the state. The state object is also able to revert the changes a node has applied. Usually the scenegraph is traversed in a depth-first order, but this depends on the implementation of the scenegraph. Every visited node can change the current state of the graphics device, e.g. a material node would apply its parameters to the graphics device using a graphics API like OpenGL. If the visitor traverses a geometry node, a render call for that geometry is invoked.

An important difference between scenegraph implementations is how state changes are propagated at group nodes. Some scenegraphs propagate the changes to the children of a node and also to the right neighbour of a node. This makes it easy to assemble a scenegraph whose nodes share the same attributes. These attributes can be placed at the *left-top* of the graph and are propagated to the whole graph during the traversal. Other scenegraphs only allow state propagation to children of a node. This makes parallel traversal implementations easier.

As with some scene graph implementations, not every information has to be stored as an explicit node in the graph. E.g. information like materials and textures can be stored as properties of another node. These properties are often called *node-components*.

To store the transformations of the objects to each other and to animate their positions, the scenegraph also provides a special kind of group node. This group node (often called

transformgroup) stores the transformation matrix which gets applied to all its children during the traversal.

If the transformation value of a transformgroup changes, this value has to be propagated to its children. But since this change invalidates the bounding volume of the transformgroup and its children, the bounding volumes of the parents towards the root node have to be updated too.

2.5.1 The Bounding Volume Hierarchy

One of the main reasons to use a scenegraph is that its hierarchical layout can be used to increase rendering performance by culling. Since every geometry node has a bounding volume which encapsulates its position vectors, a *bounding volume hierarchy* (BVH) can be built out of the nodes of the graph. In Figure 2.7 the BVH of a small scenegraph can be seen. The bounding volume of a *group node* encapsulates all bounding volumes of its children. While traversing the scenegraph for rendering, the bounding volume of every visited node is tested against the viewing frustum of the camera. If the node is outside this frustum, the node itself and all its children are stated to be *culled* and are not traversed further. Using this algorithm (also called *hierarchical view-frustum culling* [Clark76]) the overall rendering performance can be increased.

Since the scenegraph provides only a logical organization of the nodes, an additional spatial data structure could provide better culling results (concerning the performance of the culling process) than the bounding volume hierarchy of the scenegraph. Examples for such data structures are given in Section 2.4. Using the culling implementation presented in this thesis (see Section 3.9 for details) it is possible to use more than one spatial data structure to organize the nodes of the scene.

2.5.2 Data Sharing

An important aspect of scenegraphs is how data can be shared among its nodes. This is especially important if the scenegraph contains thousands of mesh nodes which all have the same geometry.

One popular approach is to allow nodes to have multiple parents. Thus, the data of the shared subgraph can be referenced multiple times as illustrated in Figure 2.8. Since this concept complicates handling of the scenegraph, some frameworks only allow sharing of leaf nodes. Or they introduce a special class of nodes which handle this feature. E.g. Java 3D [J3D] (as presented in the next section) introduces the *link* and *shared group* node, where a link node points to one instance of the shared group class. A simple scenegraph using this feature is illustrated in Figure 2.9.

Other scenegraph frameworks encapsulate the data in so called *node components* which can be referenced by multiple node instances. Node components are not derived from

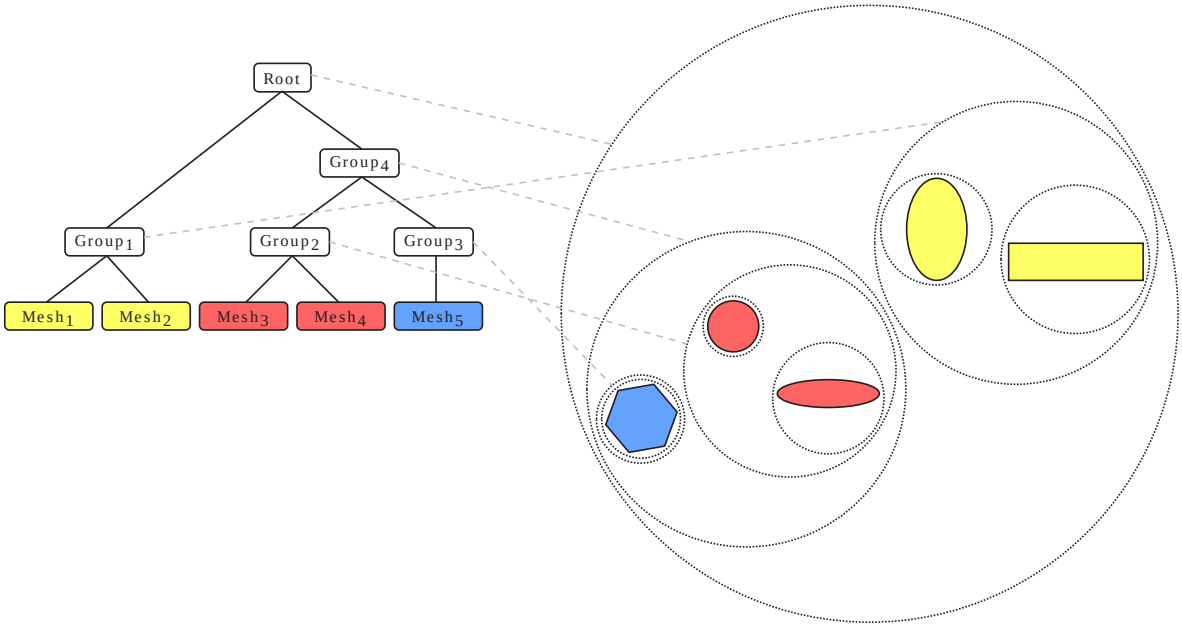

Figure 2.7: The bounding volume hierarchy (on the right) of a small scenegraph (on the left)

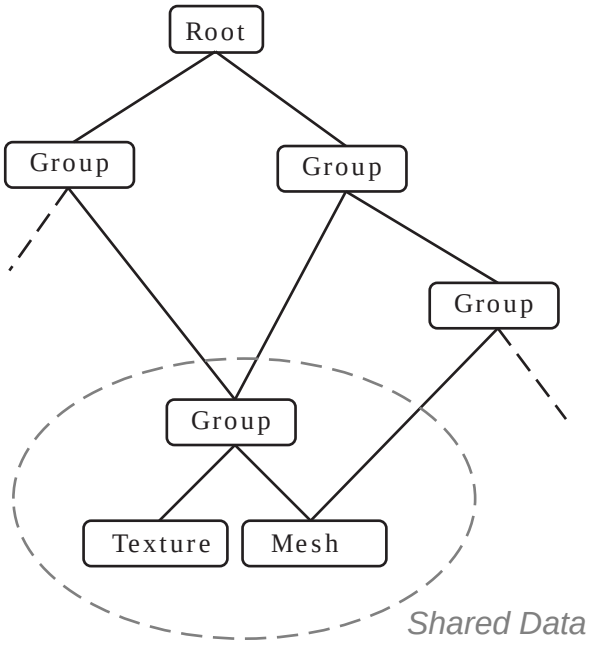

Figure 2.8: A scenegraph containing nodes with multiple parents

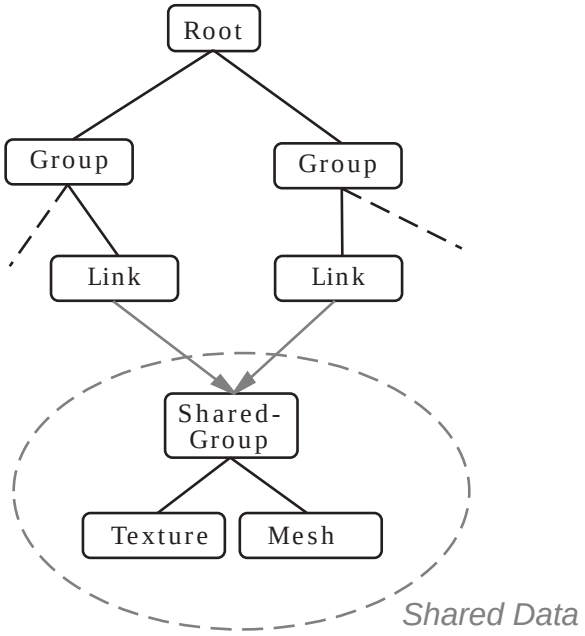

Figure 2.9: A scenegraph using link nodes and a shared group

the node class and therefore cannot directly be integrated into the scenegraph but only internally referenced by nodes. This approach simplifies the handling of the scenegraph. Its disadvantage is that for each logical instance of the data a node has to be constructed which references the shared node component, causing an overhead at the traversal. A scenegraph showing the usage of node components is illustrated in Figure 2.10.

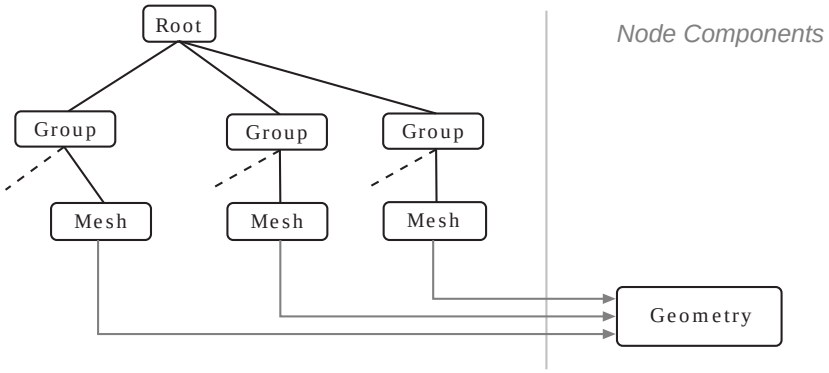

Figure 2.10: A scenegraph using node components to encapsulate its data

2.5.3 Classical Scenegraph Frameworks

A popular scenegraph framework is *Open Inventor* [OpInv] which is an open-source scenegraph API written in C++. It is provided by *SGI* and built on top of OpenGL. Therefore, it is supported on a wide range of platforms and window systems. It also defines its own file format.

Another available scenegraph API is *Java 3D* [J3D], which consists of a library of Java [JAVA] classes and was originally developed by Sun Microsystems in 1997. Since 2004, Java 3D has become an open-source project. Since version 1.4 (available since 2006), shader development is supported by Java 3D too. In contrast to Open Inventor, direct access to the underlying graphics API (OpenGL or Direct3D) is not intended. At the moment, Java 3D is available for Microsoft Windows, Linux, Mac OS and Solaris.

OpenSG [OSG] is a real-time rendering system featuring a scene graph with multithreading and clustering support. Its authors claim that OpenSG is very extensible and works on different Unix systems and Microsoft Windows.

OpenSceneGraph [OScGr] is another high performance 3D graphics toolkit. It features a multithreading aware scene graph. It is mainly used in the fields of visual simulation, games, scientific visualization and virtual reality. It is written in Standard C++ and uses OpenGL to run on Microsoft Windows, Mac OS, Linux, Solaris, HP-Ux, AIX and FreeBSD.

More rendering frameworks containing a scenegraph implementation are listed in Section 2.7.

2.6 Design Patterns

Design patterns in the context of software engineering describe simple and elegant solutions to problems in object-oriented programming. Using the design patterns presented in this section will result in flexible, modular, reusable and understandable application code layouts. More details on the presented design patterns can be found in the dedicated book of Gamma et al. [Patterns95].

Abstract Factory Pattern

With the abstract factory pattern, an interface (*AbstractFactory*) is provided which can be used by clients to create families of related objects without knowning about the concrete class which gets instanced. The client sees only the interfaces (*AbstractProducts*) of the created objects. Only the classes (*ConcreteFactory*) implementing the AbstractFactory interface know which classes they have to instantiate (*ConcreteProducts*).

Prototype Pattern

With the prototype pattern, a client can create instances of a concrete class by calling a special method of an existing object (*Prototype*). This method (often named `Clone()`) creates a copy of the prototype and returns this new instance to the client. The client does not need to know which concrete class is instanciated by the implementation (*ConcretePrototype*) of the prototype interface.

Singleton Pattern

The singleton pattern should be used if only one instance of a class is allowed within a sytem. The class itself is responsible to provide a global access point to its instance. This is accomplished by providing a class method which creates the global instance if this has not been already done and returns it to the caller. Further, the constructor of this class is declared *private* to ensure no one else can make an instance of it.

Adapter Pattern

The adapter class implements a known interface (*Target*) used by the client by *wrapping* another (unknown to the client) interface. Using this design pattern classes can work together which have incompatible interfaces. The adapter is also known as wrapper.

Bridge Pattern

The bridge pattern decouples an abstraction from its implementation by putting them in separate class hierarchies. This has the benefit that the two can vary independently. The abstraction class contains a reference to an instance (*ConcreteImplementor*) implementing the *Implementor* interface. Classes which extend the abstraction class (*RefinedAbstractions*) also use this reference to accomplish their functionality.

Proxy Pattern

With the proxy pattern the client uses a placeholder for another object. This surrogate controls the access to the other object. A popular example of the proxy pattern are reference counted pointers (also called *smart pointers*). Here, the client has only access to the smart pointer instance (*Proxy*), which controls the destruction of the real pointer by counting the references to it.

Command Pattern

The command pattern encapsulates client requests as objects which can be e.g. stored, queued and logged. These objects (ConcreteCommand) support the Command interface and define a binding between a Receiver object and an action. It is the task of the client to instantiate this command object and to set its receiver.

Observer Pattern

With the observer pattern a list of *observer objects* can register themselves with the so-called *subject* object. Whenever the subject changes its state, it notifies the observers which in turn will synchronize their internal state with the state of the subject.

Template Method Pattern

With the template method pattern the global structure of an algorithm is defined in a method of an object. This method calls other abstract methods whose subclasses of the object can override to provide concrete behavior of the implemented algorithm.

Visitor Pattern

With the visitor pattern the operations which are performed on the elements of an object structure are decoupled from the nodes themselves. Therefore, the introduction of a new operation does not force changes to the *node* classes it operates on. The operations are provided by concrete classes (*ConcreteVisitor*) of an abstract parent class (*Visitor*). The visitor class declares an operation for each node class. The client passes the visitor to the object structure. If the element of the structure *accepts* the visitor, it calls the corresponding method for its class and passes itself as parameter to that method. The concrete visitor performs its concrete operation on the element and afterwards, the visitor is sent to the next element of the structure. New functionality is introduced by defining new visitor subclasses.

2.7 Comparison of Available Rendering Engines

This section gives a comparison between available rendering engines. The list contains only free available rendering engines because in general it is not allowed to give inside information on commercial engines. The rendering engines are compared based on the topics covered previously in this chapter.

OGRE 3D

- *Reference*: [OGRE]
- *Version*: 1.4.0
- *Platforms*: Win32, Linux, Mac OS X
- *Graphics APIs*: OpenGL, Direct3D
- *Shader Support*: Assembler, Cg, GLSL, HLSL
- *Effect Handling*: Provides own material and effect file format similar to CgFX and HLSL, which supports the definition of multiple passes, renderstates and used textures.
- *Spatial Data Structures*: No built-in structures but provides a plugin system which allows the user to register own implementations with the engine. Example plugins provide Octree and BSP-tree functionality.
- *Scenegraph*: Provides only one node class to hold any kind of data the user wants to structure as a graph. The nodes are handled by a scene manager which can be implemented by user-defined graph layouts like octrees or BSP-trees.

NVSG

- *Reference*: [NVSG]
- *Version*: 3.2.0.10
- *Platforms*: Win32, Win64, Linux
- *Graphics APIs*: OpenGL
- *Shader Support*: Cg, CgFX
- *Effect Handling*: Effects and materials are directly assigned to objects in the scene. No special support for global multipassing is provided.
- *Spatial Data Structures*: none
- *Scenegraph*: The scenegraph supports several node types like groups, lights, cameras and meshes. Data sharing between nodes can be accomplished by either letting multiple nodes directly reference the shared data or reuse the same node multiple times. This is possible since all nodes support multiple parents. NVSG uses the traversal concept of visitors to perform operations like culling and rendering on the nodes.

Irrlicht

- *Reference*: [Irrlicht]
- *Version*: 1.2

- *Platforms*: Win32, Linux, Mac OS, Solaris
- *Graphics APIs*: OpenGL, Direct3D and two software renderers.
- *Shader Support*: Assembler, HLSL, GLSL
- *Effect Handling*: Effects and materials are directly assigned to objects in the scene. No special support for global multipassing is provided. The shadow volume effect as well as transparency effects are hardcoded into the rendering process.
- *Spatial Data Structures*: Octree implementation provided. But it is only a special type of scenegraph node and not a different view of the whole scene data.
- *Scenegraph*: The scenegraph supports lights, meshes, cameras, billboards, skyboxes, terrains and octrees using special node classes. Shadow volumes are integrated into the scenegraph using its own node class too.

OpenSG

- *Reference*: [OSG]
- *Version*: 1.6
- *Platforms*: Win32, Linux, Solaris
- *Graphics APIs*: OpenGL
- *Shader Support*: Assembler, GLSL
- *Effect Handling*: Material nodes are added to the scenegraph which specifiy the necessary OpenGL states to render a geometry. Multipassing is implemented by adding a node several times into the *draw tree* (see below for details on this tree).
- *Spatial Data Structures*: None
- *Scenegraph*: Multitheading safe implementation of a scenegraph provided. Data can be shared through a *node component* like concept. OpenSG uses the visitor pattern to perform operations on the nodes. For rendering the scenegraph is first traversed to cull out invisible objects. Afterwards a special kind of tree (the *draw tree*) is created for sorting and later rendering.

OpenSceneGraph

- *Reference*: [OScGr]
- *Version*: 1.2
- *Platforms*: Win32, Linux, Mac OS X, Solaris
- *Graphics APIs*: OpenGL
- *Shader Support*: Assembler, GLSL, Cg

- *Effect Handling*: Effects are integrated into the scenegraph as special node classes where one effect can only operate on one node.
- *Spatial Data Structures*: none
- *Scenegraph*: A lot of different node classes are provided by the scenegraph. The classical visitor pattern is used to operate on the nodes. Each node class has direct access to the underlying OpenGL API. The scenegraph is multithreading-aware.

3 Designing the Rendering Engine

3.1 Why another Rendering Engine?

Before this thesis, another rendering engine called YARE (Yet Another Rendering Engine) was used at the *Institute of Computer Graphics and Algorithms* of the Vienna University of Technology. The first version of this engine was developed for a project called *UrbanViz* whose aim is to provide algorithms for fast walkthroughs and visualizations of large urban environments. See [WonkaSchmalstieg99] and [WimmerEtAl99] for more details on this project.

YARE mainly consists of a scenegraph-based API written in C++ and uses OpenGL for rendering. The provided API of YARE is very similar to Java3D. Thus, the pros and cons of YARE are similar to those of Java3D:

Advantages of YARE:

- It is easy to get into YARE because of its scenegraph-based layout.
- It is easy to implement new scenegraph nodes for new functionality.
- The developer has full access to the OpenGL-API within the scenegraph.

Disadvantages:

- No support for multitexturing and vertex- or fragmentprograms by default.
- No support for global multipassing.
- YARE hides details of the rendering-pipeline from the user. This makes it hard to implement new rendering-techniques which require access to some pipeline stages.

The main reason for implementing a new engine was the lack of a complete effect framework. Such an effect framework which supports high-level shaders and multitexturing and handles local and global multipassing by itself was needed. A complete list of requirements can be found in Section 3.2. Integrating such an effect framework into YARE would have required almost a complete rewrite of the engine. Therefore, it was decided to build a completely new rendering engine. Also as Chapter 2 shows, existing engines are not able to satisfy the requirements on the effect framework.

The chosen name of the new rendering engine is **YARE 2.0**.

3.2 The Requirements

The requirements for YARE are a combination of:

- What is needed for the development of new rendering algorithms and techniques.
- What features are offered by modern graphics devices.
- What are standard features of existing rendering engines.

The most important requirements for YARE are listed below. The list is not sorted by any criterion.

- **Performance optimized data storage**: To store static geometry data directly on the graphics device, e.g. in an OpenGL vertex buffer object or display list.
- **Effect framework**: To provide a framework where standard effects like normal mapping and shadow mapping can be chosen from and new effects can easily be integrated (for rapid prototyping). This effect framework should be flexible in terms of number of lights and number of used textures. It should also be possible to add additional effects to all objects in a scene which in turn should merge with the effects an object already contains. Another important feature is that there should be a way to easily get a depth-only version of an effect.
- **Vertex and fragment programs**: To use Cg, GLSL or HLSL programs for rendering.
- **Post-processing effect framework**: To be used for tonemapping, blooming and other effects which operate on fragments of the framebuffer.
- **Multiple rendertarget support**: The rendering engine should provide a way to specify more than one rendertarget to support rendering techniques like deferred rendering [WhittedWeimer81] [DeeringEtAl88] [Ellsworth90].
- **Model importers**: At least the following formats should be supported: OpenInventor, 3ds and Ply.
- **Image format support**: At least the following image formats should be supported: Bmp, Jpeg, OpenEXR and Png.
- **No hardcoding of shadow effects**: Shadowing techniques like stencil shadow volumes and shadowmaps should not be hardcoded into the engine. Shadowing should be handled like any other rendering effect.
- **Local and global multipassing**: Support effects that need local or global multipassing. But the engine itself should handle the passes needed by an applied effect.
- **Global algorithms**: New global algorithms/techniques like culling of the scene, sorting of renderable objects should be easily includable into the engine.

- **Documentation and Examples**: A full documentation should be provided, as well as examples for all important features of the rendering engine.
- **Scenegraph algorithms**: To provide an easy way to access scenegraph data as well as traversal of the scenegraph to enable development of new scenegraph algorithms.
- **State management**: To allow developers to easily set complex renderstates. This is in contrast to just being able to set predefined material properties.
- **Scripting**: The engine should be scriptable using a common script language like Lua[Lua], Python [Python] or JavaScript.
- **API independent**: The rendering engine should be graphics API independent. Therefore, it should be possible to use OpenGL and Direct3D for rendering.
- **Occlusion queries**: The engine should provide hardware occlusion queries to be able to accelerate rendering.
- **Editor**: Provide a simple editor for creating a scenegraph and applying effects to objects.

3.3 The Layers of the Engine

To group the functionality of the rendering engine by topic and software level several modules organized as layers are introduced. These modules are called *Core*, *Graphics*, *GraphicsGL* and *GraphicsD3D*. The *Core* module works at the lowest software level of the engine and provides submodules like system functions, a general application framework and common utility functions. The *Graphics* module is built on top of the *Core* module and provides an interface to 2D and 3D graphics. Among other submodules *Graphics* contains a graphics-API-independent rendering interface, an effect framework, a scenegraph and importers for several 3D model file formats. *GraphicsGL* and *GraphicsD3D* implement the graphics-API-independent rendering interface provided by *Graphics*. This implementation builds the bridge between the rendering interface and OpenGL respectively Direct3D. For a complete listing of the submodules of Core and Graphics with a short description see Table 3.1 and Table 3.2. For a more detailed view on the submodules see the following chapters.

Figure 3.1 shows the structure of a typical application using the YARE rendering engine. The modules of YARE are kept in red in the figure.

Name of Submodule	Short Description
Platform	A set of header files defining platform- and OS-dependent data types.
Debug	Provides debugging functionality like logging, exceptions, assertions and callstack retrieval.
Reflection	Provides functions for querying interfaces, type descriptions and conversions, properties of classes with getter and setter methods. This module also provides a reference-counted base-interface class.
Io	Provides classes to support any kind of input and output of the application like files, mouse and keyboard.
Data	Provides a chunk-based resource management system.
Math	Classes and structures for geometric mathematical operations.
Engine	Provides an application framework and an actor concept implementation (See Subsection 3.4.5 for details).
Scripting	Exports all properties of actors and utility functions to a script language like Lua [Lua].
Drawing	Provides classes for loading, processing and writing different kinds of images.
Utility	Contains often needed classes like a cache, thread, events, critical sections and string utilities.

Table 3.1: The submodules of the *Core* module

To minimize the dependencies between the submodules and to put them in order, it is not allowed to include every header file in each arbitrary submodule. There is a definition called *Include Guide* that gives information about which submodule is allowed to include which other submodule. Figure 3.2 shows a graphical representation of the *Include Guide*, where the submodules of a layer are allowed to include submodules of the same or a lower layer.

Name of Submodule	Short Description
Variables	Concept of an abstract data container and its manipulation
Rendering	Provides graphics-API-independent interfaces for common rendering operations
Gui	Provides style-based 2D Gui widgets like forms, buttons and sliders
Asset	Provides importers for 3D model files
Effect	Provides an effect framework for renderable objects
Renderer	Handles the rendering of objects by feeding the rendering interface
DrawGraph	Data structures for accelerating rendering through culling
Dag	The scenegraph module of YARE
Utility	Provides utility classes like a texture cache

Table 3.2: The submodules of the *Graphics* module

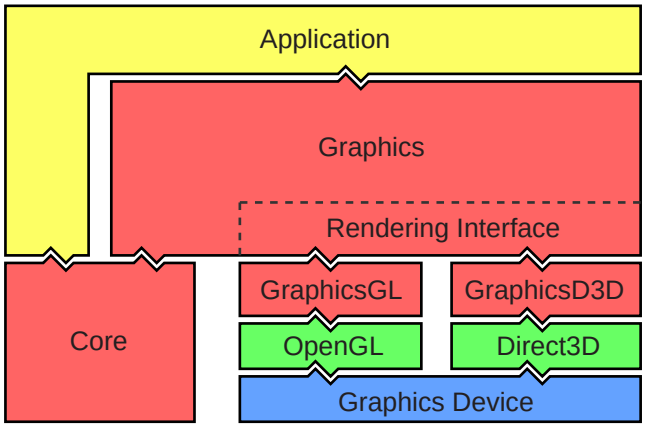

Figure 3.1: Structure of a typical application using YARE

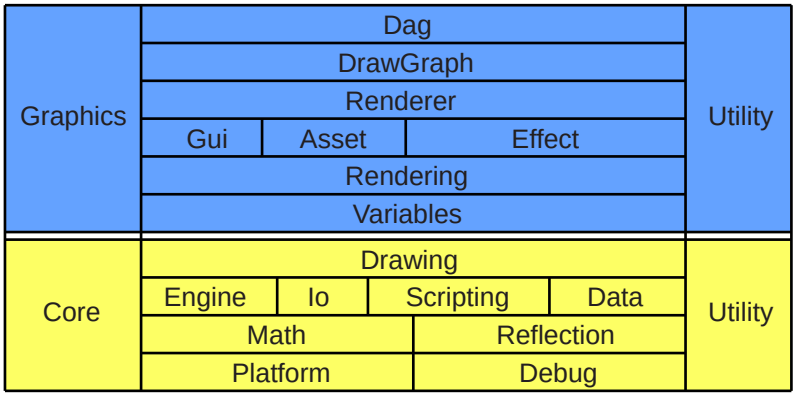

Figure 3.2: Graphical representation of the *Include Guide*. It is only allowed to include submodules of the same or a lower layer.

3.4 The Core Module

The following sections provide information about the most important submodules of the *Core* module.

3.4.1 The Math Framework

The Math Framework contains all the math classes needed for a graphics-related software like Yare. It provides all the definitions and operations needed for vector math, interpolation and volumes.

The following floating-point tupel types are supported with single and double precision:

- Vec2
- Vec3
- Vec4
- Matrix4x4
- Quaternion
- Plane
- Line
- Ray
- Spline

The following interpolation methods are supported:

- Linear
- Bezier
- Tension - Continuity - Bias

The Math Framework can also handle keyframe-animation of all interpolation types.

This module can create simple meshes (position vectors and indices for triangle lists) of the following types:

- Box
- Cone
- Cylinder
- Sphere
- Plane

It also provides intersection tests for common primitives and volumes. Table 3.3 shows the complete matrix of possible intersection test. An asterix marks the implemented intersection routines of YARE.

	Point	Ray	Line	Plane	Triangle	Box	Sphere	Cone	Frustum
Point	*	*	*	*	*	*	*		*
Ray	*			*	*	*	*		*
Line	*			*	*				
Plane	*	*	*	*					
Triangle	*	*	*						
Box	*	*				*	*		*
Sphere	*	*				*	*	*	*
Cone							*		
Frustum	*	*				*	*		*

Table 3.3: Implemented intersection routines in the math submodule

3.4.2 The I/O submodule

The I/O submodule provides classes to support any kind of input and output of the application. It provides a codec-manager to handle different codecs of resources in a uniform way. A codec is an algorithm encapsulated in a class which can read and/or write data in one specific format. An example for a codec is the *XML-codec*, which is able to read and write XML files. It supports filestreams and memorystreams and also handles user-input from common input-devices like keyboard and mouse.

3.4.3 The Data Interface

The data submodule provides a chunk-based resource-management system. A resource manager allows loading any resource from disk. First, the resource manager tries to find an importer for the given type of resource and then passes the work to this importer. All resources are cached and will not be loaded twice.

The data module also provides built-in support for document-based resources like XML files (text based and binary optimized XML files are supported). A document in this context is simply a tree of data chunks.

A data chunk has the following attributes:

- A unique name.
- A value as string.
- An attribute map with a string as key and a string as value.
- A list of child chunks.

The data-module strongly depends on the I/O submodule (as described in Subsection 3.4.2) to load or write data to the harddisk.

3.4.4 The Reflection System

The tasks of the reflection system are the following:

- It allows to retrieve a type descriptor for any object in the engine.
- The type contains an ID as string and provides methods for creating another instance of that type.
- The type stores a link to a parent type.
- It also provides methods for querying for interfaces.
- It enables the developer to convert a type ID to the type-class and vice versa.
- The developer can specify properties of types by using macros to define get- and set-methods to member variables of a class.

The core of the reflection system is the `IType`-interface which provides most of the functionality listed above. Using macros for declaring user-defined types and their properties inheritants of the `IType`-interface are created implicitly.

At runtime developers can create objects by passing the type ID and configure the new object by passing a chunk-based data tree.

The following listing gives an example on how to use the macros of the reflection system to define a type.

```
// In the header file:
class Light : public Leaf
{
DECLARE_ACTOR( Light ) // This macro defines the new type.
...


// In the implementation file:

// Implement that type and inherit all properties of the parent type
```

```
IMPLEMENT_OBJECT_PARENT_BEGIN(Light,Leaf)  'Leaf'.
PROPERTY_DECLARE("Intensity") // Define some properties.
PROPERTY_DECLARE("Attenuation")
IMPLEMENT_OBJECT_PARENT_END(Light,Leaf)

// Define member variables and getter methods for the properties.
PROPERTIES_BEGIN_GET( Light )
// Direct access to a member var:
PROPERTIES_ADD_GET( "Intensity", mIntensity )
// First call the getter method, then fetch the value from the member var:
PROPERTIES_ADD_GET_CALL( "Attenuation", mAttenuation, RecalcAttenuation() )
PROPERTIES_END_GET()

// Define member variables and setter methods for the properties.
PROPERTIES_BEGIN_SET( Light )
// Direct access to a member var:
PROPERTIES_ADD_SET( "Intensity", mIntensity, Vec3f )
// First assign the value to the member var then call the setter method:
PROPERTIES_ADD_SET_CALL( "Attenuation", mAttenuation, Vec3f, UpdateProps() )
PROPERTIES_END_SET()
```

3.4.5 The Engine Framework

The engine framework implements the *actor concept* of Yare. An actor is a class, which can be instanced by just providing its type ID and its properties are configured by a data chunk. E.g. by providing an XML-representation of a chunk, an actor and all its children can be instanced and configured. The actor class is also the base class for all published classes, e.g. classes which should be accessible from a scripting language or which should be serializable. There are some predefined actors for loading XML description files and plugins.

This submodule also provides a simple framework by which a developer can create applications that have a main-window and are configured by an XML description file.

3.4.6 Making the Engine Scriptable

The scripting module exports all actors defined in Yare to the LUA scripting engine. This includes all defined properties, methods and eventhandlers of a class. No extra code to export a user-defined class is needed. The only task needed is to inherit from the Actor-class and define all properties through the macros described in Subsection 3.4.4.

3.5 The Graphics Pipeline

This chapter gives a global overview of how and where graphics data is defined in YARE. It also describes how this data is processed to generate an image of the scene on an output device.

The scene data in this context consists of:

- Meshes containing vertex attributes like position and normal vectors.
- Texture maps.
- Light parameters, e.g. direction, position and intensity.
- Camera parameters, e.g. position, direction and field of view.
- Effect parameters, e.g. diffuse material coefficients, specular exponent and texture coordinate scale.

The operations and algorithms performed on the scene data include:

1. Convert the meshes to renderable objects. Details described in Subsection 3.10.6.
2. Assign the effects of the scene to the renderable objects. Details described in Subsection 3.10.6.
3. Find visible objects with respect to a culling object like a camera. Details described in Section 3.9.
4. For rendering put the objects into the correct order. Details described in Subsection 3.8.6.
5. Assign the values of light and camera parameters to the renderable objects. Details described in Subsection 3.8.5.
6. Send the drawing commands to the graphics device to render the objects. Details described in Section 3.7.

The graphics pipeline of YARE is presented in the following sections. An illustration of the pipeline stages can be found in Figure 3.3.

3.5.1 The Scenegraph

The scenegraph as presented in Section 3.10 is used to organize the objects of a scene in a hierarchical way. The user can create different kinds of nodes to arrange the meshes, lights and cameras of a scene. The rendering effects are not directly applied to the meshes but are distributed to the scene by creating instances of special node classes.

With every update of the scenegraph its renderable objects are synchronized with the underlying drawgraph as presented in the following chapter. Renderable objects in this context consist of a geometry object along with an effect defining the way this geometry is rendered.

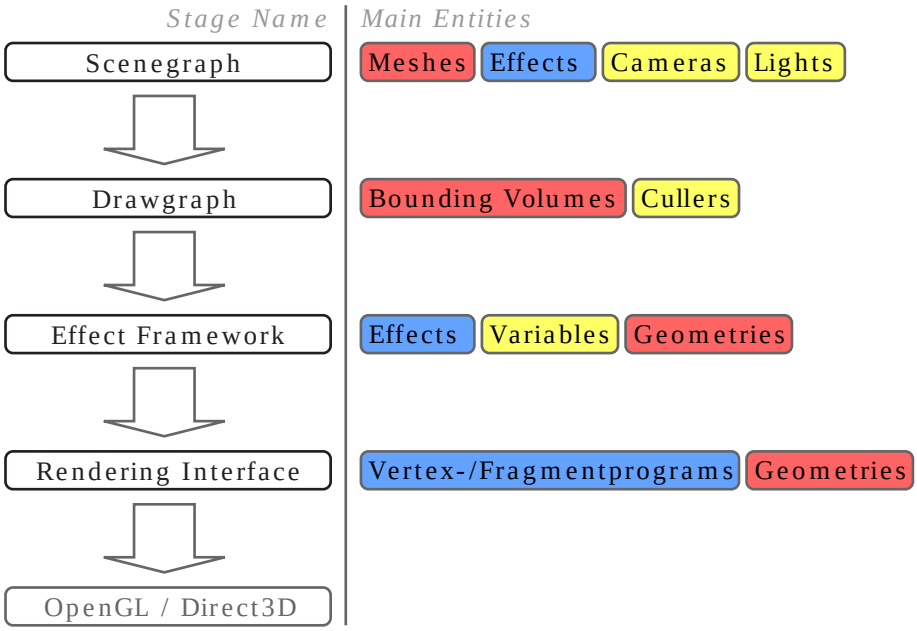

Figure 3.3: The graphics pipeline of YARE

3.5.2 The Drawgraph

The task of the drawgraph is to provide an efficient way to find the visible items with respect to culling objects like e.g. a camera. Details on the drawgraph can be found in Section 3.9. For every rendering frame the drawgraph is asked for all visible objects. The control is passed to the effect framework by commanding the effect of the object to render the assigned geometry.

3.5.3 The Effect Framework

The effect framework as described in Section 3.8 takes a provided rendering pass and updates the parameters of it. For rendering a geometry, the vertex- and fragment programs of this pass are activated and the data of the geometry are sent to the graphics device. All commands sent to the graphics device are done via the rendering interface presented in the next section.

3.5.4 The Rendering Interface

The task of the rendering interface is to convert the graphics commands called by the stage above into graphics API (e.g. OpenGL) specific commands which can be sent

to the graphics device. A more detailed view on the rendering interface is given in Section 3.7.

3.6 The Variables Concept

3.6.1 What is a Variable?

Each data related to graphics in YARE is stored in a container called *variable*. A variable consists of an array of data elements like floats, integers, matrices or strings and the description of the variable. The description of the variable contains information about the variable such as the name, the number of elements and sharing attributes. Examples for often used variables are:

- Vertex Positions,
- Texture Coordinates,
- Indexbuffers,
- Transformation Matrices,
- Light Directions,
- Camera Positions,
- Material Properies.

For standard variables (as named in the list above) predefined variable descriptions are provided.

For a detailed description of every field of the `VariableDesc` structure see Table 3.4.

3.6.2 Creating a Variable

The only thing needed for creating a variable is its description. This description is handed over to the manager of a collection of variable factories which picks the correct variable factory (the one which can create a variable with the given description) for creating a matching variable. New variable factories can be added to the system dynamically by registering and unregistering implementation classes of the interface of all variable factories (`Yare::Graphics::Variables::IFactory`). The system default factory can create variables with data in the main memory of the PC and without special behavior. With the OpenGL-implementation of the Rendering Interface (see chapter 3.7 for details) an additional factory will be registered. This factory can instantiate variables with data lying on the graphics device and direct use for rendering. An additional variable factory could (not implemented at the moment) create variables containing data needed for physics calculations and thereby providing a clean interface to a third party physics simulation framework. Variables also support direct cloning of themselves by calling the `Clone()`-method.

Fieldname	Description
name	The name of the variable, e.g. `"VertexPosition"`.
count	The number of elements the corresponding variable contains.
index	Indicates the usage index of the corresponding variable. This is needed if a set contains more than one variable with the same name. An example are a set of texture coordinate variables, where every variable has a different usage index.
type	The C++ type name of the elements of the variable. See Subsection 3.4.4 for details on the C++ type name.
freq	Indicates the usage frequency of the corresponding variable. The data of a variable can be used per vertex, per primitive, per primitive group, per geometry or for the whole scene.
dynamic	Indicates whether the data of the corresponding variable will be respecified repeatedly or not.
shareable	Indicates whether the data of the corresponding variable can be shared among other variables or not.
geometryrelated	Indicates whether the data of the corresponding variable is related to geometry (e.g. vertex normals) or not (e.g. the direction of a camera).
additional	Stores some additional information about the variable, e.g. if the data of the corresponding variable is stored on the graphics device.

Table 3.4: Details of the *VariableDesc* structure

The following code snippet shows how a variable for vertex positions can be created:

```
using namespace Yare::Graphics::Variables;

// Create a variable description.
VariableDesc my_desc;
my_desc.name = "VertexPosition";
my_desc.type = Yare::Core::Reflection::GetCppTypename<Vec3f>();
my_desc.freq = VariableDesc::FREQ_VERTEX;
my_desc.shareable = true;
my_desc.geometryrelated = true;

// Create a compatible variable to that description.
// Internally uses the abstract factory concept.
IVariablePtr my_pos_var = CreateVariable(my_desc);
```

```
// Fill the variable data.
Vec3f *dataptr = (Vec3f*)my_pos_var->Lock(IVariable::LOCK_WRITE_ONLY);
for (uint32 i = 0; i < my_desc.count; i++)
{
  *dataptr++ = my_positions[i];
}
my_pos_var->Unlock();
```

3.6.3 What are the Benefits of a Variable Container?

An important reason for this design decision is to have a clean and common interface for all type of rendering data. Another advantage is the high degree of transparency provided to the user by hiding the graphics API specific implementation details. Furthermore, the developer does not need to know whether the variable data is lying in RAM or in video memory.

With the variable description concept arises the possibility to add meta information to the variable which in turn can be used to instantiate a variable with only its description.

The system allows the developer to register additional variable factories. This is implemented according to the *abstract factory design pattern* as descriped in Section 2.6.

3.6.4 Variable Manipulators

The variables module provides a manipulation concept allowing to manipulate or automatically create missing variables at runtime. The provided manipulators (all of them implementing the `Yare::Graphics::Variables::IManipulator` interface) can create the following variables:

- Vertex to Primitive Relation
- Primitive to Vertex Relation
- Primitive Center Vectors
- Per Vertex Normals
- Per Primitive Normals
- Spherically Mapped 2D Texture Coordinates
- Cylindrically Mapped 2D Texture Coordinates
- Cubically Mapped 2D Texture Coordinates
- Vertex Tangent and Binormal
- Binary Space Partitioning Data

3.6.5 Emitting Variables

Not only the user/developer of the system can instanciate and distribute variables, but specific objects in the system can, too. These objects are called *variable emitters* and their classes are all implementing one common interface:
`Yare::Graphics::Variables::IVariableEmitter`. Using this interface the objects can be asked which variables they emit and in which volume they do so. Examples for typical emitters can be found in Table 3.5. Thereby other objects are free to decide whether they want to *receive* the emitted variables or not. An example for an object receiving variables is a *rendering effect*. It receives the data of the scene in the form of variables. This data can include light properties and texture files.

The following emit modes are supported:

- *Global*: The object emits its variables to each other object in the system.
- *Structural*: The object emits its variables only to objects which lie in the same structure, e.g. in the same scenegraph group. A renderable object (as described in Subsection 3.9.2) retrieves only structural variables if it lies in the subgraph below or to the right of the emitter.
- *Volume Object Space*: The object emits its variables only to objects which lie in the emit-volume. This volume is given in object space of the emitter.
- *Volume World Space*: The object emits its variables only to objects which lie in the emit-volume. The volume is given in world space.

The concept of emitted variables is used in the scenegraph as described in Section 3.10.

In Table 3.5 some examples of variable emitters are provided.

3.6.6 The Interfaces

See Section A.1 for the C++ interfaces of this chapter.

Classname	Emit Mode	Emit Volume	Emitted Variables
DirectionalLight	`EM_GLOBAL`	`NULL`	Intensity, Attenuation, Direction
SpotLight	`EM_VOLUME_WORLD_SPACE`	frustum defined by the light cone	Intensity, Attenuation, Position, Direction, MinimumTheta, ViewMatrix, ProjectionMatrix, ViewProjectionMatrix
FileTexture	user selectable	user definable	Sampler2D

Table 3.5: Examples for variable emitters used in the scenegraph

3.7 The Rendering Interface

The main goal of the rendering interface is to provide an easy-to-use interface for coding 2D and 3D graphics. Even with more advanced rendering techniques like high-level shaders, the developer should not bother about graphics API specific details. Therefore, this interface abstracts features of the graphics device in a graphics-API-independent way.

The rendering interface itself provides interface classes for all common rendering operations like creating geometries and textures, setting rendering states and creating shaders. Those interfaces are then implemented by a rendering driver (like an OpenGL rendering driver) which can be loaded (and switched) at runtime. This concept makes the Graphics module completely independent from the underlying hardware and any graphics API (like OpenGL and Direct3D). The main C++ class is the device interface (`Yare::Graphics::Rendering::IDevice`) which is the entry point to the rendering functionality of YARE. This interface is also the factory class of important rendering objects like textures and geometries.

3.7.1 Details about renderable geometries

A geometry in YARE consists of a list of variables (see Section 3.6 for details) where every variable maps to a vertex attribute (e.g. position, normal or color). The geometry can also contain an index buffer variable into the attribute variables. Only one index buffer per geometry is supported, since more index buffers are not supported by graphics devices at the moment. A geometry must also contain variables indicating the primitive types and the number of elements used by each primitive type. Figure 3.4 shows a list of variables making up a geometry.

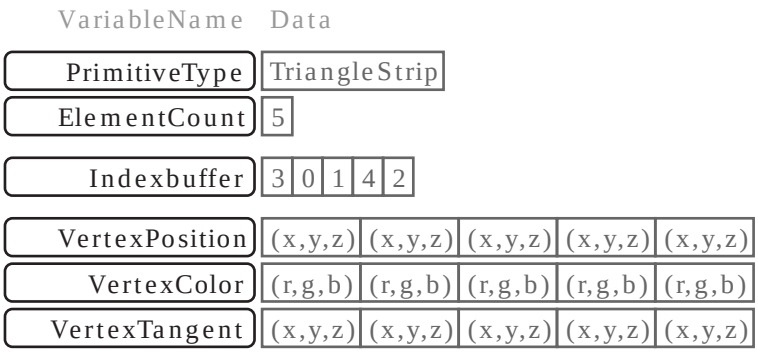

Figure 3.4: Variables forming a geometry

YARE is designed to handle more than one primitive type per goemetry. The following figures (3.5, 3.6, 3.7 and 3.8) demonstrate variable setups to build geometries containing

different primitive types. Each row illustrates one variable containing either the primitive type, the element count, vertex attributes or the index buffer.

On the one hand this flexibility of renderable geometry setups allows the developer to use directly imported data, on the other hand the data layout may not fit hardware constraints which can result in a significant performance penalty. At the moment, for optimal performance graphics hardware requires that geometries contain only one primitive type.

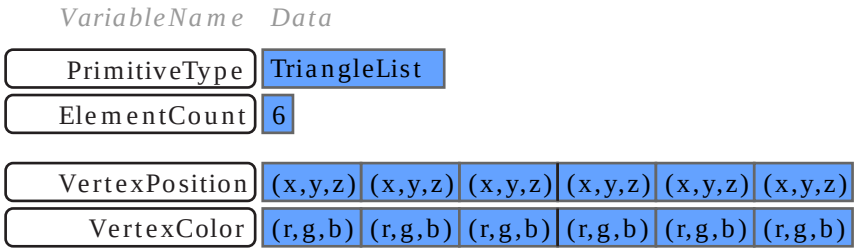

Figure 3.5: A geometry containing two triangles

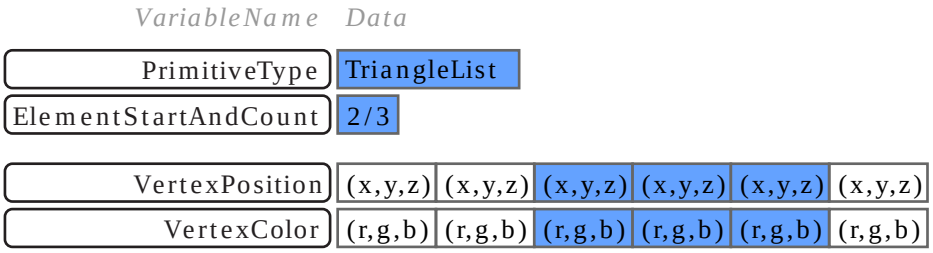

Figure 3.6: A geometry containing one triangle starting at offset 2

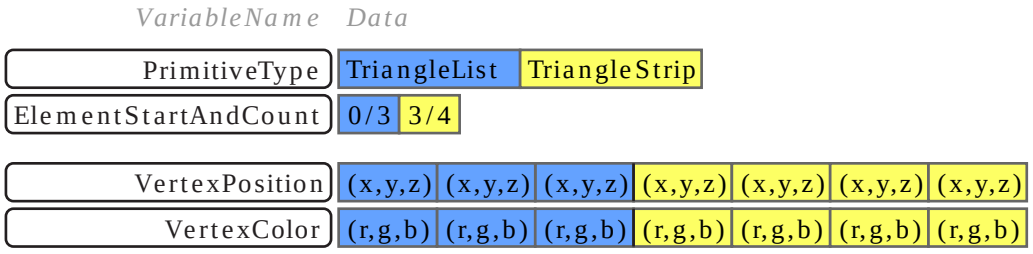

Figure 3.7: A geometry containing one triangle and a trianglestrip made up by 4 vertices

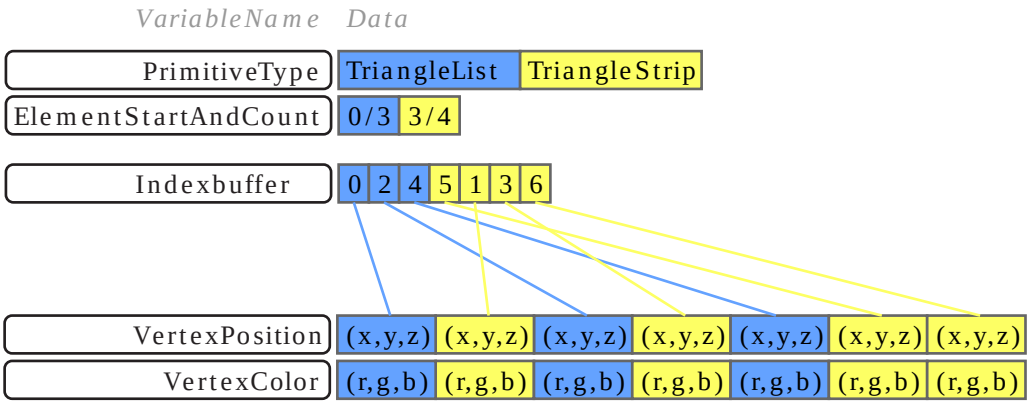

Figure 3.8: A geometry using an indexbuffer and containing one triangle and one trianglestrip

3.7.2 Vertex- and Fragment Programs

Using the `ICompiler`-interface, the user can retreive `IProgram`-instances by providing the source code of a high level shading language program.

The following high-level shading languages are supported by YARE:

- C for Graphics (Cg)
- Cg Effects (CgFX)
- The OpenGL Shading Language (GLSL)
- The DirectX High Level Shading Language (HLSL)

By calling the `GetParameters()` method of the `IProgram`-instance, the user retrieves a list of all uniform input variables of the program. Using these variables the values of the uniform inputs can be set. When declaring the uniform inputs in the source code of the shader program, the developer can also use predefined names for the inputs. The values of the inputs are then automatically set by YARE and need no further user attention. These predefined names are as follows:

- `"WorldViewMatrix"`: The combined world view matrix. This is the matrix describing the transformation from the object- to the view space. This matrix is also called *Modelview-Matrix* in OpenGL.
- `"WorldViewMatrixI"`: The inverted combined world view matrix.
- `"WorldViewMatrixT"`: The transposed world view matrix.
- `"WorldViewMatrixIT"`: The transposed inverted combined world view matrix.
- `"ViewProjectionMatrix"`: The combined view projection matrix.
- `"WorldViewProjectionMatrix"`: The combined world view projection matrix.
- `"WorldMatrixI"`: The inverted world matrix.

- `"WorldMatrixT"`: The transposed world matrix.
- `"WorldMatrixIT"`: The transposed inverted world matrix.
- `"ViewMatrixI"`: The inverted view matrix.
- `"ViewMatrixT"`: The transposed view matrix.
- `"ViewMatrixIT"`: The transposed inverted view matrix.
- `"CamPos"`: The position of the camera in world space.
- `"TimeInSeconds"`: The number of seconds that have elapsed since the operating system was started.
- `"TimeInMilliseconds"`: The number of milliseconds that have elapsed since the operating system was started.
- `"gl_ModelViewMatrix"`: Same as `"WorldViewMatrix"`.
- `"gl_ProjectionMatrix"`: The projection matrix.
- `"gl_ModelViewProjectionMatrix"`: Same as `"WorldViewProjectionMatrix"`.
- `"gl_ModelViewMatrixInverse"`: Same as `"WorldViewMatrixI"`.
- `"gl_ProjectionMatrixInverse"`: The inverted projection matrix.
- `"gl_ModelViewProjectionMatrixInverse"`: The inverted combined world view projection matrix.
- `"gl_ModelViewMatrixTranspose"`: The transposed combined world view matrix.
- `"gl_ProjectionMatrixTranspose"`: The transposed projection matrix.
- `"gl_ModelViewProjectionMatrixTranspose"`: The transposed combined world view projection matrix.
- `"gl_ModelViewMatrixInverseTranspose"`: The transposed inverted combined world view matrix.
- `"gl_ProjectionMatrixInverseTranspose"`: The transposed inverted projection matrix.
- `"gl_ModelViewProjectionMatrixInverseTranspose"`: The transposed inverted combined world view projection matrix.
- All renderstate names, e.g. `"WorldMatrix"`, `"ClearColor"`, `"AlphaTestEnable"` and `"ClipPlane0"`.

Working with Cg interfaces

As with Cg, the uniform inputs can also be interfaces and not only value-based types. Those interface input parameters are retreived by the `GetParameters()` method of the `IProgram`-instance as well. All possible implementations of a Cg interface are returned by that method too.

To connect an implementation of a Cg interface with the interface itself, the implementation variable is set as an element of the interface variable. The following source code snippet shows an example.

```
// Retreive all uniform inputs of the program.
VariableSetPtr uniform_vars = vertex_program->GetParameters();

// Return the variable for the interface input parameter
// named "MaterialInterface" of type "IMaterial".
IVariablePtr interface_var = uniform_vars->GetVariable( "MaterialInterface" );

// Return the variable named "Blinn" implementing the "IMaterial" interface.
IVariablePtr blinn_var = uniform_vars->GetVariable( "Blinn" );

// The implementation structure has some member variables.
// Here the diffuse coefficient is set.
IVariablePtr blinn_diffuse_var = uniform_vars->GetVariable( "Blinn.diffuse" );
blinn_diffuse_var->SetElement(0, Vec3f(0.8f,0.4f,0.1f));

// Connect the Blinn material with the interface input parameter.
interface_var->SetElement(0, blinn_var);
```

3.7.3 Using the Rendering Interface

The following example demonstrates how to setup the device.

```
// Load the OpenGL driver ( = implementation of the rendering interface).
// The driver is loaded from YareGraphicsGL.dll.
Yare::Core::Reflection::LoadPlugin( "YareGraphicsGL" );

// Create the rendering device.
IDevicePtr device = Yare::Graphics::Rendering::CreateDevice();

// Initialize the device.
DeviceInit init_struct;
init_struct.bpp = 32;
init_struct.fullScreen = false;
init_struct.width = 800;
init_struct.height = 600;
init_struct.multiSampling = 4;
init_struct.showFps = true;
init_struct.showFpsConstSize = true;
init_struct.verticalSync = true;
device->Initialize(window, init_struct);
```

The following example demonstrates how to setup a geometry and render it.

```
using namespace Yare::Graphics::Variables::Manipulators;

Sphere3f sphere;
sphere.center = Vec3f(0,0,0); sphere.radius = 1;
Yare::Core::Math::MeshGenerators::Sphere::GetGeometry( sphere,
  10, 10, pos_data, idx_data );

// Create a position variable.
VariableDesc pos_desc = VertexPositionDesc();
pos_desc.additional = GFX_GPU;
IVariablePtr pos_var = CreateVariable<Vec3f>(pos_desc, pos_data);

// Create an indexbuffer.
VariableDesc idx_desc = IndexBufferDesc();
idx_desc.additional = GFX_GPU;
IVariablePtr idx_var = CreateVariable<uint32>(idx_desc, idx_data);

// Create variables indicating primitive type and element count.
VariableDesc type_desc = PrimitiveTypeDesc();
type_desc.additional = GFX_MEMORY;
std::vector< uint16 > type_data;
type_data.push_back( TriangleListType() );
IVariablePtr type_var = CreateVariable<uint16>(type_desc, type_data);

VariableDesc count_desc = ElementCountDesc();
count_desc.additional = GFX_MEMORY;
std::vector< uint32 > count_data;
count_data.push_back( idx_var->GetDesc().count );
IVariablePtr count_var = CreateVariable<uint32>(count_desc, count_data);

// Collect the variables.
VariableSetPtr variableset = VariableSetPtr( new VariableSet() );
variableset->Add(pos_var, false);
variableset->Add(idx_var, false);
variableset->Add(type_var, false);
variableset->Add(count_var, false);

// Create texture coordinates.
TextureMapper::CreateSphericalTextureMapping(variableset,
  RESULT_STORAGE_INDEX0, GFX_GPU);
tex_var = variableset->GetVariable( VertexTexCoord2DDesc().name );

// Create geometry from variables.
IGeometryPtr geo = device->CreateGeometry( false );
```

```
geo->Add( variableset );

// Rendering loop:
device->Begin();
device->GetTargetManager()->SetSingleTarget( device->GetFramebuffer() );
device->GetTargetManager()->Done();

geo->Draw();

device->End();
device->Present();
```

The example below demonstrates how to setup a texture and use it for rendering.

```
// Load image from file.
ISurfacePtr image = ISurfacePtr( new Bitmap("sample.png") );
std::vector<ISurfacePtr> images;
images.push_back(image);

// Create texture from image.
ITexturePtr texture = device->CreateTexture(TYPE_INT,
  1.0f, CAP_MIPMAPPING, image->GetWidth(), image->GetHeight(), SAMPLER_2D);
texture->Upload(0, images);

// Create a sampler for the texture.
ISamplerPtr sampler = device->CreateSampler();
sampler->SetTexture(texture);
sampler->SetFilter(FILTER_BILINEAR);

// Rendering loop:
...

// Set transformation for the geometry.
Matrix4x4f world_matrix(1);
world_matrix.SetT(0,0,-100);
device->GetState()->Set( States::cWorldMatrix, world_matrix );

sampler->Bind(texture, 0);
geo->Draw();
sampler->UnBind(texture, 0);
...
```

The following example demonstrates how to use a vertex- and fragmentprogram for rendering.

```
std::string vp_source = Yare::Core::Io::ReadTextFile("simple_vp.cg");
```

```
std::string fp_source = Yare::Core::Io::ReadTextFile("simple_fp.cg");
ICompilerPtr compiler = device->GetCompiler();

// Compile the programs.
IProgramPtr vertex_program = compiler->Build(vp_source, "",
  Backends::Cg(), PRG_VERTEX);
IProgramPtr fragment_program = compiler->Build(fp_source, "",
  Backends::Cg(), PRG_FRAGMENT);

// Set value for uniform parameter of fragmentprogram.
VariableSetPtr uniform_vars = fragment_program->GetParameters();
IVariablePtr specular_var =
  uniform_vars->GetVariable( "SpecularCoefficient" );
specular_var->SetElement(0, 64.0f);

// Bind the variables to the appropriate vertexprogram inputs.
device->GetBindingManager()->AddBinding(pos_var, vertex_program, "Position");
device->GetBindingManager()->AddBinding(tex_var, vertex_program, "TexCoords");

// Rendering loop:
...

// Get the number of passes of the programs.
uint16 vp_passes = vertex_program->GetPasses();
uint16 fp_passes = fragment_program->GetPasses();

for (uint16 vp_pass = 0; vp_pass < vp_passes; ++vp_pass)
{
  // Begin pass of vertex program.
  vertex_program->Begin(vp_pass);

  for (uint16 fp_pass = 0; fp_pass < fp_passes; fp_pass++)
  {
    // Begin pass of fragment program.
    fragment_program->Begin(fp_pass);
    geo->Draw();
    fragment_program->End();
  }
  vertex_program->End();
}
...
```

3.7.4 The Interfaces

See Section A.2 for the main C++ interfaces of this chapter.

3.8 The Effect Framework

This chapter can be considered as the main part of this thesis because it describes a new approach on how to integrate complex rendering effects seamlessly into a complete rendering engine. Solving this issue was the reason for writing this thesis at all. Implementing a demo application showing a rendering effect can be simple, however, a lot of research effort is made lately to enable a rendering engine handling all different kinds of effects without breaking its main design. See Section 2.3 for different approaches on this topic.

The effect framework requires as important feature the equal treatment of effects. This means e.g. that from the implementation point of view the shadow mapping effect and a diffuse lighting effect must be treated equally in the source code.

Defining which lights and objects *cast* shadows and which objects *receive* shadows should not be decided by the lights and objects themselves but simply by the configuration of the scenegraph. This is because storing a shadow flag at the lights and meshes would need an introduction of special methods in the source code classes, which in turn would break the feature request mentioned in the paragraph above.

The requirements of the framework are:

- All standard effects should be implemented, e.g. different materials, different light types, shadow mapping, texture mapping and normal mapping.
- Any number of such effects should be combinable by the user. The system should then generate necessary passes and vertex- or fragment programs automatically.
- Fast and easy integration of new effects including new Cg interfaces/implementations and additional render passes.
- Post-Processing framework.
- Some kind of level-of-detail concept for effects.

3.8.1 Overview

This section gives an overview of the effect framework as a short summary of the following sections. This should help to easier understand the following detailed explanations of the concepts of the effect framework.

Using the effect framework, the user of the system (e.g. a 3D artist) can use predefined small basic effects (e.g. diffuse lighting, shadow mapping and texture mapping) to easily build larger and more complicated rendering effects. For this, the user does not have to have knowledge about the underlying shader source code, multipassing or renderstates. These issues are automatically handled by the effect framework. A list of available basic effects can be found in Subsection 3.8.10. How to add additional basic effects is explained in Subsection 3.8.11.

Each basic *rendering effect* consists of one or more *techniques* implementing the desired effect. One technique is chosen at runtime depending on the available hardware and level of detail. Such a rendering technique consists of one or more *technique parts* which represent the smallest task unit to achieve the rendering effect. Technique parts have inputs they require and outputs they produce. These inputs can take their value from a variable (as described in Section 3.6), the output of another technique part or can use a predefined default value. More details on the internal structure of rendering effects can be found in Subsection 3.8.2.

Technique parts can run their code (C++ source code or shader code) on different resources as the CPU, the GPU vertex processing unit or the GPU fragment processing unit. If a technique part wants to use shader code, it implements one of the *predefined Cg-interfaces* as described in Subsection 3.8.3. An example for a technique part *not* using the GPU is the generation of shadow volumes on the CPU.

At runtime, a list of technique parts per geometry is collected and converted into a list of *rendering passes*. Each rendering pass can have its own rendering target (like the framebuffer or a dynamic texture). If one or more technique parts of a rendering pass want to execute shader code on the GPU, **Cg framework programs** are automatically generated to execute vertex- and fragment shader code. These framework programs call the list of implementations of the Cg-interfaces of the technique parts. More details on this topic can be found in Subsection 3.8.3.

To find the correct order for all rendering passes in a scene, the state dependencies and state changes of the passes are inspected. A state dependency graph (as explained in detail in Subsection 3.8.6) is built out of this information. The correct rendering order of the passes can be found by traversing the dependency graph bottom-up.

A graphical representation of this overview text can be found in Figure 3.10 and Figure 3.11.

3.8.2 Structure of Effects

Every effect is structured in the following way: An effect (`IEffect`) consists of one or more techniques (`ITechnique`). Every single technique completely implements the desired effect for itself in a special way, however, only one of them is chosen at runtime to be used for rendering. This decision is based on which technique is supported on the current graphics hardware. Additionally, a Level-of-Detail (LOD) range can be specified for each technique, which is then used to select the proper technique for the current distance between camera and object. The details on the implemented LOD concept can be found in Subsection 3.8.7.

Since techniques can be complex algorithms and constructs, they are split into one or more parts (`ITechniquePart`). Each technique part handles a small subset of tasks to achieve the desired effect. Examples for such tasks are:

- Calculate shadow volume for a given geometry and light.
- Transform the camera direction into the tangent space of a vertex.
- Calculate the texture coordinate offset for relief- or parallaxmapping.
- Iterate over all lights and sum up their contribution to the fragment shading.
- Detect if a fragment is in shadow with respect to a given light.

Some of those tasks are solved using the CPU, but others are implemented by using a vertex- or fragment program running on the GPU. As it must be possible to process the tasks above within one rendering call, the technique parts which require the GPU must be *connected* together to expand to only one vertex- and fragment program. Such parts are derivations from the C++ class `PipelinePartTechniquePart`. The name arises from the fact that each such technique part implements a part from the graphics pipeline. The idea of the effect framework is that such pipeline parts should be combinable automatically. The user selects which effects to use and the effect framework converts the containing `PipelinePartTechniqueParts` to a renderable vertex- and fragment program.

Technique parts have inputs which change the parts' behavior and may have outputs calculated by the parts. The corresponding C++ interfaces are `IInput` and `IOutput`. Inputs can be categorized according to their variation: they can be on a per-vertex basis (also called *varying* inputs or vertex attributes) meaning that they can vary per vertex, or they can be equal for the whole rendered geometry (also called *uniform* inputs). Outputs are always calculated on a per vertex basis. Some inputs also have useful default values, thus, the user is not required to set all inputs in the first place. Only uniform inputs can have default values.

Examples for inputs of technique parts:

- The vertex position in object space (varying input).
- The texture coordinate (varying input).
- The current view matrix (uniform input).
- The light direction and intensity (uniform input).

Examples for outputs calculated by technique parts:

- The normal vector in tangent space.
- The camera direction in object space.
- The texture coordinate offset for the first texture stage.

An important feature of effects is that they can change the culling volume of objects which are rendered with that effect. This is needed if an object A is not inside the viewing frustum of the scene camera but is visible due to an effect of another object B which makes object A to appear in the final image. An example for such a constellation is a scene containing a mirror as shown in Figure 3.9. In such a setup objects outside

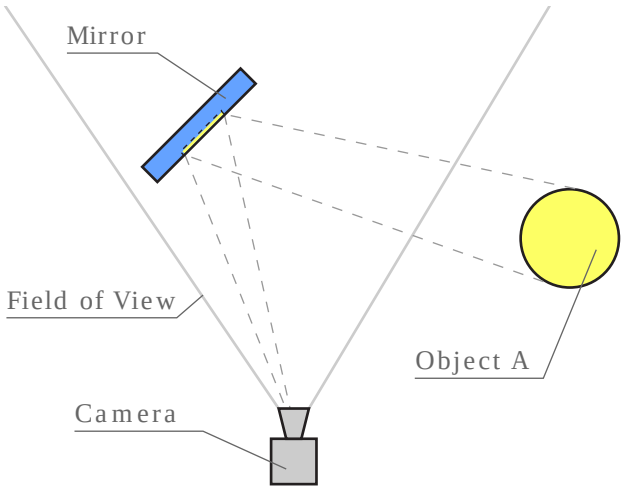

Figure 3.9: A scene containing a mirror that makes it necessary that the rendering effect of the mirror changes the bounding volume used for culling of object A. Otherwise object A would not be sent to the graphics device.

the viewing frustum but reflected by the mirror still need to be drawn. More examples of effects changing the culling volume can be found in Subsection 3.8.10.

The UML diagram of the effect classes can be found in Appendix B. An overview of the main components which take part in the effect framework can be found in Figure 3.10. The runtime behaviour of the system is outlined in Figure 3.11.

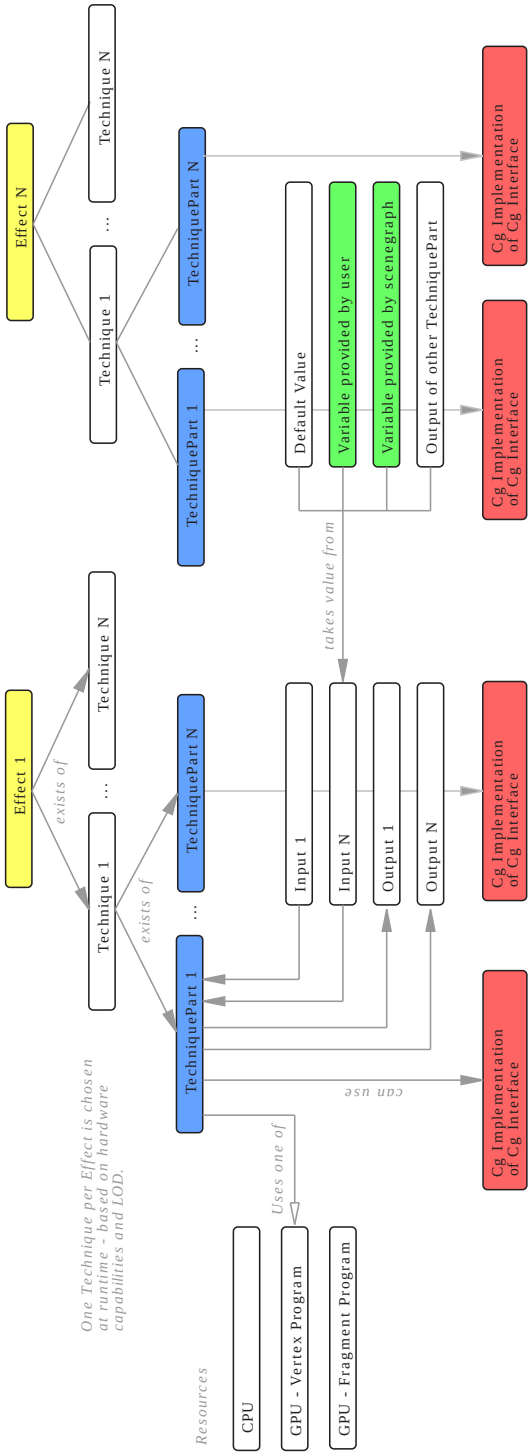

Figure 3.10: The layout of the main components of the effect framework

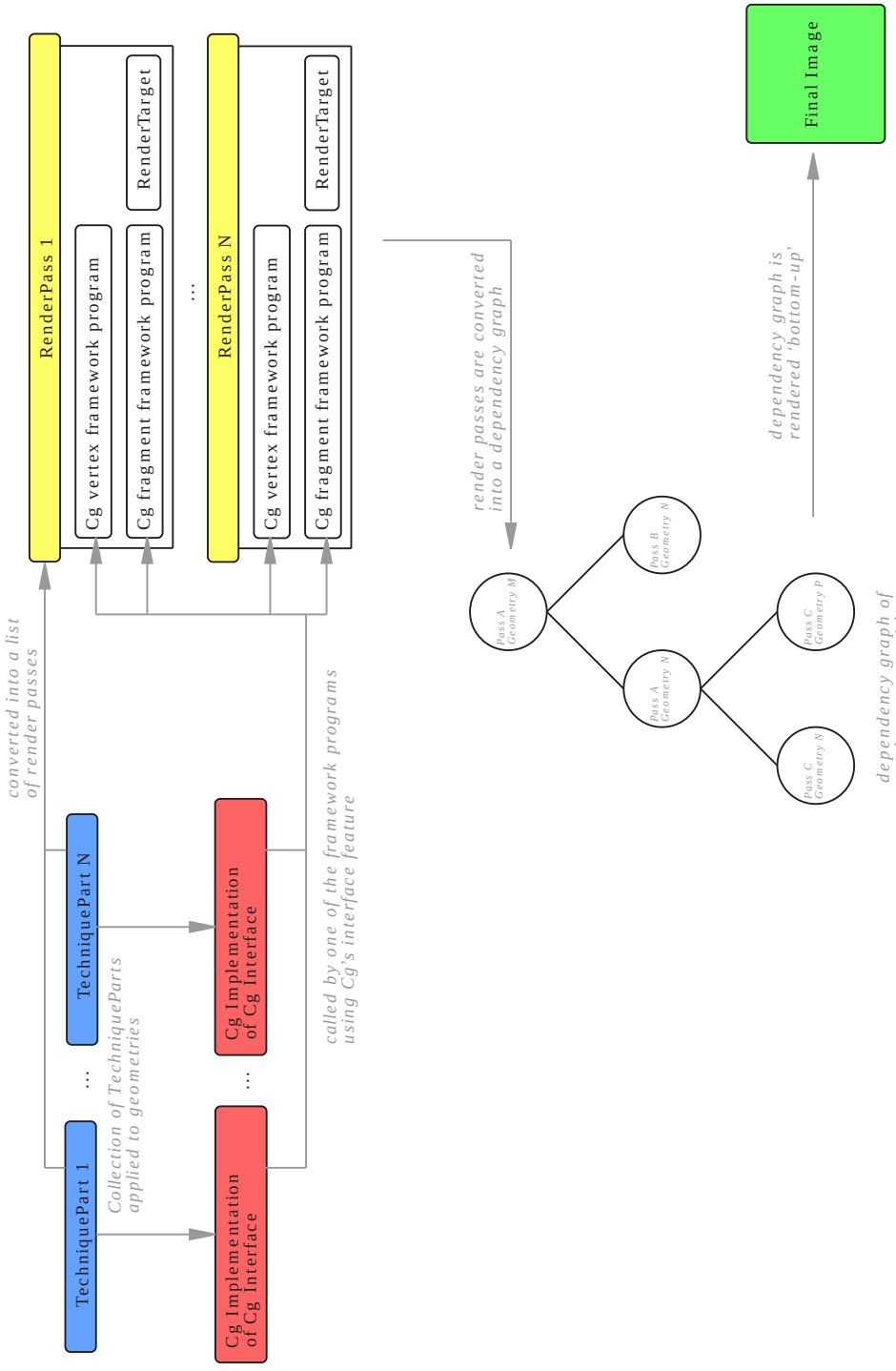

Figure 3.11: Outline of the runtime behaviour of the effect framework

3.8.3 Generating the Vertex- and Fragment Framework Programs

The task of this chapter is to describe the conversion of a list of `PipelinePartTechniqueParts` into one or more vertex- and fragmentprograms. The implementation of this feature makes heavy use of Cg and its *interface*-feature (see Subsection 2.2.3 for details). The **center of the algorithm** is made up by a Cg vertex- and fragment program (*framework programs*) with the only task to call arrays of special Cg-interfaces. Therefore, if a `PipelinePartTechniquePart`-implementation wants to put its Cg-code into the framework program, it has to implement one of the given Cg-interfaces and connect its implementation to the framework program. See Subsection 3.7.2 (*Working with Cg interfaces*) on how to connect implementations of Cg-interfaces to vertex- or fragment programs. See Table 3.6 for a complete list of available Cg-interfaces.

The reasons and also requirements for choosing Cg as the shading language for the effect framework were:

- Cg is the only high-level shading language which is supported by OpenGL and DirectX.
- Cg is supported on nVidia and ATI graphics hardware.
- Cg is supported on Windows, Linux, MacOS and Solaris.
- Cg supports interfaces to types, which is a major requirement for the described effect framework.

These were important requirements as the effect framework should not be limited to any soft- or hardware platform.

An overview on how the Cg framework program, the Cg interfaces and the Cg implementations work together can be found in Figure 3.12.

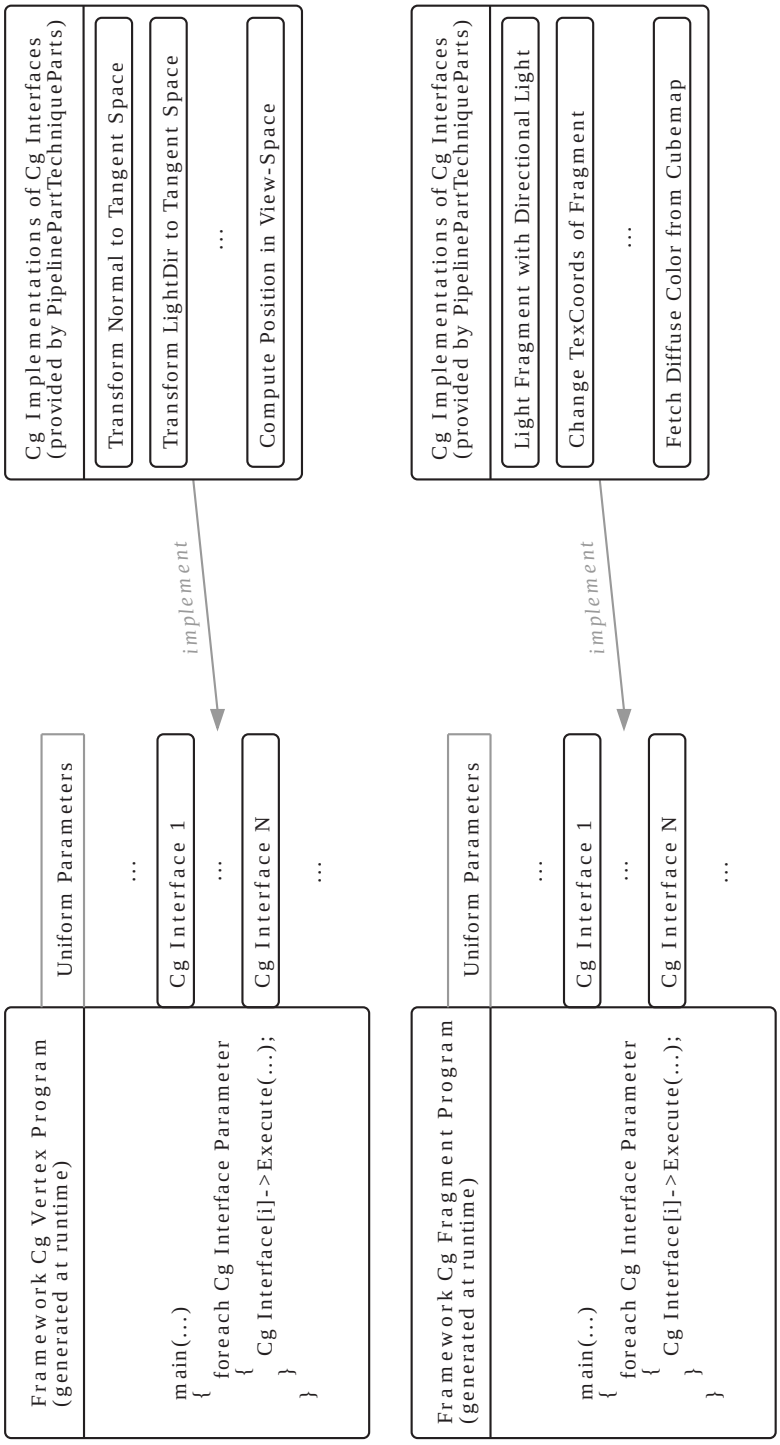

Figure 3.12: Usage of Cg in the effect framework

Parameter Passing in the Framework Program

The uniform parameters of the *vertex* framework program are the list of vertex Cg-interfaces and the list of needed transformation matrices. The uniform parameters of the *fragment* framework program are the list of fragment Cg-interfaces and the structure definition of the data of one fragment (as described in the next paragraph *Handling of Varying Program Inputs and Outputs*). The name of the Cg-interface parameters for the Cg source code is generated by taking the name of the Cg-interface and postfixing it with the predefined constant `Parameter`.

To call a single Cg-interface from the framework program, the preconfigured calling code of this interface is simply placed into the source code. This calling code is defined in the XML configuration file of that interface as described in the paragraph *Configuration and Extension of Cg-interfaces* of this subsection. The calling code also contains the parameters for this interface method call. For vertex Cg-interfaces the available parameter names are:

- `vertexInput`: The structure definition of the varying vertex inputs.
- `fragmentInput`: The structure definition of the varying fragment inputs.

For fragment Cg-interfaces the available parameter names are:

- `fragmentInput`: The structure definition of the varying fragment inputs.
- {Cg-interface name}`Parameter`: The list of Cg-interfaces of the named type {Cg-interface name}. For example, to get the list of all *ILight* Cg-interfaces as input to this configured interface, simply put the string `ILightParameter` to the calling code. By this concept, it is possible to let Cg-interface implementations call other Cg-interfaces.
- `ambientColor`: The current ambient color of the fragment.
- `diffuseColor`: The current diffuse color of the fragment.
- `specularColor`: The current specular color of the fragment.

Since all parameters are passed by reference, the implementations of the Cg-interfaces can change the value of the parameters. Using this possibility, one implementation can propagate values to other implementations.

Another way to communicate with other implementations is by the means of *global variables* placed into the Cg source code of the implementations. However, using this technique, it is not guaranteed that a variable naming conflict is avoided.

The following example shows the configuration and usage of the `IMaterial` Cg-interface. This interface takes the current fragment properties, all *lights* and *normal transformers* as input to calculate the ambient, diffuse and specular color.

The Cg-interface source code:

```
#ifndef _IMATERIAL_CG_
#define _IMATERIAL_CG_

#include "../Interfaces/ILight.cg"
#include "../Interfaces/INormalTransformer.cg"

interface IMaterial
{
  float3 Evaluate(FragmentInput fragment,
                  ILight lights[],
                  INormalTransformer normalTransformers[],
                  inout float3 ambientColor, inout float3 diffuseColor,
                  inout float3 specularColor);
};

#endif // _IMATERIAL_CG_
```

The XML configuration of this interface:

```
<FragmentProgramInterface>
  <IndependentLoop>True</IndependentLoop>
  <Call>
    Evaluate(fragmentInput,
             ILightParameter, INormalTransformerParameter,
             ambientColor, diffuseColor, specularColor)
  </Call>
</FragmentProgramInterface>
```

A source snippet of the generated framework program when using this interface:

```
for (int i = 0; i < IMaterialParameter.length; i++)
{
  IMaterialParameter[i].Evaluate(fragmentInput,
                                 ILightParameter, INormalTransformerParameter,
                                 ambientColor, diffuseColor, specularColor);
}
```

Handling of Varying Program Inputs and Outputs

Every `PipelinePartTechniquePart`-implementation must define which vertex attributes it requires as inputs and which attributes it generates as outputs. This is done in the C++ source code of the implementation class. Examples for such inputs and outputs are:

- Position and normal in object space
- Position and normal in view space
- Tangent and binormal in object space
- Texture coordinates

Using the information of inputs and outputs (as described above) an algorithm can be developed which creates a list of

- the inputs the generated Cg vertex program expects,
- the outputs the generated Cg vertex program generates,
- the inputs the generated Cg fragment program expects,
- vertex attributes which must be passed through the generated Cg vertex program to get to the fragment program (since no connected `PipelinePartTechniquePart`-implementation generates them).

These lists are required to generate the varying inputs and outputs for the framework Cg-programs as illustrated in Figure 3.13.

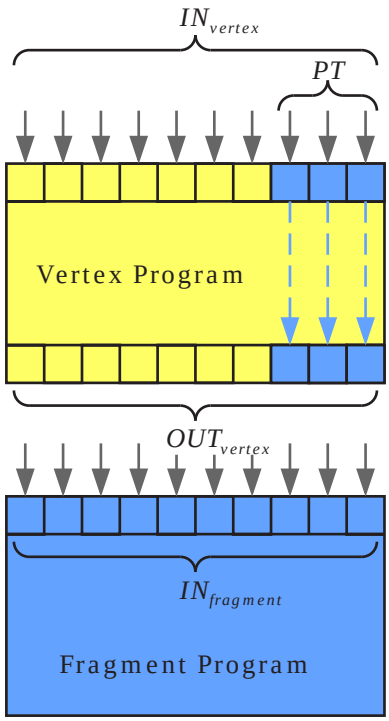

Figure 3.13: Illustration of the inputs and outputs of vertex and fragment programs. Some vertexprogram inputs (PT) are just passed through to the inputs of the fragment program.

The solution to this task is the following formalization:

- Let IN_V^i be the i-th input of the connected vertex Cg-interfaces.
- Let N_V^{IN} be the total number of inputs of the connected vertex Cg-interfaces.
- Let OUT_V^i be the i-th output of the connected vertex Cg-interfaces.
- Let N_V^{OUT} be the total number of outputs of the connected vertex Cg-interfaces.
- Let IN_F^i be the i-th input of the connected fragment Cg-interfaces.
- Let N_F^{IN} be the total number of inputs of the connected fragment Cg-interfaces.

A *connected Cg-interface* is a Cg-interface which is included into the Cg framework program.

Then the list $IN_{fragment}$ of all inputs of the fragment program is defined as in Equation 3.1.

$$IN_{fragment} = \bigcup_{i=1}^{N_F^{IN}} IN_F^i \tag{3.1}$$

The list OUT_{vertex} of all outputs of the vertex program is defined as in Equation 3.2.

$$OUT_{vertex} = IN_{fragment} \tag{3.2}$$

The list PT of all attributes passed through the vertex program directly to the fragment program is defined as in Equation 3.3.

$$PT = \{i \in IN_{fragment} \mid i \notin \bigcup_{k=1}^{N_V^{OUT}} OUT_V^k\} \tag{3.3}$$

The list IN_{vertex} of all inputs of the vertex program is defined as in Equation 3.4.

$$IN_{vertex} = \bigcup_{i=1}^{N_V^{IN}} IN_V^i \cup PT \tag{3.4}$$

Configuration and Extension of Cg-interfaces

This section gives an overview on how to integrate new functionality into the effect framework by adding additional Cg-interfaces.

To make a new Cg-interface available to a `PipelinePartTechniquePart`-implementation, it is necessary to add two textfiles to the interface data directory (`Data\Effects\Interfaces`). The first textfile is simply the Cg source file declaring the interface. The second file is an XML configuration file containing the following information about the Cg-interface:

- Is the interface intended for a vertex- or a fragment program
- Is it called directly by the framework program or only by another interface implementation
- The calling code including the return variable, the interface method and the parameters
- For vertexprogram interfaces the coordinate space is defined (object-, world-, view- or projectionspace)

To add a new implementation for a given Cg-interface, it is necessary to simply put the Cg source code file into the implementation data directory (`Data\Effects\Implementations`). The `PipelinePartTechniquePart`-implementation can choose to implement a Cg-interface by simply defining its name and the name of the Cg-implementation of that interface. A list of all available interfaces can be found in Table 3.6.

To specify in which order the Cg-interfaces are called by the framework programs, two XML configuration files exist in the config-directory of the effect framework (`Data\Effects\Config`). They contain the calling order for the vertex- and fragment Cg-interfaces.

3.8.4 The State Table

The state table is a global entity used by effects to store the states of their resources and exchange common resource data. Examples of resources are any kind of rendertargets and other global data, which is needed by many effects.

The state table can also be used to enforce a specific order between effects, e.g. the effect that fills the depthmap with the depth values of the scene must be rendered before the effect using this depthmap. Therefore, the state table is also an important instrument for managing global multipassing as described in Subsection 3.8.6.

The state table consists of a set of resource slots, where every resource slot has

- a unique name which is the name of the resource.
- a lock-flag indicating whether the resource is locked or not.
- a ready-flag indicating whether the resource is ready or not.
- a value of any type.

To simplify the usage of the state table two classes are introduced. The first class is called `StateChange` whose instances can change the lock-flag, the ready-flag and the value of a resourceslot. The second class is called `StateDependency` whose instances describe a dependency on the state of a resource slot. Thereby this dependency is only fulfilled if the state of the defined rosource slot matches the defined state of the `StateDependency`.

Usage examples of the state table with its resource slots,
`StateDependencies` and `StateChanges` can be found in Subsection 3.8.10.

3.8.5 Rendering the Effects

This section describes how the generated vertex- and fragment program from the previous chapter are transparently embedded into the effect framework.

The most important interface in this context is `IPass` which is the interface to all render passes and which is implemented by the following classes:

- `DefaultPass`: This pass renders to the framebuffer and has no special behavior.
- `AdditionalPass`: This pass class also implements the `ITechniquePart`-interface and therefore can be used as a part of a technique. Usually, this instance has its own rendertarget.
- `PipelinePass`: Internally, this pass has its own rendering management - as the passes of a CgFX-effect have. This is also the intended usage of this class. So they are not directly instanced by the user but are provided by `CgFXEffects`. This class is also derived from `AdditionalPass`.
- `PostProcessingPass`: This pass requires the filled framebuffer and depthbuffer as input and renders to the framebuffer or a separate rendertarget. This class is also derived from `AdditionalPass`.

Another important class is the rendertarget stack. Rendertargets can be pushed on a stack and are activated by this class. When popping rendertargets from the top of the stack, the underlying rendertargets are activated again. This class (`RenderTargetStack`) is needed when it comes to multipassed rendering with different rendertargets for each pass.

The following paragraph gives a complete step-by-step introduction on the internally performed steps for retrieving the final image in the framebuffer from a list of user-selected effects.

1. Depending on the current graphics hardware and the current level-of-detail of the object, a list of techniques is chosen for implementing the desired effects.
2. A list of technique parts is retrieved from the technique list.
3. This list is split into two groups: the `AdditionalPasses` and the `PipelinePartTechniqueParts`.

4. The list of `PipelinePartTechniqueParts` is handed over to the `PartConnector` which builds groups of technique parts, where every group will be connected together into a single vertex- or fragmentprogram. If the user whishes to put technique parts into different groups, he must assign different group indices.
5. Every technique part group is assigned to an instance of the `AdditionalPasses` and to an instance of the `DefaultPass`. By assigning technique parts to a render pass, the pass adopts all inputs, outputs and state dependencies of the technique parts.

At this point a list of renderable passes has been generated and is waiting to be executed by calling the `Render()`-method.

The order in which the passes are executed is crucial. See Subsection 3.8.6 for details on this topic.

1. When calling the `Render()`-method of an `IPass`, the assigned technique parts are converted to a vertex- and fragmentprogram.
2. If any vertex attributes of the geometries passed to the `Render()`-method are missing (but are required by the vertex program), they are generated automatically if possible. See Subsection 3.6.4 for details on how to create missing vertex attributes.
3. If the pass has its own rendertarget assigned, it is pushed on the `RenderTargetStack`.
4. Renderstate variables of the variables passed to the `Render()`-method are *executed*. This means that their encapsulated renderstate is applied to the graphics device.
5. The values of the variables passed to the `Render()`-method are copied to the inputs of the pass.
6. All pass inputs are assigned to the generated vertex- and fragmentprogram.
7. The geometries passed to the `Render()`-method are drawn.
8. Assigned renderstates are deactivated.
9. If the pass has a special rendertarget assigned, it is popped from the `RenderTargetStack`.
10. The `StateChanges` of the assigned technique parts are executed.

The effect framework still supports rendering using the fixed function pipeline of the graphics device. This is accomplished by assigning no `PipelinePartTechniqueParts` to an `IPass`-implementation. So no vertex- or fragmentprogram will be generated but the other steps remain the same.

3.8.6 Handling of Multipassing

This section gives information about how management of local and global multipassing is implemented in YARE.

The effect framework makes no difference between local- and global multipassing as described in Section 2.3. Each render pass is treated the same way independent of how many objects are rendered with a pass. This is possible since it is not the task of the effect framework to decide which objects are rendered with which passes. This is the task of the user by correctly configuring the scenegraph (as can seen in Section 3.10). The system detects the list of passes a single object is rendered with. Afterwards, the effect framework automatically finds the correct order of when to render an object with which pass as described in the next paragraphs. Therefore, it makes no difference to the effect framework if a render pass is applied to all objects in the scene or just to a single one.

After the list of renderable passes has been generated (as described in the previous chapter), this list is handed over to the `PassSorter`. Its task is to find the correct order for the render passes. A correct order is achieved when the `StateDependencies` of all passes are fulfilled.

The only important information about a pass at this point are its `StateDependencies` and `StateChanges` as described in Subsection 3.8.4. Examples of state dependencies and state changes can be found at the description of the *Render to Cubemap*-, *Dynamic Cube Mapping*-, *Render to Depth Texture*- and the *Spot Light Shadow Mapping* effect as listed in Subsection 3.8.10.

Using the state dependencies and state changes a dependency-graph can be built where every node represents a group of passes with equal state dependencies. The pseudo code of the implemented algorithm to build this dependency-graph can be found in Algorithm 3.1. All pass groups which fulfill the dependencies of a node are inserted as child nodes. A pass group A fulfills the dependencies of another pass group B if one or more state changes of pass group A will change the state table in a way that one or more state dependencies of pass group B are fulfilled. In this way a pass group can only be rendered as soon as all its children are rendered. If no other pass group can completely fulfill the dependencies of a given group, the passes in this group cannot be rendered. Another problem arises if cycles occur in the graph, meaning there is no chance to find a pass group in the cycle with which rendering can be started.

The algorithm traverses bottom-up and collects all pass groups which have no children and removes the collected nodes from the parents. This is done until all pass groups have been collected.

The order of the passes in a single pass group is not crucial since they all have the same state dependencies. This fact is used to sort the passes of a group by vertex- and

```
groups ⇐ BuildGroupsWithEqualStateDependency();
for i = 0 to groups.Size − 1 do
  for j = 0 to groups.Size − 1 and i <> j do
    if StateChanges(groups[j]).FulFill(StateDependencies(groups[i])) > 0 then
      groups[i].AddChild(groups[j]);
    end if
  end for
end for
```

Algorithm 3.1: Pseudocode for building the dependency-graph

fragment program, then by texture and finally front to back by the distance between the center of the bounding volume and the camera position.

The following example illustrates this algorithm working on 5 passes A - F. Pass A and C have equal state dependencies and their state changes can fulfill the dependencies of pass F. Pass B and D can fulfill the state dependencies of pass A and C. The generated dependency graph is illustrated in Figure 3.14. Following the algorithm above the leaf nodes are rendered first which gives the following pass order: D - B - A - C - F.

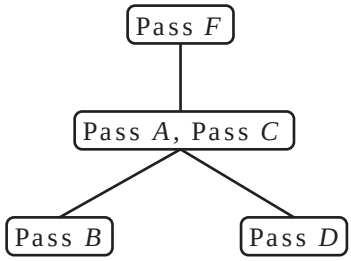

Figure 3.14: A dependency graph example

3.8.7 Effect Level of Detail

If objects are far away from the camera, it is not necessary to render them with every little detail since these details cannot be seen at all. Simplifying their appearance can e.g. be achieved by reducing the faces of the geometry. Another option to achieve better performance is to simplify the algorithm used to render the geometry.

The effect framework of YARE allows specifying a minimum and maximum distance for a technique implementing an effect. Taking the current distance between the camera position and the world position of the rendered object, the effect framework automatically chooses the technique predetermined for this value. The example presented in Figure 3.15 shows a teapot with an effect consisting of three techniques applied. The

technique used for the nearest rendering implements *relief mapping* as presented in Subsection 3.8.10. The next two levels of detail implement *parallax mapping* and *normal mapping*.

Figure 3.15: Example showing the effect LOD in use. From left to right: *relief mapping*, *parallax mapping* and *normal mapping*.

If more than one technique is available for a given distance, the algorithm chooses the technique which will also be used for further rendering calls. For this wanted behavior the algorithm watches the variations of the distance for being able to make assumptions about which technique will be selected in the future. This is achieved by assuming that the distance variation stays constant and then minimizing rendering technique switches when walking along the distance axis. This selection algorithm helps minimizing switches of rendering techniques which can be an expensive operation if different vertex- and fragmentprograms must be loaded or activated on the graphics device. The behavior of the implemented algorithm is presented in Figure 3.16.

The main disadvantage of this algorithm is the assumption that the distance variation stays constant. This can lead to many rendering technique switches if the sign of the distance variation changes in a band where more than one rendering technique is valid. On the other hand, frequent changes of the sign of the distance variation are rare in practice because this would imply moving the rendering camera alternating forth and back.

3.8.8 Performance Optimizations

Optimization for the generated vertex framework program

Transforming the vertex position from one reference coordinate system to another requires an expensive vector-matrix multiplication which sould be avoided if possible. This means if no Cg-interface for a given coordinate system is needed, it is not necessary to transform the position to that coordinate space. This is the reason why the XML configuration file of the Cg-interface must contain the coordinate space of the interface. The following example should clarify this optimization: If only a view and projection space Cg-interface is connected to the framework program, it is not necessary to transform

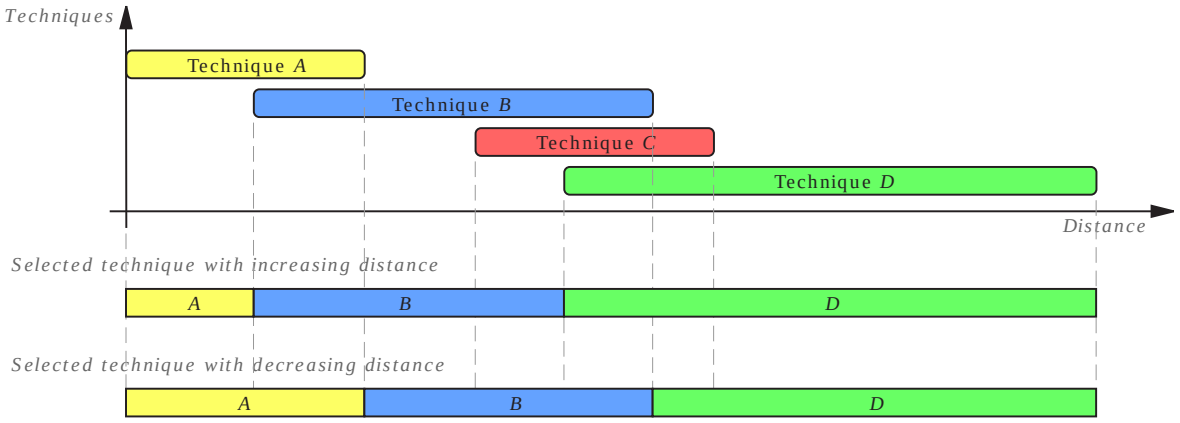

Figure 3.16: An example of an effect having four techniques with different minimum and maximum *LOD distances*. The selected techniques are illustrated below for increasing and decreasing distance between camera and object. Technique C is never selected to minimize render technique switches.

the position to the world space. Therefore, the object position given as variable program input is directly transformed to the view space and given the Cg-interfaces of this coordinate space.

Filtering the technique parts

Some passes only require a simplified version of a technique part to fulfill its purpose.

A pass producing a depthmap of a scene for example, does not require texturing, normal mapping or lighting effects to generate the depthmap. In fact, such effects would slow down rendering and therefore should be avoided. This can be done by asking the technique parts itself for a special version of them fitting the purpose of a render pass. This can be achieved by calling the `GetFilteredVersion`-method of the `ITechniquePart`-interface. This method takes a filter as parameter describing the part version a pass needs. If e.g. the filter is set to `TPF_DEPTH_CHANGING` technique parts that do not change the depth value of a fragment will return `NULL`. Others can return a reference to themselves if they change the depth value.
Filtering out unwanted technique parts is done when applying the list of technique parts to the render passes which is described in Subsection 3.8.5.

Caching

An important performance optimization is caching of results. In the context of the effect framework, the following calculation results are cached:

- The conversion from technique parts to render passes

- The generation of vertex- and fragment programs from technique parts
- The conversion from a list of technique parts into a group of technique part lists

3.8.9 Post-Processing Effect Framework

The YARE effect framework provides a singleton manager for post-processing effects like tonemapping [TumblinRushmeier93], blooming [SpencerEtAl95] and similar techniques which operate on the content of the framebuffer.

Since current graphics hardware is not designed to enable fast processing with read-back of the content of the framebuffer, it is necessary to render the scene directly into a texture instead. Since some post-processing effects also require the z-buffer, the depth values of the fragment are redirected into another texture as well.

Thus, all render passes which would render into the framebuffer must redirect their output into the mentioned textures. This can easily be achieved by pushing the rendertargets onto the rendertarget stack (as described in Subsection 3.8.5) before starting processing of the render passes of the scene. When all render passes have finished and the rendertargets of the post-processing manager contain the image of the scene (as would normally appear in the framebuffer) the post-processing passes are performed on these rendertargets. Since only the last post-processing pass renders into the framebuffer, the passes before must use a different rendertarget as output, which in turn is then used as input for the next pass. The pseudo code of the implemented algorithm can be found in Algorithm 3.2.

3.8.10 Implemented Effects

This chapter describes all implemented effects delivered with the standard version of YARE. Each paragraph gives an overview and details on special behavior of the quoted effect.

Ambient Material

The *ambient material* effect implements ambient lighting using a fragment `PipelinePartTechniquePart`. This technique part implements the `IMaterial` Cg-interface and adds an ambient term for any comprised light on each fragment.

Inputs of the technique part:

- The ambient coefficients of the material as a float triple. Default value is (0,0,0).

Diffuse Material

The *diffuse material* effect implements per-pixel diffuse lighting using a vertex and a fragment
`PipelinePartTechniquePart`. The vertex part implements the `IViewSpaceTransformer` Cg-interface and transforms the vertex position and normal from object- to view space. The fragment part implements the `IMaterial` Cg-interface and evaluates the diffuse lighting equation for any comprised light on each fragment.

Inputs of the vertex part:

- The transposed and inverted world-view-matrix
- The position vectors in object space
- The normal vectors in object space

Outputs of the vertex part:

- The position vectors in view space
- The normal vectors in view space

Inputs of the fragment part:

- The diffuse coefficients of the material as a float triple. Default value is (1,1,1).
- The position vectors in view space
- The normal vectors in view space

Specular Material

The *specular material* effect implements per-pixel specular lighting using a vertex and a fragment
`PipelinePartTechniquePart`. The vertex part implements the `IViewSpaceTransformer` Cg-interface and transforms the vertex position and normal from object- to view space. The fragment part implements the `IMaterial` Cg-interface and adds the specular light contribution for any comprised light on each fragment.

Inputs of the vertex part:

- The transposed and inverted world-view-matrix
- The position vectors in object space
- The normal vectors in object space

Outputs of the vertex part:

- The position vectors in view space
- The normal vectors in view space

Inputs of the fragment part:

- The specular coefficients of the material as a float triple. Default value is (1,1,1).
- The specular exponent of the material as a float. Default value is 32.
- The position vectors in view space
- The normal vectors in view space

Blinn Material

The *blinn material* effect is a combination of the ambient, diffuse and specular material descibed in the previous sections. It is also implemented using a vertex and a fragment `PipelinePartTechniquePart`. The vertex part implements the `IViewSpaceTransformer` Cg-interface and transforms the vertex position and normal from object- to view space. The fragment part implements the `IMaterial` Cg-interface and evaluates the blinn lighting equation for any comprised light on each fragment.

Inputs of the vertex part:

- The transposed and inverted world-view-matrix
- The position vectors in object space
- The normal vectors in object space

Outputs of the vertex part:

- The position vectors in view space
- The normal vectors in view space

Inputs of the fragment part:

- The ambient coefficients of the material as a float triple. Default value is (0,0,0).
- The diffuse coefficients of the material as a float triple. Default value is (1,1,1).
- The specular coefficients of the material as a float triple. Default value is (1,1,1).
- The specular exponent of the material as a float. Default value is 32.
- The position vectors in view space
- The normal vectors in view space

Point Light

The *point light* effect adds light contribution to `IMaterial`-implementations using a fragment
`PipelinePartTechniquePart`. Therefore, this Cg-interface is usually called by an `IMaterial`-implementation. This technique part implements the `ILight` Cg-interface. The task of an `ILight`-implementation is to calculate the light direction and the light intensity at a given fragment. Its task is *not* to shade the fragment.

The incoming light direction at a fragment is calculated as the direction from the light position to the fragment position. The light intensity at a fragment is equal to the specified effect input.

Inputs of the technique part:

- The position of the light as a float triple. Default value is (0,0,0).
- The intensity of the light as a float triple - one float for every color channel. Default value is (1,1,1). These values also can be greater than one if using HDR-rendering (as described in [Schlick94]).

Attenuated Point Light

The *attenuated point light* effect adds light contribution to `IMaterial`-implementations whose intensity is decreasing with the distance of the fragment to the light position. Therefore, this Cg-interface is usually called by an `IMaterial`-implementation. It is implemented using a fragment `PipelinePartTechniquePart`. This technique part implements the `ILight` Cg-interface. The incoming light direction at a fragment is calculated as the direction from the light position to the fragment position. The light intensity I_f at a fragment is calculated using the formula in Equation 3.5, where I_L is the light intensity at the light position, d is the distance between the light position and the fragment position, A_C is the constant attenuation value, A_L is the linear attenuation factor and A_Q is the quadratic attenuation factor.

$$I_f = \frac{I_L}{A_C + A_L * d + A_Q * d^2} \tag{3.5}$$

Inputs of the technique part:

- The position of the light as a float triple. Default value is (0,0,0).
- The intensity of the light as a float triple - one float for every color channel. Default value is (1,1,1). These values also can be greater than one if using HDR-rendering.
- The light attenuation values as a float triple (constant, linear, quadratic). Default value is (1,0,0).

Spot Light

The *spot light* effect adds light contribution to `IMaterial`-implementations using a fragment
`PipelinePartTechniquePart`. Thus, this Cg-interface is usually called by an `IMaterial`-implementation. This technique part implements the `ILight` Cg-interface. The incoming light direction at a fragment is calculated as in the *Point Light* effect. The light intensity I_f at a fragment is calculated using the formulas in Equation 3.6, Equation 3.7, Equation 3.8, Equation 3.9, where I_L is the light intensity at the light position, P_L is the light position, P_f is the fragment position, D_L is the light direction, θ_{min} is the minimum value of θ_f for a fragment to be lit.

$$D_f = P_L - P_f \tag{3.6}$$

$$\theta_f = max(dotproduct(D_f, D_L), 0) \tag{3.7}$$

$$smoothstep(a, b, x) = \begin{cases} 0 & \text{for} \quad x < a \\ 1 & \text{for} \quad x > b \\ -2 * (\frac{x-a}{b-a})^3 + 3 * (\frac{x-a}{b-a})^2 & \text{for} \quad a \leq x \leq b \end{cases} \tag{3.8}$$

$$I_f = I_L * smoothstep(\theta_{min}, 1, \theta_f) \tag{3.9}$$

Inputs of the technique part:

- The position of the light as a float triple. Default value is (0,0,0).
- The intensity of the light as a float triple - one float for every color channel. Default value is (1,1,1). These values also can be greater than one if using HDR-rendering.
- The direction of the light spot as a float triple. Default value is (0,0,-1).
- The light minimum θ_f as calculated in Equation 3.7 as a single float. The default value is 0.995.

Attenuated Spot Light

The *attenuated spot light* effect adds light contribution to `IMaterial`-implementations using a fragment
`PipelinePartTechniquePart` whose intensity is decreasing with the distance of the fragment to the light position. Therefore, this Cg-interface is usually called by an `IMaterial`-implementation. This technique part implements the `ILight` Cg-interface. The incoming light direction at a fragment is calculated as in the *Spot Light* effect. The light intensity

I_f is calculated as described in Equation 3.9. Additionally I_f is multipled by the attenuation factor f_A of Equation 3.10, where d is the distance between the light position and the fragment position, A_C is the constant attenuation value, A_L is the linear attenuation factor and A_Q is the quadratic attenuation factor.

$$f_A = \frac{1}{A_C + A_L * d + A_Q * d^2} \tag{3.10}$$

Inputs of the technique part:

- The position of the light as a float triple. Default value is (0,0,0).
- The intensity of the light as a float triple - one float for every color channel. Default value is (1,1,1). These values also can be greater than one if using HDR-rendering.
- The direction of the light spot as a float triple. Default value is (0,0,-1).
- The light minimum θ_f as calculated in Equation 3.7 as a single float. The default value is 0.995.
- The light attenuation values as a float triple (constant, linear, quadratic). Default value is (1,0,0).

Directional Light

The *directional light* effect adds light contribution to `IMaterial`-implementations using a fragment
`PipelinePartTechniquePart`. Thus, this Cg-interface is usually called by an `IMaterial`-implementation. This technique part implements the `ILight` Cg-interface. The incoming light direction at a fragment is equal to the specified light direction of the effect. The light intensity at a fragment is equal to the specified light intensity of the effect.

Inputs of the technique part:

- The direction of the light as a float triple. Default value is (0,0,-1).
- The intensity of the light as a float triple - one float for every color channel. Default value is (1,1,1). These values also can be greater than one if using HDR-rendering.

Vertex Color Effect

The *vertex color* effect takes the vertex color attribute of a vertex as input and uses it to manipulate either the ambient, diffuse or specular color of a fragment. Possible manipulation operations include add, substract, multiply and setting the value.

This effect is implementing the `IColorChanger` Cg-interface using a fragment
`PipelinePartTechniquePart`.

Inputs of the technique part:

- The vertex color attribute of the vertices

Changing Texture Coordinates

The *texcoordinate changer* effect manipulates the texture coordinates of a fragment by first scaling the coordinates and adding an offset value afterwards.

This effect is implementing the `ITexCoordChanger` Cg-interface using a fragment `PipelinePartTechniquePart`.

Inputs of the technique part:

- The vertex texture coordinate of the vertices
- The scaling factor as a simple float. Default value is 1.
- The offset value as a simple float. Default value is 0.

Texture Mapping

The *texture mapping* effect changes either the ambient, diffuse or specular color of a fragment by fetching a value from a 2D texture and manipulating the current color value of the fragment. Possible manipulation operations include add, substract, multiply and setting the color.

This effect is implementing the `IColorChanger` Cg-interface using a fragment `PipelinePartTechniquePart`.

Inputs of the technique part:

- The vertex texture coordinate of the vertices
- A 2D texture to fetch the color values from

Normal Mapping

The *normal mapping* effect changes the normal vector of the vertices by fetching the displacement values from a 2D texture and is implemented using a vertex and a fragment `PipelinePartTechniquePart`. This effect is often referred to as *bump mapping* as well. The vertex part implements the `IViewSpaceTransformer` Cg-interface and transforms the vertex binormal and tangent vector from object- to view space. The fragment part implements the `INormalTransformer` Cg-interface and displaces the normal vector with the fetched value from the 2D texture.

Inputs of the vertex part:

- The world-view-matrix
- The tangent vectors in object space
- The binormal vectors in object space

Outputs of the vertex part:

- The tangent vectors in view space
- The binormal vectors in view space

Inputs of the fragment part:

- The tangent vectors in view space
- The binormal vectors in view space
- The vertex texture coordinate of the vertices
- A 2D texture to fetch the displacement values from

Parallax Mapping

The *parallax mapping* effect implements parallax mapping [KanekoEtAl01] using one vertex and two fragment
`PipelinePartTechniqueParts`. The vertex part implements the `IViewSpaceTransformer` Cg-interface and transforms the vertex binormal and tangent vector from object- to view space. The first fragment part implements the `ITexCoordChanger` Cg-interface and changes the texture coordinates of the fragment by fetching the displacement value from a 2D texture. The second fragment part implements the `INormalTransformer` Cg-interface and displaces the normal vector similar as with the *Normal Mapping* effect.

Inputs of the vertex part:

- The world-view-matrix.
- The tangent vectors in object space
- The binormal vectors in object space

Outputs of the vertex part:

- The tangent vectors in view space
- The binormal vectors in view space

Inputs of the `ITexCoordChanger` *part:*

- The tangent vectors in view space
- The binormal vectors in view space
- The fragment position in view space
- The fragment normal in view space

- The vertex texture coordinate of the vertices
- A 2D texture to fetch the displacement values from. This texture is supposed to have the height-values stored in its alpha channel.
- A height scale factor as a single float. Default value is 0.01.

Inputs of the `INormalTransformer` *part:*

- The tangent vectors in view space
- The binormal vectors in view space
- The vertex texture coordinate of the vertices
- A 2D texture to fetch the displacement values from. This is usually the same texture object as of the `ITexCoordChanger` part.

Relief Mapping

The *relief mapping* effect implements relief mapping using one vertex and two fragment `PipelinePartTechniqueParts`. Details on relief mapping
can be found in [OliveiraAtAl00]. The implementation is very similar to the one used in the *Parallax Mapping* effect. The only difference lies in the Cg source code.

All inputs and outputs of the technique parts are the same as with the *Parallax Mapping* effect.

Render to Cubemap

The *render to cubemap* effect renders objects into a cubemap where the camera position is placed at the center of the bounding volume of an object. Therefore, this effect is dependent on a mesh node.

This effect is needed to use dynamic cubemapping. The description of the *Dynamic Cube Mapping* effect explains the details on how to use the generated cubemap.

Effect Properties:

- MeshID: The `Yare::Core::Engine::Identifier` of a mesh
- CubeMapSize: The size of one face of the cubemap as float, e.g. *256* creates a cubemap where each face has *256 x 256* pixels.

This effect changes the culling volume of the objects to the volume defined by the view frustums of the 6 cameras used to render the scene into the cubemap. This volume is defined by the center of the bounding volume of the object and the far plane of the used camera.

This is necessary to ensure that objects are still rendered to the cubemap even when they are not visible from the scene camera's point of view. The objects need to be included into the cubemap as long as they are visible from the object's point of view.

The default technique of this effect (`RenderToCubeMapTechnique`) consists of only one `ITechniquePart` implementation: `RenderToCubeMapPass`, which is derived from `AdditionalPass`.

Since this effect interacts with the state table, a unique name to interact with a resource-slot of the latter is needed. As only one cubemap per mesh exists, the mesh ID with a prefix ('*CubeMap*') is used as resource ID.

Effect Inputs: none

State Dependencies Before Rendering:

- The resource-slot must not be locked

State Changes After Rendering:

- The ready flag of the resource-slot is set
- The resource-slot is locked
- The value of the resource-slot is set to the `ISampler` of the cubemap

The pass of this effect performs the following steps:

1. Remove the view and projection matrix variables of the passed variables, so they do not overwrite the set renderstates of this effect.
2. Activate the projection matrix for a 90 ° field of view.
3. Activate the cubemap as rendertarget.
4. Then, for every face of the cubemap the according view matrix is set and the provided geometries are directly rendered into the current face.
5. Deactivate the cubemap as rendertarget.
6. Execute the state changes.

Dynamic Cube Mapping

The *dynamic cube mapping* effect uses the cubemap generated by a *Render to Cubemap* effect to implement a reflective material behavior of an object. To select a specific cubemap this effect also needs the ID of a mesh as parameter.

The effect uses one vertex and one fragment `PipelinePartTechniqueParts`. The vertex part implements the `IViewSpaceTransformer` Cg-interface and transforms the vertex position and normal vector from object- to view space. The fragment part implements the `IReflector` Cg-interface and reflects the view

direction at the fragment and fetches the color value from the cubemap using the reflected ray. This color value is then blended with the current color value of the fragment.

To get the cubemap's `ISampler` value (which is created by the *Render to Cubemap* effect) this effect uses a special type of input called *state input* (`StateInput`). This is an effect input which fetches its value from the state table by using an ID of a resource-slot and taking the value of this slot. The ID of the resource-slot of this effect input is the ID as described in the *Render to Cubemap* effect.

Inputs of the vertex part:

- The transposed and inverted world-view-matrix
- The position vectors in object space
- The normal vectors in object space

Outputs of the vertex part:

- The position vectors in view space
- The normal vectors in view space

Inputs of the fragment part:

- The cubemap texture sampler taken from the state table as described above
- The blending weights to mix the color value of the cubemap with the current color of the fragment
- The position vectors in view space
- The normal vectors in view space

State Dependencies Before Rendering:

- The ready flag of the resource-slot is set

Glass Effect

The *glass* effect uses reflection and refraction to simulate a glass-like appearence of an object. The implementation is very similar to the *Dynamic Cube Mapping* effect. It only needs some more effect inputs of the user:

- The red, green and blue refraction indices of the material as a float triple. Default value is (0.8, 0.82, 0.84).
- Blending weights for current fragment color, the reflected color and the refracted color

Displacement Mapping

The *displacement mapping* effect displaces the vertex positions in object space by a value which is fetched from a 2D texture. The normal vector of the vertex is taken as the displacement direction. The value from the 2D texture is multiplied by the float scaling effect input.

This effect is implementing the `IObjectSpaceTransformer` Cg-interface using a vertex `PipelinePartTechniquePart`.

Inputs of the technique part:

- The vertex texture coordinate of the vertices
- The normal vector of the vertices
- A 2D texture to fetch the displacement values from
- A float scaling factor. Default value is 1.

Render to Depth Texture

The *render to depth texture* effect renders objects into a depth texture (also called *Shadowmap*) from a light point of view. Therefore, this effect is dependent on a light source which emits variables with the name `LightViewMatrix` and `LightProjectionMatrix`.

This effect is needed to use shadow mapping for a scene. The description of the *Spot Light Shadow Mapping* effect explains the details on how to use the generated depth texture for shadow mapping.

Effect Properties:

- LightID: The `Yare::Core::Engine::Identifier` of a light
- ShadowMapSize: The size of the shadowmap as float, e.g. *1024* gives a *1024 x 1024* pixels sized shadowmap.

This effect changes the culling volume of the objects to the light frustum of the associated light. This is necessary to ensure that objects are still rendered to the depth texture even when they are not visible from the camera's point of view. The objects need to be included into the shadowmap if they intersect the light frustum and are not culled as long as this volume is visible to the camera.

The default technique of this effect (`RenderToShadowMapTechnique`) consists of only one `ITechniquePart` implementation: `RenderToShadowMapPass`, which is derived from `AdditionalPass`.

Since this effect interacts with the state table, a unique name to interact with a resource-slot of the latter is needed. As only one shadowmap per light can exist, the light ID with a prefix ('*ShadowMap*') is used as resource ID.

Effect Inputs:

- The light view matrix of the associated light
- The light projection matrix of the associated light

State Dependencies Before Rendering:

- The resource-slot must not be locked

State Changes After Rendering:

- The ready flag of the resource-slot is set
- The resource-slot is locked
- The value of the resource-slot is set to the `ISampler` of the shadowmap

When the pass converter assigns the `ITechniqueParts` of a scene object to the additional pass of this effect, every added technique part is asked for a *depth-only* version of itself. If the asked part has no version which manipulates the depth buffer of the rendered object, it is discarded. This method is used to speed up rendering into the depth buffer by ignoring technique parts which e.g. only change the color but not the depth value of the fragments. See Subsection 3.8.8 for more information on this topic.

The pass of this effect performs the following steps:

1. Replace the view and projection matrix variables of the scene objects with the light view and projection matrix variables.
2. Disable color updates of the red, green, blue and alpha component of the rendertarget. This is done to speed up the rendering process.
3. Push the depth texture onto the rendertarget stack.
4. Render the objects using a generated vertex- and fragment-program. (It is not necessary that programs are generated - this depends on the other technique parts of the scene object.)
5. Pop the depth texture from the rendertarget stack.
6. Execute the changes to the state table.
7. Restore the old color update renderstates.

Spot Light Shadow Mapping

The *spot light shadow mapping* effect implements shadow mapping with respect to a spot light using a depthmap created by a *Render to Depth Texture* effect as described in the previous chapter. Therefore, it also needs the ID of a spot light to refer to the depthmap.

The effect uses one vertex and one fragment `PipelinePartTechniqueParts`. The vertex part implements the `IWorldSpaceTransformer` Cg-interface and transforms the vertex position from object space to the projection space of the corresponding spot light. The fragment part implements the `ILight` Cg-interface and calculates the incoming light direction and light intensity at a fragment as described in the *Spot Light* effect. Additionally, it checks every fragment if it is in shadow. Thus if a fragment receives no light, this effect sets the light intensity of this fragment equal to zero.

To get the depthmap's `ISampler` value (which is created by the *Render to Depth Texture* effect) this effect uses a *state input* as described in *Dynamic Cube Mapping*. The ID of the resource-slot for this input is the same ID as described in the *Render to Depth Texture* effect.

Inputs of the vertex part:

- The combined view and projection matrix of the corresponding spot light
- The constant depth bias value as described in [Williams78]
- The vertex position in object space

Outputs of the vertex part:

- The position vectors in the lights projection space

Inputs of the fragment part:

- The depthmap sampler taken from the state table as described above
- The position vectors in the lights projection space
- The position of the light as a float triple. Default value is (0,0,0).
- The intensity of the light as a float triple - one float for every color channel. Default value is (1,1,1). These values also can be greater than one if using HDR-rendering.
- The direction of the light spot as a float triple. Default value is (0,0,-1).
- The light minimum θ_f as described with the *Spot Light* effect. The default value is 0.995.

State Dependencies Before Rendering:

- The ready flag of the resource-slot is set

Visualize Tangent Space

The *visualize tangent space* effect uses an additional pass to the render tangent space vectors of the vertices of a geometry. At every render call of this pass the effect checks if it has already calculated the visual tangent space vectors and renders them as a line list.

Effect Properties:

- VectorLength: The length of the visual tangent space vectors
- Boolean flags if to render the normal, the binormal and the tangent vector

Cg File Effect

The *Cg file* effect uses the `PipelinePass` class (as described in Subsection 3.8.5) to use an external vertex- and fragment program for rendering. These programs are specified by the user per filename.

CgFX Effect

The *CgFX* effect completely represents the contents of a CgFX effect file using techniques and passes of the YARE effect framework. For this, the user specifies an effect filename and this effect loads all techniques and passes contained in this file.

Blooming

The *blooming* effect is an example for using the postprocessing framework of YARE. For this it extends the `PostProcessingPass`-class by inheriting from it. At runtime this new pass uses several sub-steps for getting the final blooming effect:

1. Downsample the original image by rendering into a smaller rendertarget.
2. Perform a horizontal blur on the downsampled image using a special fragment program.
3. Perform a vertical blur on the blurred image using a special fragment program.
4. Generate the final image by combining the blurred image with the original image using a special fragment program.

The sub-passes of the blooming technique are presented in Figure 3.17.

Figure 3.17: The passes to implement *blooming*. From left to right: The original image, the downsampled image, the horizontally blurred image, the complete blurred image, the final image.

3.8.11 Extending the Framework

The very first task for adding a new effect to the framework is to evaluate which technique parts are required to implement the desired effect. This can include implementations of existing Cg interfaces, new Cg interfaces or even additional render passes.

The next step is to provide the necessary C++ classes. This includes an effect class (implementing `IEffect`) with one or more `ITechnique`-implementations and one or more `ITechniqueParts`.

To add a new implementation of an existing Cg interface, it is sufficient to put the Cg source code into a textfile placed in the implementation data directory as mentioned in the section *Configuration of Cg-interfaces*.

To add a new Cg interface, the Cg source code must be placed into the interface data directory (`Data\Effects\Interfaces`) along with an XML file describing this interface. Again see section *Configuration of Cg-interfaces* for details. The interface name must also be put into the XML file in the config-directory of the effect framework (`Data\Effects\Config`) to specify the calling order.

To add an additional render pass class it is sufficient to derive a new C++ class from `AdditionalPass` and implement the `Render()`-method.

When it happens that the provided Cg framework programs are not flexible enough, new programs can be provided/generated by extending or deriving the `BaseFrameworkProgram`-class.

The Cg framework programs define the point were the general concept of introducing arbitrary Cg interfaces meets the specialization for a concrete rendering pipeline implementation. It is allowed to define and implement any kind of Cg interfaces, but at a certain point the interaction between these interfaces has to be implemented. Therefore, the Cg framework programs call the interfaces in the correct order and pass the desired parameters to their methods.

3.8.12 Emitting Effects

Emitting effects in the system is very similar to emitting variables as described in Section 3.6. The only difference is that objects that distribute effects are called *effect emitters* and their classes are all implementing the `IEffectEmitter`-interface.

3.8.13 Comparison with Existing Solutions

This section compares the implemented effect framework with existing solutions of effect and shader handling as presented in Subsection 2.3.3.

Individual-Program Approach

New vertex- and fragment programs can be integrated into YARE nearly as easy as with the individual-program approach (see Subsection 3.8.11 for details). But YARE is also able to manage the resulting rendering passes and can handle scene-wide effects.

Effect Files

Effect files can also be used with YARE through the support of CgFX effect files. The used effect code will replace the Cg framework program (as described in Subsection 3.8.3) during rendering. Effect files are a convenient way for fast prototyping of new effects. Later on, this new effect could be integrated into the effect framework (using Cg-interfaces and their Cg-implementations) to make them more reusable and mergable with other effects.

Uber Shaders

With Uber Shaders, it is the task of the user to integrate all possible effects into one valid shader - YARE is able to automatically solve this task. Additionally, the source code of the vertex- and fragment programs stays readable through the seperation into small technique parts.

Micro Shaders

The idea of Micro Shaders to concatenate small code fragments to build the desired shader code is very similar to the concept implemented in YARE. The main difference is that Micro Shaders work on the source code level, but YARE merges the shader fragments at a higher level. This enables YARE to reorder and to disable unneeded fragments. Using the Cg interface feature, YARE does not have to care about variable names, register allocations or other code breaking problems. Additionally, the shader fragments (called technique parts in this context) can include render states, which are merged and applied automatically by the system.

Abstract Shade Trees

Abstract Shade Trees provide similar functionality to the *low-level part* of the effect framework of YARE (generation of shader source code). Additionally, Abstract Shade

Trees can solve dependencies on the source code level. The *high-level part* of YARE (automatic management of multipassing, render targets and render states, and solving dependencies between multiple rendering effects) is not implemented by the Abstract Shade Trees. The reason for this is that Abstract Shade Trees are designed to generate the source code for *one rendering effect*. YARE, on the other hand, is designed to allow easy configuration of many rendering effects in a scene and to automatically handle their interactions.

3.8.14 The Interfaces

See Section A.3 for the C++ interfaces of this chapter.

Name	Related	Space	Desc
IObjectSpaceTransformer	Vertex	Object	Transforms vertex attributes in object space.
IWorldSpaceTransformer	Vertex	World	Transforms vertex attributes in world space.
IViewSpaceTransformer	Vertex	View	Transforms vertex attributes in view space.
IProjectionSpaceTransformer	Vertex	Projection	Transforms vertex attributes in projection space.
IColorChanger	Fragment	-	Changes the ambient-, diffuse and/or specular color of a fragment.
ITexCoordChanger	Fragment	-	Changes the texture coordinates of a fragment.
INormalTransformer	Fragment	-	Transforms the normal of a fragment.
ILight	Fragment	-	Illuminates a given fragment.
IMaterial	Fragment	-	Evaluates a shading equation using provided ILight Cg-interfaces.

Table 3.6: Available Cg-interfaces of the effect framework

```
RenderTarget.Activate(T_0)
DepthBuffer.Activate(D_0)
Scene.Render();
current_colormap ⇐ 0;
current_depthmap ⇐ 0;
for i = 0 to passes.Size − 1 do
  if i == passes.size() − 1 then //The last pass.
    RenderTarget.Activate(Framebuffer)
    if i == 0 then
      current_colormap ⇐ T_0;
      current_depthmap ⇐ D_0;
    else if i is even then
      current_colormap ⇐ T_1;
      current_depthmap ⇐ D_1;
    else
      current_colormap ⇐ T_2;
      current_depthmap ⇐ D_2;
    end if
  else if i is even then //The 1st, 3rd, 5th,... pass.
    RenderTarget.Activate(T_2)
    DepthBuffer.Activate(D_2)
    if i == 0 then
      current_colormap ⇐ T_0;
      current_depthmap ⇐ D_0;
    else
      current_colormap ⇐ T_1;
      current_depthmap ⇐ D_1;
    end if
  else //The 2nd, 4th, 6th,... pass.
    RenderTarget.Activate(T_1)
    DepthBuffer.Activate(D_1)
    current_colormap ⇐ T_2;
    current_depthmap ⇐ D_2;
  end if
  passes[i].SetInputTextures(current_colormap, current_depthmap);
  passes[i].Render();
end for
```

Algorithm 3.2: Pseudocode for rendering post-processing passes

3.9 The Drawgraph

The drawgraph is a data structure used to accelerate rendering. As input the drawgraph needs a complete data view of all objects (with their bounding volumes) that could potentially be rendered, and as output the graph gives a list of objects that are visible with respect to the list of culling objects and therefore are sent to the graphics device for rendering afterwards.

Given the culling function as described in Equation 3.11

$$cf(o,c) = \begin{cases} 1 & \text{object } o \text{ is culled by culler } c \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

and the visibility of an object as described in Equation 3.12

$$v(o) = \begin{cases} 1 & \sum_{i=1}^{N_c} cf(o,c_i) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

the task of the drawgraph for finding the set of visible objects $O_{output} \subseteq O_{input}$ can be formulated the following way:

$$O_{output} = \{o \in O_{input} \mid v(o) = 1\} \tag{3.13}$$

3.9.1 Solving the task

The easiest approach for solving the task of the drawgraph as described in Equation 3.13 would be to simply enumarate every single object of the input list and testing it against all given cullers. Even though this algorithm is mathematically correct it is not recommendable since testing every object can result in an even lower performance than without using a drawgraph. This is the case if rendering an object is faster than testing it against all culling objects.

As the name *drawgraph* implies, common solutions to this problem use a graph as internal data structure. Possible approaches are using a kd-tree (Subsection 2.4.2), an octree (Subsection 2.4.1) or similar data structures as described in Section 2.4.

The drawgraph in YARE is kept abstract using an interface definition (`Yare::Graphics::DrawGraph::IDrawGraph`) and thus is open for all possible implementations using the algorithms and data structures as described above. At the moment an octree implementation is provided to the users of YARE. A screenshot of using this technique can be seen in Figure 3.18. Since it is not efficient to rebuild the graph every frame, the drawgraph interface also provides methods for removing single objects from the graph and telling it when an object has changed its position or bounding volume.

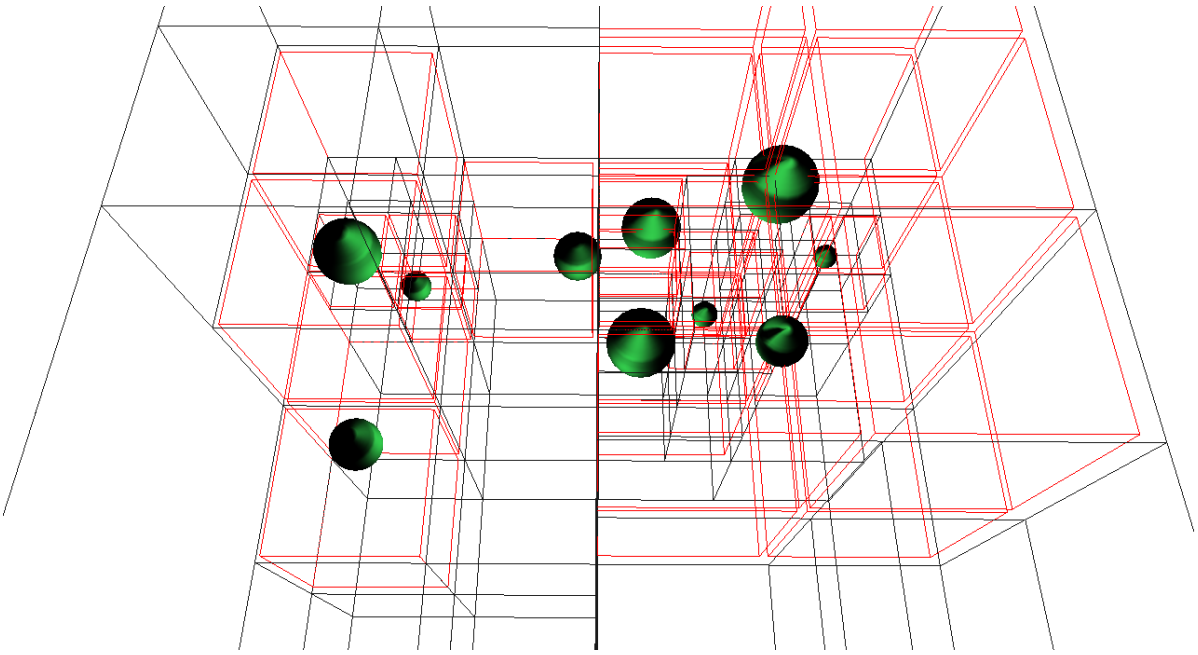

Figure 3.18: The octree algorithm in use. Cells that contain renderable objects are drawn in red.

3.9.2 The Renderable Class

Implementations of the drawgraph interface (like the *Coherent Hierarchical Culling* algorithm presented by Bittner et al. [BittnerEtAl04]) might need to render the objects given as input beforehand. For this reason not only the geometry with its bounding volume is provided to the drawgraph, but a collection of all data needed to render an object. This includes

- geometric data,
- passes ready for rendering,
- source effects from which the render passes originate,
- variables used for rendering (renderstates, pass inputs),
- bounding volume in object- and world space,
- effect bounding volume (the world space bounding volume influenced by all effects this object is rendered with)

This data is grouped together in a class called `Yare::Graphics::DrawGraph::Renderable`. Thus, this class is the smallest collection of data needed to correctly render an object and is often needed for resorting and pre-rendering (e.g. occlusion culling) objects. As with *Coherent Hierarchical Culling*, an implementation of the algorithm could implement the drawgraph interface and would handle the complete rendering of the scene inside the drawgraph (including occlusion culling and rendering into the framebuffer). This is only

possible, since the renderable objects contain all necessary data for rendering (including rendering effects and passes). To prevent additional rendering through the rest of the system, this drawgraph implementation would not expose any visible objects.

3.9.3 The Drawgraph Interface

See Section A.4 for the C++ interface for the drawgraph.

3.10 The Scenegraph

The task of the scenegraph in YARE is:

- Provide a structured view of all data in the scene.
- Enable the user of the scenegraph to get access to the data of the scene (including meshes, lights, effects, textures and similar elements).
- Synchronize information about renderable objects with the underlying drawgraph (as described in Section 3.9).
- Store hierarchical transformation information of the scene objects.
- Enable the user to group logically connected objects together.
- Build a memory representation of common scenegraph file formats like Open Inventor files.

To further circumscribe the task of the scenegraph it is also described what the scenegraph is *not* supposed to do:

- Culling of objects.
- Send drawing commands to the graphics device.
- Call of any graphics API functions like OpenGL and Direct3D.

So the main task of the scenegraph in YARE is to organize the objects in a structured, hierarchical way. Neither are algorithms performed by the scenegraph, nor does it directly render any objects. The reason for this approach is to completely disentangle the data and the operations performed on it. The separation into these two parts is illustrated in Figure 3.19. That way it makes it easier to change or replace existing algorithms. Not even the introduction of new operations makes it necessary to change any code on the classes which only provide the data for the algorithms. This leads to a clean internal behavior and well-structured API of the scenegraph. This approach also has some disadvantages which can be found in Subsection 3.10.1.

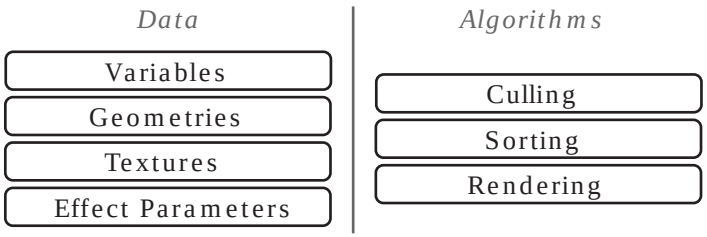

Figure 3.19: The complete disentangling of data and algorithms in YARE

To solve the task the scenegraph in YARE organizes the objects in a directed acyclic graph (DAG). A DAG is a directed graph with no directed cycles. If a node class of the

scenegraph can have child nodes, it is called a *group node*, if it cannot have children it is called a *leaf node*. Subsection 3.10.2 gives a complete listing of group and leaf nodes available in YARE.

3.10.1 Differences to Conventional Scenegraphs

The following paragraphs give details on different concepts used by YARE with respect to other available scenegraph frameworks as listed in Section 2.5.

Direct graphics API calls are not allowed in the scenegraph of YARE. This is due to the abstraction of the graphics API through the rendering interface (Section 3.7). Another reason is that the scenegraph is not responsible for rendering the objects and therefore is not needed to send drawing commands or renderstate changes to the graphics device. As disadvantage can be seen that it is more difficult to add special rendering behavior to the system. With other scenegraph frameworks it is possible to directly manipulate renderstates, e.g. for rapid implementation of new effects. This is not allowed and even not possible with YARE. On the other hand it is not necessary at all since the responsibility of rendering the scene objects in a desired way is shifted to the effect framework. So there is still a place where quick code changes for testing can be made.

The scenegraph is not responsible for culling out renderable objects. This task is shifted to the drawgraph (as described in Section 3.9). An advantage of this approach is that the culling algorithm is not fixed to the hierachical bounding volume culling as implemented in other scenegraphs. In fact, the used culling algorithm (e.g. using a kd-tree, Octree and/or occlusion queries) can be selected and replaced at runtime by simply replacing the drawgraph with another implementation. The disadvantage of this approach over other scenegraph frameworks is that the latter have the possiblity for early culling out objects which are not visible on the screen. This prevents that costly operations are performed on objects which are not visible later on. This is not possible with the implemented graphics pipeline of YARE anyway since applied rendering effects can change the culling volume of an object (see Section 3.8 for details) which makes culling in the scenegraph impossible.

Scenegraph traversals using e.g. the visitor design pattern (see Section 2.6 for details) are an important instrument for other scenegraph frameworks. Operations on the scenegraph nodes are performed by means of traversals. Since the scenegraph of YARE does not need to perform operations on the nodes (as described above), traversals are not necessary. Another approach is taken which allows the user of the system to directly look up information information without costly traversals. This approach is explained in detail in Subsection 3.10.4.

The scenegraph uses no *traversal state object* during the processing of the graph to record data changes of the scene nodes. This concept has been replaced by the concept of *emitted variables* as described in Subsection 3.6.5. This gives greater flexibility to the propagation/emitting scheme, since every variable emitter can decide to where its

data is emitted (e.g. the structural context, the whole scene graph or to all objects lying in a specific bounding volume). Most other scenegraph APIs have a fixed propagation scheme which applies to all nodes in the graph. See Section 2.5 for propagation examples in traditional scenegraph APIs.

3.10.2 Scenegraph Objects

Node Classes

This section lists available scenegraph node classes of YARE and gives details on their functionalities.

The base class of all objects in the scenegraph is the `Node`-class. The main properties of a scenegraph node are:

- A link to the parent node. This can be `NULL` if the node is the root node of the scenegraph.
- Different kinds of bounding volumes. The node can be asked for a bounding sphere and bounding box. It can also be defined whether the returned bounding volume should contain only the node itself or all child nodes too.
- The `Node`-class also implements the `IPositioned`-interface which lets the user ask for a position of the node in world space.

The `Group`-class implements methods to organize two or more nodes together as children of this node.

Special kind of group nodes are implemented by the following classes:

- `RootGroup`: This node cannot have a parent but only children. The intended application of this class is that it is used as the root node of the scenegraph.
- `Separator`: This group does not allow transport of structural data like variables and render effects outside of this group. For more information on structural data see Subsection 3.6.5, Subsection 3.8.12 and Subsection 3.10.6.
- `TransformGroup`: This group additionally stores transformation information which is applied to all children of this group.
- `Switch`: This group can dynamically include and exclude its children from the scenegraph. The interface allows selecting all, none and mask-based children. With mask-based selection a boolean flag for every child is used to include or to exclude this node.

The opposite to a group node is the leaf node as implemented with the `Leaf`-class which has no child nodes and therefore cannot be the parent of any other node.

Special kind of leaf nodes are implemented by the following classes:

- `Camera`: This class represents a camera by its position, direction, up-vector, field of view, near plane and far plane. It also implements the `IVariableEmitter` -interface (as described in Subsection 3.6.5) - the emitted variables contain all the data as listed in the previous sentence.
- `Effect`: This scenegraph node is just a wrapper for an effect as described in Section 3.8, so that the effect can be used in the scenegraph. The task of this node is to emit the contained effect as described in Subsection 3.8.12.
- `FileTexture`: This class wraps a texture object loaded from file. The texture is then emitted as a variable to be used within the scenegraph.
- `Light`: This node emits a variable containing the intensity of the light, and another variable containing the attenuation factors as described in Subsection 3.8.10. This is also the base class of all other lights.
- `DirectionalLight`: This class is derived from the `Light`-class above and additionally emits a variable containing the direction of the light.
- `PointLight`: This class is derived from the `Light`-class above and additionally emits a variable containing the position of the light.
- `SpotLight`: This class is derived from the `Light`-class above and additionally emits variables containing the following data: the position and direction of the spot light, the angle of the light cone, the view matrix and the projection matrix of the spot light.
- `Mesh`: This node contains one or more `IGeometry`-objects. It can emit variables containing textures, renderstates and material properies of the contained geometries. This class is used to construct renderable objects out of its geometries as described in Subsection 3.10.6.
- `DefaultVariableEmitter`: This node emits a used definable variable to the scenegraph. The variable can be of the base types float, double, bool and integer and any tupel of them.
- `RenderStateEmitter`: This node emits a user-selectable renderstate variable which can be used as described in Subsection 3.8.5.

The UML diagram of the scenegraph classes can be found in Appendix B.

Additional Classes

The scenegraph framework of YARE contains the following additional classes:

- `Global`: The class which puts everything together: it has a reference to the root node of the scenegraph and a reference to the drawgraph. It handles the events sent by the scene nodes and contains the method which an application must call every frame to update the scenegraph. Since this class must be instanced only

once per application, it is implemented according to the *singleton design pattern*. This instance is also called the *global scenegraph context*.

- `TransformStack`: A stack implementation with matrices as stack elements. The class can be asked for the combined transformation of all elements on the stack.
- `Database`: Described in detail in Subsection 3.10.4.

User-Controlled Nodes

Since the scenegraph data should be changeable by the user, the concept of the controller and controllable objects has been introduced. The controller (which implements the `IController`-interface) reacts to the user input, e.g. keyboard or mouse. Then it changes the data of the assigned controllable object which implements the `IController`-interface.

An example for a controller is the `FPS`-class which takes keyboard and mouse input to control objects in a way the player controls the main character in a 'first person shooter' game, hence the name of the class. Examples for controllable objects are the camera (`Camera`), the transformation group (`TransformGroup`) and all light classes (`Light`, `DirectionalLight`, `PointLight` and `SpotLight`).

3.10.3 Data Sharing

Scene setups can contain the same geometry multiple times, e.g. the four similar wheels of a car or a forest where a specific tree model is used a thousend times. Using a new geometry for every instance would consume more memory than if a single geometry is used and only referenced by the meshes in the scenegraph.

Some scenegraph frameworks (as presented in Section 2.5) use an approach to this problem by letting a scene node have multiple parents, or by introducing a special group and target node. YARE follows the concept of directly referencing geometry instances by the meshes in the scenegraph which is similar the the concept of node components as described in Subsection 2.5.2. E.g. for every similar tree in a forest a `Mesh`-instance is needed which is only a thin wrapper around a reference to a single geometry object of a tree. An example of this concept can be seen in Figure 3.20.

The reason for not using approaches where a node in the scenegraph is referenced multiple times is that with YARE every node in the scenegraph must be unique and can only have one path from the root node. That way the *structural context* of a node can be cached and directly looked up at any time.

The **structural context** of a node is its parent, its transformation, its bounding volume and the structural emitted variables and effects. See Subsection 3.6.5 for details on structural emitted data. With multiple referencing of nodes this would not be possible.

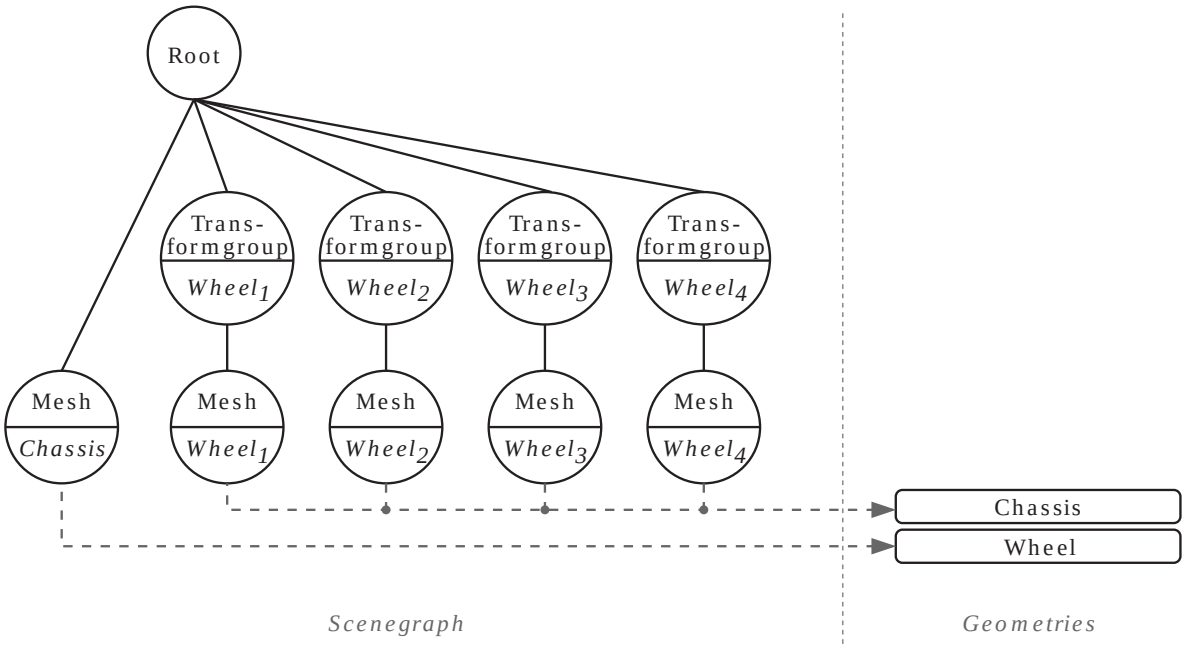

Figure 3.20: A sample scenegraph representing the chassis and the four wheels of a car. The wheel meshes share the same geometry data

E.g. if the user wants to know the position of the wheel, the system cannot know if the user means the left-front or the right-back wheel of the car. The user always has to follow the path from the root node to the intended node to find its structural context. This leads to a performance drawback compaired to the direct lookup of the data as presented in Subsection 3.10.4.

3.10.4 The Scene Database

The scenegraph itself with its nodes which are connected to each other using a parent-child relation is only kept to allow the user to organize the objects in a hierarchical structured way. Algorithms running on this graph are not supposed to directly reference leaf nodes but find them by traversing from the root node to the leafs. With scenegraphs containing hundreds of thousands of objects grouped together in thousands of groups (e.g. a scenegraph modelling a city), the traversal leads to non-negligible processing overheads. Especially, when the culling and rendering algorithm has to traverse the whole graph at every frame. The least traversal overhead is achieved if all nodes are direct children of the root node. However being forced to put all objects as direct children of the root for performance reasons does not allow the user to build logically or spatially organized groups.

To address both issues (let the user organize the nodes in a structured way and simul-

taneously allow fast traversals) the concept of a *Scene Database* is introduced in YARE. The task of the scene database is to allow algorithms direct access to the data of scene nodes. This includes:

- A list of all meshes in the scene.
- Direct lookup of structural context data of any node. See Subsection 3.10.3 for details on the structural context of nodes.
- A list of all emitted variables and effect in the scenegraph.
- A list of all culling objects as described in Section 3.9.
- A list of all controllers as described in Subsection 3.10.2.
- A list of all `Renderables` as described in Subsection 3.9.2.

The scene database is just another view of the data of the scene - it provides the same data as the scenegraph but in a flat and fast way. The only problem is to keep the two views synchronized. Therefore, if the user or a controller (as described in Subsection 3.10.2) changes data in the scenegraph, these changes must be *committed* to the scene database to update its internal state. The solution to this task is described in Subsection 3.10.5.

3.10.5 Updating the Scene Database

To synchronize the dataview of the scenegraph with the scene database, an event-based system is introduced in YARE. The processing sequence from a registered change in the scenegraph to an update in the scene database has the following steps:

1. The data of a node in the scenegraph has changed.
2. The node sends an event to the global scenegraph context containing the event ID and the node itself as sender object. The event ID indicates what type of data has been changed. A list of all available event IDs can be found below.
3. At the beginning of a frame the global scenegraph context processes all raised events.
4. For every event a list of tasks is generated which has to be performed on the scene database. A list of all available tasks can be found in the listing below. The generated tasks are collected in a list.
5. The list of all tasks is searched for duplicate tasks and is sorted by a given order. E.g. the world transformation matrices of the nodes need to be updated before their bounding volumes in world space can be recalculated.
6. Finally, the list of tasks is executed.

Using this concept, redundant work per scenegraph node is prevented. This is achieved by the elimination of duplicate tasks as listed above. Another benefit is, that a task is only performed on nodes which need to update their data. For example, the system only updates the structural context of those nodes, which are influenced by a data change in the scenegraph.

The performed steps are additionally explained by the sequence diagram in Figure 3.21.

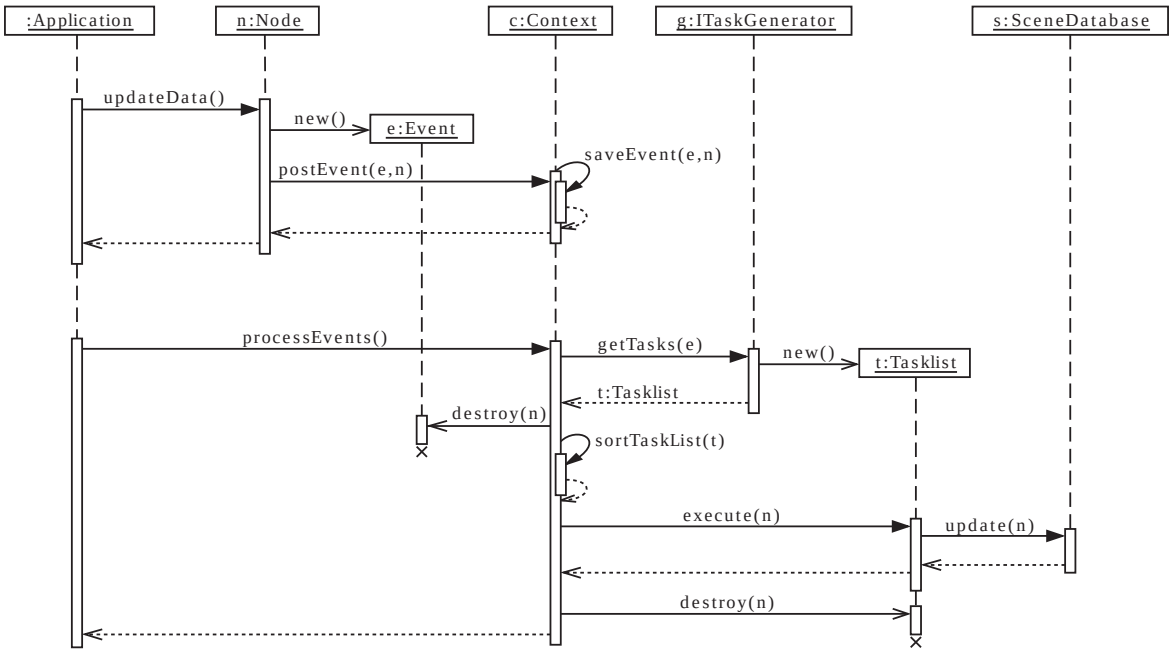

Figure 3.21: The sequence diagram of the event processing in YARE

The following list shows all available event IDs to report a change of data in a node to the global scenegraph context.

- `EID_CONTROLLER_ADDED`: A controlling object has been added to the scenegraph.
- `EID_CONTROLLER_REMOVED`: A controlling object has been removed from the scenegraph.
- `EID_TRANSFORMATION_CHANGED`: The transformation value of a transform group has changed.
- `EID_TRANSFORMATION_GROUP_ADDED`: A transformation group has been added to the scenegraph.
- `EID_TRANSFORMATION_GROUP_REMOVED`: A transformation group has been removed from the scenegraph.
- `EID_GROUP_ADDED`: A group has been added to the scenegraph.
- `EID_GROUP_REMOVED`: A group has been removed from the scenegraph.

- `EID_MESH_ADDED`: A mesh has been added to the scenegraph.
- `EID_MESH_REMOVED`: A mesh has been removed from to the scenegraph.
- `EID_MESH_CHANGED`: The number of geometries of a mesh has changed.
- `EID_BOUNDING_VOLUME_CHANGED`: The bounding volume of a node has changed.
- `EID_CAMERA_ADDED`: A camera object was added to the scenegraph.
- `EID_CAMERA_REMOVED`: A camera object was removed from the scenegraph.
- `EID_GLOBAL_VARIABLE_ADDED`: A global variable emitter was added to the scenegraph.
- `EID_GLOBAL_VARIABLE_REMOVED`: A global variable emitter was removed from the scenegraph.
- `EID_GLOBAL_VARIABLE_CHANGED`: The global variables of an emitter have changed.
- `EID_GLOBAL_EFFECT_ADDED`: A global effect emitter was added to the scenegraph.
- `EID_GLOBAL_EFFECT_REMOVED`: A global effect emitter was removed from the scenegraph.
- `EID_GLOBAL_EFFECT_CHANGED`: The global effects of an emitter have changed.
- `EID_VOLUME_VARIABLE_ADDED`: A volume variable emitter was added to the scenegraph.
- `EID_VOLUME_VARIABLE_REMOVED`: A volume variable emitter was removed from the scenegraph.
- `EID_VOLUME_VARIABLE_CHANGED`: The volume variables of an emitter have changed.
- `EID_VOLUME_EFFECT_ADDED`: A volume effect emitter was added to the scenegraph.
- `EID_VOLUME_EFFECT_REMOVED`: A volume effect emitter was removed from the scenegraph.
- `EID_VOLUME_EFFECT_CHANGED`: The volume effects of an emitter have changed.
- `EID_STRUCTURAL_VARIABLE_ADDED`: A structural variable emitter was added to the scenegraph.
- `EID_STRUCTURAL_VARIABLE_REMOVED`: A structural variable emitter was removed from the scenegraph.
- `EID_STRUCTURAL_VARIABLE_CHANGED`: The structural variables of an emitter have changed.
- `EID_STRUCTURAL_EFFECT_ADDED`: A structural effect emitter was added to the scenegraph.
- `EID_STRUCTURAL_EFFECT_REMOVED`: A structural effect emitter was removed from the scenegraph.
- `EID_STRUCTURAL_EFFECT_CHANGED`: The structural effects of an emitter have changed.

- `EID_BOUNDING_VOLUME_TYPE_CHANGED`: The bounding volume type of the scenegraph has changed.

The following list shows all available tasks to perform on the scene database.

- `TC_REMOVE_GLOBAL_VARIABLE`: All the emitted global variables of the provided variable emitter are removed from the scene database.
- `TC_REMOVE_GLOBAL_EFFECT`: All the emitted global effects of the provided effect emitter are removed from the scene database.
- `TC_REMOVE_VOLUME_VARIABLE`: All the emitted volume variables of the provided variable emitter are removed from the scene database.
- `TC_REMOVE_VOLUME_EFFECT`: All the emitted volume effects of the provided effect emitter are removed from the scene database.
- `TC_UNREGISTER_CONTROLLER`: The provided controller is removed from the scene database.
- `TC_UNREGISTER_CULLER`: The provided culling object is removed from the scene database.
- `TC_CREATE_RENDERABLES`: A list of renderable objects is generated from the provided meshes. See Subsection 3.10.6 for details.
- `TC_UPDATE_WORLD_MATRIX`: The world transformation matrix of the provided nodes is updated at the scene database.
- `TC_UPDATE_BOUNDING_VOLUME`: The bounding volume of the provided nodes is updated at the scene database.
- `TC_REGISTER_CULLER`: The provided culling object is registered at the scene database.
- `TC_REGISTER_CONTROLLER`: The provided controller is registered at the scene database.
- `TC_ADD_GLOBAL_VARIABLE`: The emitted global variables of the provided emitter are added to the scene database.
- `TC_ADD_GLOBAL_EFFECT`: The emitted global effects of the provided emitter are added to the scene database.
- `TC_ADD_VOLUME_VARIABLE`: The emitted volume variables of the provided emitter are added to the scene database.
- `TC_ADD_VOLUME_EFFECT`: The emitted volume effects of the provided emitter are added to the scene database.
- `TC_STRUCTURAL_VARIABLE_CONTEXT_UPDATE`: The variables of the structural context of the provided nodes are updated.
- `TC_STRUCTURAL_EFFECT_CONTEXT_UPDATE`: The effects of the structural context of the provided nodes are updated.

- `TC_UPDATE_WORLD_MATRIX_VARIABLE`: Update the world transformation variable of the provided nodes.
- `TC_UPDATE_VOLUME_VARIABLES`: Update the volume variables of the provided renderable objects.
- `TC_UPDATE_VOLUME_EFFECTS`: Update the volume effects of the provided renderable objects.
- `TC_UPDATE_GLOBAL_VARIABLES`: Update the global variables of the provided renderable objects.
- `TC_UPDATE_GLOBAL_EFFECTS`: Update the global effects of the provided renderable objects.
- `TC_UPDATE_STRUCTURAL_VARIABLES`: Update the structural variables of the provided renderable objects.
- `TC_UPDATE_STRUCTURAL_EFFECTS`: Update the structural effects of the provided renderable objects.
- `TC_ADD_RENDERABLES_DRAWGRAPH`: Add the provided renderable objects to the drawgraph.
- `TC_REMOVE_RENDERABLES_DRAWGRAPH`: Remove the provided renderable objects from the drawgraph.
- `TC_CHANGE_RENDERABLES_DRAWGRAPH`: Tell the drawgraph that the bounding volume or the transformation matrix of the provided renderable objects has changed.
- `TC_OPTIMIZE_DRAWGRAPH`: Tell the drawgraph that is can now optimize its internal state.

To generate a list of tasks for a sent event, the `ITaskGenerator`-interface is introduced where every implementation class represents one event ID. Every concrete class generates only the tasks which are necessary to process the event it represents. The generated task is only a class instance which holds a task code as listed above along with a list of nodes. When the task is executed the operations of the task are only applied to that list of nodes.

Every task code has a corresponding class instance which can only execute one specific task. These execution classes are all supporting the `IExecuter`-interface. Thus, when a specific task gets executed by the global scenegraph context, the latter is searching for a `IExecuter`-object which can handle the code of the task. Then the `Execute()`-method of the returned object is called. This method gets the list of nodes as parameter on which the executer should operate.

Both the task generator and the task executer concept are following the command design pattern as described in Section 2.6.

The *visitor pattern* implemented using *double dispatch* is also integrated into the scenegraph of YARE. The reason for this is that YARE should support already implemented

algorithms of other scenegraph frameworks, which are then easy to port to the YARE scenegraph. Also the task executer for updating the structural context of nodes is using the visitor pattern to collect the data from the scenegraph.

3.10.6 Retrieval of Renderable Objects

This section explains the steps taken to convert meshes, variables and effects to renderable objects. Renderable objects in this context are instances of the `Renderable`-class as introduced in Subsection 3.9.2.

1. First, for each geometry in a mesh object an instance of the `Renderable`-class is created.
2. Afterwards, for each `Renderable`-instance the following steps are performed.
   a) The renderable gets the corresponding geometry assigned.
   b) The bounding volume for the renderable is retrieved from the mesh object.
   c) The *structural context* for the renderable is retrieved from the scene database (including structural variables and effects) and assigned to the renderable.
   d) All global variables and effects are fetched from the scene database and assigned to the renderable.
   e) All volume variables and effects which volume the renderable intersects are fetched from the scene database and assigned to the renderable.
   f) If the renderable instance has been created during this update process, it is added to the drawgraph.

This process is illustrated in Figure 3.22 as well.

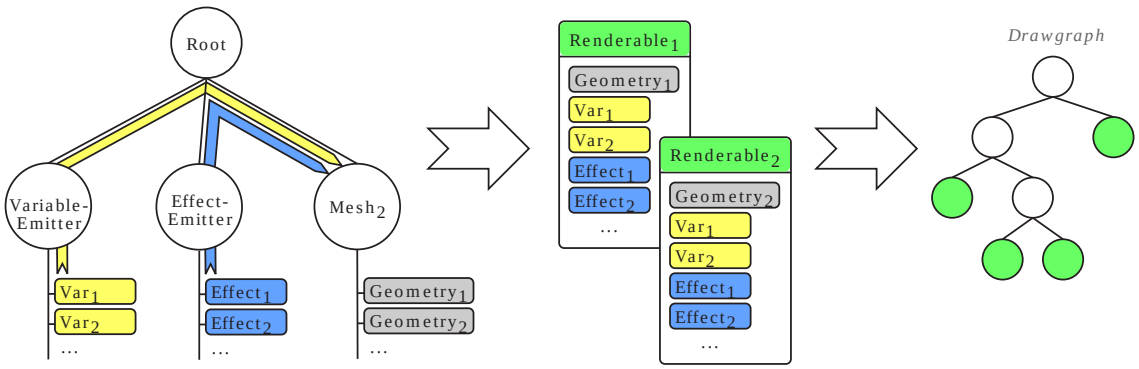

Figure 3.22: The renderable objects are constructed from the geometries, variables and effects of the scenegraph and are inserted into the drawgraph.

The concept of first retrieving and later assigning the structural context variables and effects to the renderables objects allows the user to specify different cameras for different meshes. This is done by creating the desired cameras and placing the meshes

which should use a specific camera for rendering to the right subgraph of the camera. The example in Figure 3.23 illustrates how the variables of the cameras are emitted to different meshes.

This concept also enables the user to apply the same effect to multiple geoemtries, but use the variables in the structural context to provide different configurations at each geometry. An example for such a configuration is the usage of a normal mapping effect for the whole scene, but the light sources in the structural context of a geometry emit the necessary light direction variables for the correct lighting of the object.

An important step in the process of generating the renderable objects is their *insertion into the drawgraph*. Using this concept, no additional synchronization step between the scenegraph and the drawgraph is necessary.

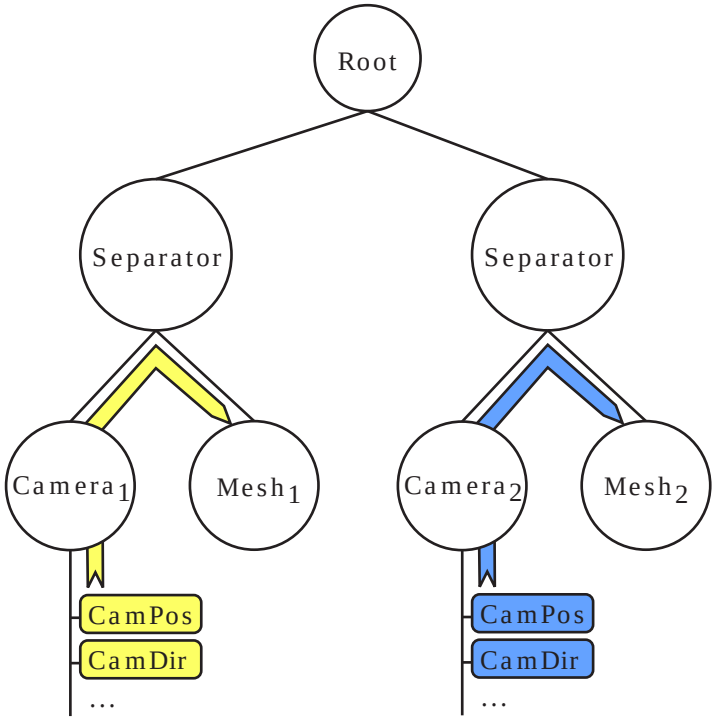

Figure 3.23: An example of a scenegraph with multiple cameras

3.10.7 Scenegraph Examples

Creating a Scenegraph by Source Code

The following example shows the source code to build the scenegraph of Figure 3.24.

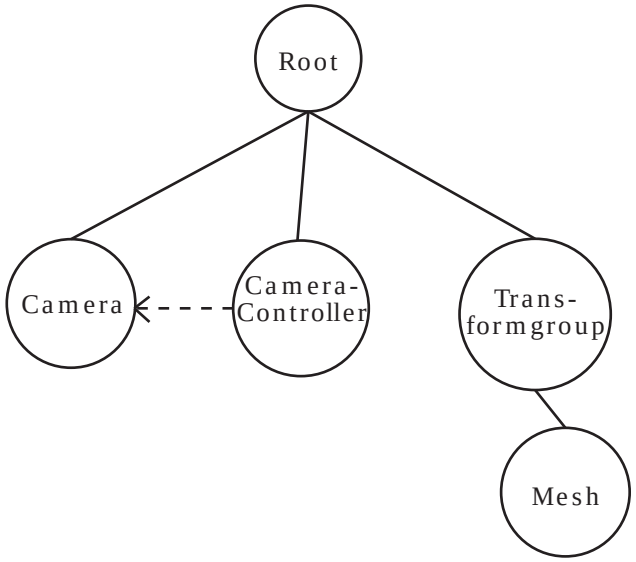

Figure 3.24: A simple example of a scenegraph

```
// Create the root node.
RootGroupPtr root = RootGroupPtr(new RootGroup());

// Create a camera and set the root as parent.
CameraPtr camera = CameraPtr(new Camera());
camera->SetParent(root);

// Create a controller which controlls the camera.
// The mouse and keyboard are used to control the position and
// orientation of the camera.
IControllerPtr camera_controller = FPSPtr(new FPS());
camera_controller->SetTarget(camera);
camera_controller->SetParent(root);

// Transform the mesh ten units in the negative Z direction.
GroupPtr transform_group = TransformGroupPtr(new TransformGroup());
Matrix4x4f trans(1);
trans.SetT(0, 0, -10);
transform_group->SetMatrix(trans);
transform_group->SetParent(root);

// Create a mesh node from a 3ds file.
MeshPtr mesh = MeshPtr(new FileMesh("sample.3ds"));
mesh->SetParent(transform_group);
```

Implementing Shadow Mapping

The scenegraph in the following example consists of:

- A camera and a controller for it
- A mesh containing four geometries to build a simple scene
- A spot light node which emits variables for solving the light equation at each fragment
- An effect applying a blinn material to the meshes as described in Subsection 3.8.10
- An effect which renders the depth values of the scene in a texture from the point of view of the spot light.
- An effect which uses the generated depth texture to implement shadow mapping and light the meshes with the spot light.

The scenegraph is illustrated in Figure 3.25 and the rendered image of this scene is shown in Figure 3.26.

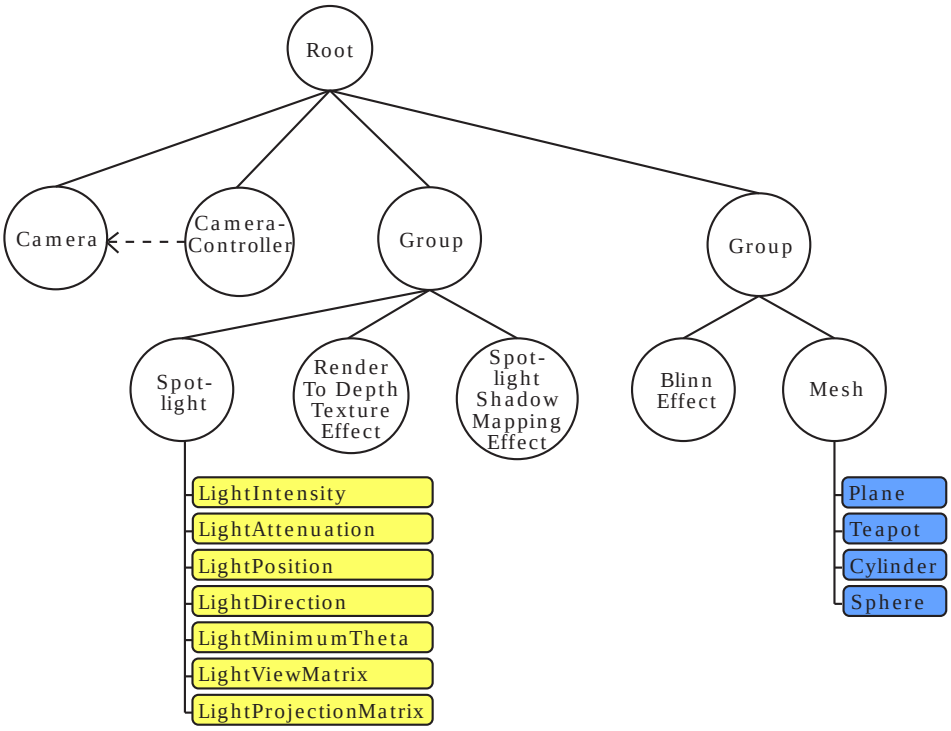

Figure 3.25: A scenegraph setup to implement *shadow mapping*

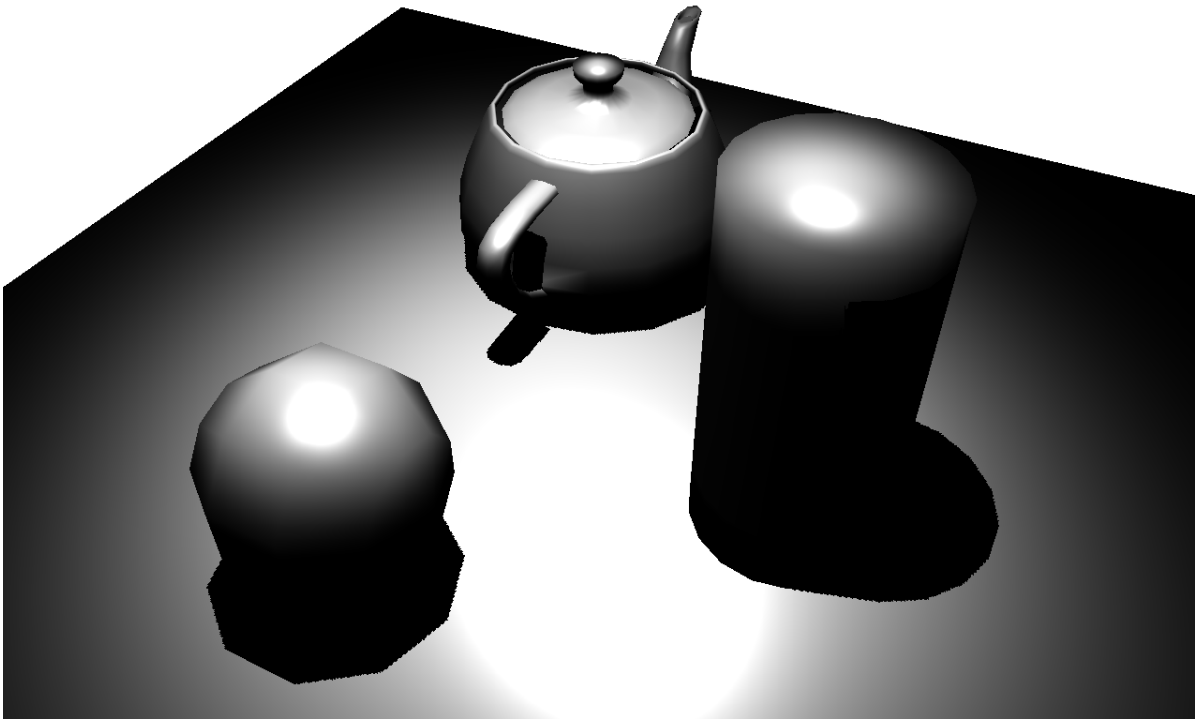

Figure 3.26: The rendered image of the scenegraph from Figure 3.25

4 Implementation Details

The following chapters give an overview of all implementation related topics, issues and their solution.

4.1 The Coding Environment

As programming language C++ and as development environment Microsoft Visual Studio ([MSVS]) was chosen. YARE can be built with version 2003 and 2005 of Visual Studio. The main reason to support version 2005 was the integration of *performance optimization tools* into the IDE of Microsoft. Since it is not sure if YARE should only support Microsoft Windows ([Windows]) as operating system (OS) in the future, the *Core* module was written as platform independent as possible.

All the modules described in Section 3.3 are packaged in dynamic link libraries (DLLs), since any application using YARE should be able to decide which modules it loads at startup. Specifically the rendering driver should be selectable by the user of the application.

An important implementation question was how to manage memory allocation and deallocation. Since deallocation is very error prone (can lead to memory leaks in the case of exceptions and coding errors made by developers) it was decided to use smart pointers with reference counting. A mature library providing such pointers is *boost* ([Boost]). It provides shared and intrusive pointers to minimize such memory deallocation problems.

Since in the near future more than one developer will use and extend YARE, an instrument was needed to keep a uniform style of the source code. Therefore, an own *Coding Guide* for YARE was introduced by the author of this thesis which can be downloaded at [YareCodingGuide].

Another introduced guideline used to reconstruct the source code changes is the *Message Header System* (MHS). Using the MHS every source code method has comment lines above it indicating the changes of the code. Every line of the MHS contains the following data: the date and time of the change, a letter token indicating what has been done (N = New, A = Added, C = Changed, P = Pre-Reviewed, R = Reviewed), the sign of the developer (usually the first letter of the forename and surname) and additional

comments for the change. The tokens P and R are used for communication between reviewer and developer during code reviews.

An example method following the MHS rules could look the following way:

```
//------------------------------------------------------------------------
//2007-01-15 12:51 N,BM: introduced because of....
//2007-01-16 12:51 A,BM: also calculating the vertex colors now
//2007-01-18 12:51 P,CT: add more return value checks
//2007-01-19 12:51 C,BM: done
//2007-01-20 12:51 R,CT:

void foo(const uint64 &param)
{
  ...
}
```

4.2 The OpenGL Driver

This chapter gives details on the OpenGL techniques/extensions used to implement the rendering interface presented in Section 3.7.

4.2.1 Resource Handling

Since most created OpenGL objects need to be released on shutdown, the concept of a resouce manager was introduced. All classes that encapsulate an OpenGL handle implement the `IOpenGLResource-` interface which only provides a `Shutdown()` method. In the implementation of this class method, the OpenGL resource must be released. When a resource object is created, the instance is automatically registered with the resource manager singleton class. And it is the task of this manager to call the `Shutdown()` method of all resources when the graphics system is shut down.

Placing the code which releases the OpenGL resource in a destructor of a class could lead to unfreed OpenGL resources. The reason for this is that YARE uses smart pointers to the instances and a developer does not exactly know when they are freed. Since an application using YARE can have references to `IOpenGLResource-` instances they will not get freed when the OpenGL driver of YARE is shut down.

4.2.2 Variables

Variables created by the variable factory of the OpenGL driver are supposed to be used for storing vertex attributes like position, normal und tangent vectors. Therefore, they should be stored in a performance optimized way to ensure fast rendering of the meshes. To ensure

this requirement, the OpenGL *vertex buffer object* (VBO) extension is used to directly store the data at the graphics device. See [OGLExt] for details on this extension. If the available graphics device does not support this extension, the data is kept in the main memory of the PC.

Since the two variable types share most of the source code, the *template method design pattern* is used: a new base class (`GPUVariableBase`) is introduced that in turn calls pure virtual methods which are implemented by the two concrete variable classes. See Section 2.6 for more information on the *template method design pattern*.

Both variable types use the *vertex array* extension for sending the drawing commands to the graphics device. If this extension is not available, OpenGL *immediate mode* rendering is used as a slow fallback method.

4.2.3 Rendertargets

To support *rendering to textures* and *multiple render targets* (MRTs), the frame buffer object (FBO) extension is used. If this extension is not available, the *Pbuffer* extension is used. Since FBOs and PBuffers have almost the same functionality, a common interface was introduced (`IHWBuffer`) which lets the render target manager deal with the two target classes in a uniform way.

4.3 Applied Design Patterns

The listing in this section gives an overview of the used design patters in YARE. With every listed design pattern one or more examples are given where this extension is used in YARE.

- Abstract factory pattern: Variable factories. See Subsection 3.6.2 for details.
- Prototype pattern: Cloning of volumes (Subsection 3.4.1), variables (Subsection 3.6.2) and render passes (Subsection 3.8.5).
- Singleton pattern: Applied to all global classes of which only one instance is allowed, e.g. the global scenegraph context and the mesh pool as described in Subsection 3.10.2, the code generator and pass converter classes as descibed in Section 3.8, the post-processing manager as described in Subsection 3.8.9, the rendertarget stack as described in Subsection 3.8.5 and many more.
- Adapter pattern: Wrapping different OpenGL implementations of the same functionality using one common C++ interface and an adapter class for every OpenGL technique. Wrapping some C++ STL containers with more convenient to use classes of the *Core* - module.
- Bridge pattern: The graphics API-independent rendering interface is one example where the *Bridge pattern* is used in YARE. In this context the rendering interface as described in Section 3.7 is the *implementor*, the OpenGL and the Direct3D rendering drivers are the *concrete implementors*, the `IPass`-interface (as described in Subsection 3.8.5) is

the *abstraction*, and render passes like the `AdditionalPass` and `PostProcessingPass` classes are the *refined abstraction*.

- Proxy pattern: A well-known example of the proxy pattern are the *reference counted pointers*. This technique is used by YARE through the shared- and intrusive pointers of the *boost* library. See Section 4.4 for details.
- Command pattern: Used by the synchronization code of the scenegraph with the scene database: the task generators and the task executers in Subsection 3.10.5 follow the *command pattern*.
- Observer pattern: Also used with the task generators of the scenegraph: the task generators are registered with the scenegraph context object which waits for an event to get fired. The context object then asks the registered task generators if they can handle the fired event.
- Template method pattern: Used with a lot of base classes in YARE. Base classes implement interfaces and are then further derived by concrete classes. Examples of base classes in YARE are `BaseEffect`, `BaseTechnique`, `BaseTechniquePart` and `BasePass`. A concrete class is e.g. the `BlinnEffect` class. The needed work which is common to all concrete classes is already done in the base classes, which also introduce pure virtual methods to implement the desired behavior.
- Visitor pattern: The visitor pattern is used to perform operations on the scenegraph of YARE. See Subsection 3.10.5 for details.

4.4 External Frameworks

Table 4.1 lists all external software packages used with YARE. These packages (except the *boost* library - for size reasons) are delivered together with the source code distribution of YARE. The reason for packaging external software together with the rendering engine is to avoid any version compatibility problems.

4.5 The Editor

For rapid prototyping of scenes and new effects, it is not comfortable having to write C++ source code and recompile an application. This was the reason to provide an editor to the main features of the rendering engine.

The following listing gives an overview of the main features of the YARE editor.

- Lets the user build a complete scenegraph using drag'n'drop techniques to instantiate the nodes.
- Change the properties of a node by selecting it and using the 'Properties' - window to adjust the values.

- Changing properties immediately updates the 'Output' - window showing the results of the change.
- Show/Hide the debug output of the drawgraph.
- Show/Hide the bounding volumes of the objects.
- Switch the bounding volume type between sphere and box.
- Import the complete scenegraph from an Inventor [OpInv] file.
- Cut/Copy/Paste operations for scenegraph nodes.
- Saving and loading created scenegraphs in/from an XML text file.

A screenshot of the editor can be seen in Figure 4.1.

Name	Usage
Boost [Boost]	Shared and intrusive pointers. Class methods callbacks. `Boost::Any` as clean `void*` alternative. String formating.
Cg [Cg]	High level shading language compiler.
Collada [Collada]	3D model file format importer.
Expat [Expat]	XML file parser.
FreeType [Freetype]	Font rendering engine.
Radiance RGBE [Radiance]	Radiance RGBE high dynamic range image format.
JasPer [JasPer]	Jpeg 2000 file format.
Jpg [Jpg]	Jpeg file format.
lib3ds [Lib3ds]	3ds file format.
Lua [Lua]	Lua scripting engine.
Luabind [Luabind]	Building the connection between C++ classes and Lua.
OpenEXR [OpenEXR]	OpenEXR high dynamic range image format.
libpng [Libpng]	Portable network graphics image format.
RPly [RPly]	PLY 3D model file format importer.
wxWidgets [WxWidgets]	Platform independent GUI construction library. Used for the implemented editor (see Section 4.5 for details).
zlib [Zlib]	ZIP file compression library.

Table 4.1: External software packages used by YARE

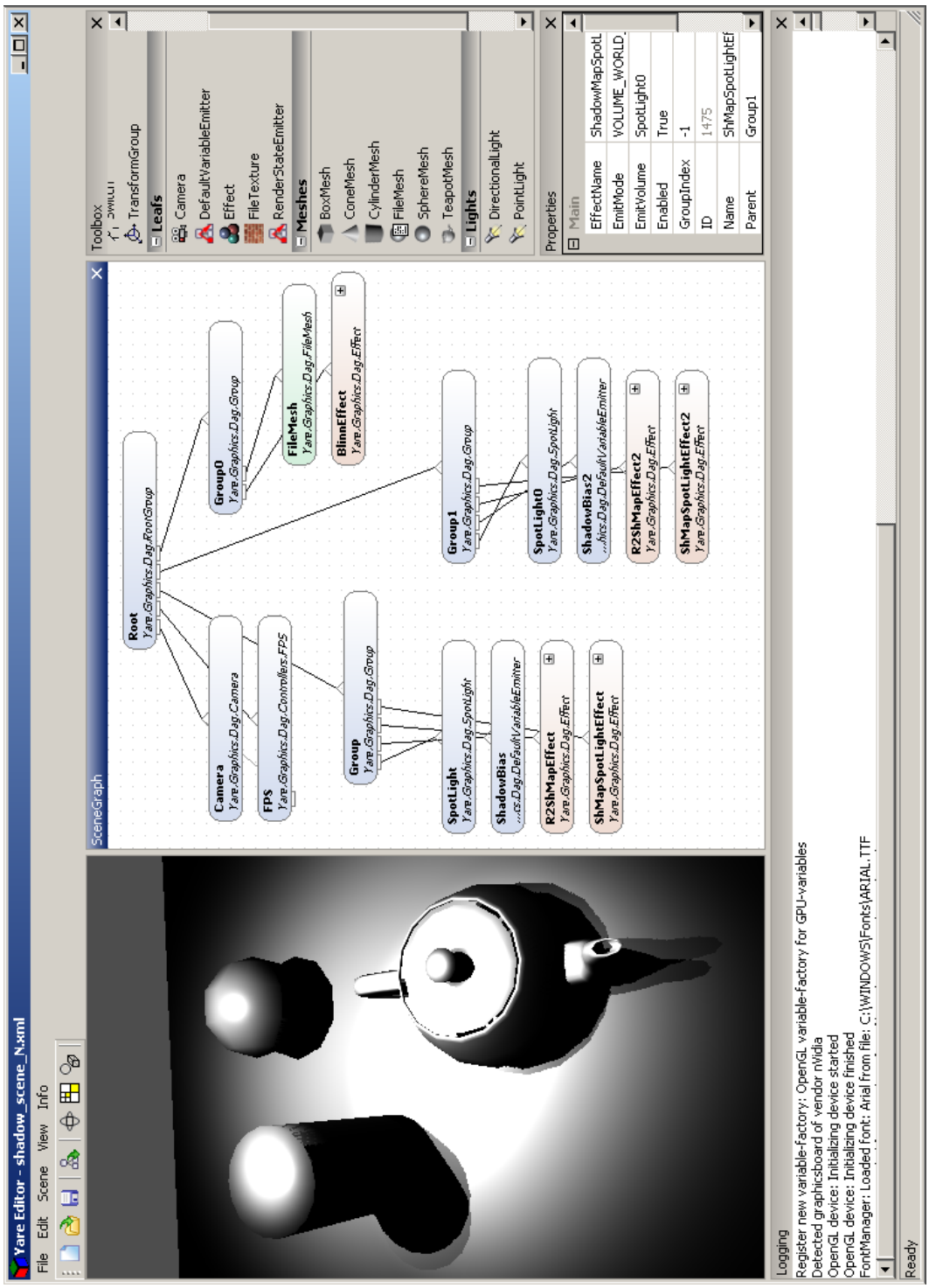

Figure 4.1: A screenshot of the editor featuring the output window, the scenegraph window, the toolbox and the properties window. The logging window is displayed at the bottom.

4.6 Feature Summary

This section gives a rough feature overview of YARE and references to sections were the interested reader can find more information about a specific feature.

- Graphics API independent.
  Details: Section 3.7.
- C++, 150.000 LoC, Visual Studio 2003 and 2005.
  Details: Section 4.1.
- Modulized architecture.
  Details: Section 3.3.
- Actor Concept, scriptable using Lua, C++ Reflection System.
  Details: Section 3.4.
- Graphics-API/-device abstraction.
  Details: Section 3.7.
- Effect framework: Automatic generation of shadercode from user-selectable effects.
  Details: Section 3.8.
- Automatic handling of multipassing.
  Details: Subsection 3.8.6.
- Easy integration of new effects.
  Details: Subsection 3.8.11.
- Complete separation of data and algorithms.
- Inventor-like scenegraph with novel scene database synchronization concept.
  Details: Section 3.10.
- Efficient culling through drawgraph (Quadtree, Octree, kd-tree).
  Details: Section 3.9.
- Implemented common effects: shadowmapping, normal-, parallax-, relief mapping, different kinds of lightsources, post-processing framework.
  Details: Section 3.8.
- 2D canvas including style-based widgets (Win32,WinXP,MacOS).
- Support for Cg, CgFX, GLSL, HLSL.
  Details: Subsection 3.7.2.
- Asset preparation pipeline.
- Supported image formats: BMP, DDS, GIF, HDR, JPG, JPG2000, OpenEXR, PCX, PNG, TGA.
- Supported model formats: 3DS, Collada, Inventor, OBJ, PLY, TRI.
- WYSIWYG-Editor. Loading scenegraphs from Inventor-files.
  Details: Section 4.5.

5 Evaluation

5.1 Fulfilling the Requirements

This section checks if all requirements listed in Section 3.2 are fulfilled. For this purpose the requirements are listed here again along with a short description how they are fulfilled.

- **Performance optimized data storage**: Implemented in the OpenGL driver (see Section 4.2 for details). It uses various OpenGL extensions (e.g. VBOs and vertex arrays) to provide the most efficient data storage on the graphics device available.
- **Effect framework**: The implemented effect framework presented in Section 3.8 allows the user to combine effects in a flexible manner. This includes global emitted effects, handling of multipassing, shadowing and performance optimizations. The flexibility of the effect framework is based on various facts:
  - The framework is designed from ground up to support effect extensions. This means that there are well defined paths how this can be achieved. No *hacking* of any form is necessary.
  - A new effect can be integrated into the system without having to handle the relations to other effects - they are combined automatically.
  - The extensions of the framework can be done at different levels. The easiest way is to provide a new implementation of a Cg interface, followed by the introduction of a new Cg interface. If that is not flexible enough the Cg framework program can be changed. Finally, new additional rendering passes can be introduced if a more global manipulation of the rendering process is needed.
- **Vertex and fragment programs**: A list of supported shader profiles and languages can be found in Subsection 3.7.2.
- **Post-processing effect framework**: The implemented post-processing framework is presented in detail in Subsection 3.8.9.
- **Multiple rendertarget support**: Rendering to multiple rendertargets (MRTs) is supported by the rendering interface as presented in Section 3.7.
- **Model importers**: A list of the supported model formats can be found in Section 4.6.
- **Image format support**: A list of the supported image formats can be found in Section 4.6.
- **No hardcoding of shadow effects**: As presented in Subsection 3.8.10 the shadow mapping effect is split into two ordinary rendering effects. The first effect renders the scene into a depth texture and the second effect (integrated as a spot light effect) uses

that texture to shadow the scene. Therefore, no hardcoded paths or any *cast-/receive-shadowflags* are introduced to the source code, which makes it easy to either enhance the implemented shadow mapping or introduce even a new shadowing technique like shadow volumes.

- **Local and global multipassing**: The rendering engine handles multipassing automatically as described in detail in Subsection 3.8.6.
- **Global algorithms**: The engine provides multiple entry points to change the global behavior of rendering. E.g. using an own implementation as drawgraph it is possible to integrate advanced culling algorithms in a hierarchical way. Additionally, the scenegraph processing can be extended easily by using the event-driven update mechanism presented in Subsection 3.10.5. Using new events and tasks, any operation can be performed on the nodes of the scenegraph.
- **Documentation and Examples**: The rendering engine comes with a fully documented source code (using *doxygen* [Doxygen] style), design documents, class diagrams, screenshots and even this thesis. Supplementary test cases are provided, which also show how to use the features of the engine. The test cases cover the following topics: engine initialization, 2D canvas rendering, rendering to textures, using vertex- and fragment programs, handling user input, creation of geometry with various vertex attributes, high dynamic range rendering, using occlusion queries, streaming of variables, using multiple render targets, rendering to variables, using multithreaded rendering, loading and displaying models, using the 2D GUI widgets and a lot more.
- **Scenegraph algorithms**: The scenegraph nodes can either be accessed using classical graph traversals or the scene database as presented in Subsection 3.10.4. By deriving node classes, new functionality can be integrated into the scenegraph.
- **State management**: The graphics API renderstates can be set using the `IState`-interface of the rendering interface presented in Section 3.7.
- **Scripting**: The Lua scripting language is integrated into the engine, providing access to all actor classes as described in Subsection 3.4.6.
- **API independent**: Since the access to a specific graphics API is only allowed by the implementation of the rendering interface, it is possible to support any graphics API available.
- **Occlusion queries**: Occlusion queries are supported by the `IOcclusion`-interface provided by the rendering engine. The rendering API driver implements that interface using graphics hardware features.
- **Editor**: An overview of the implemented editor is provided in Section 4.5.

As the list above shows, all requirements are fulfilled.

5.2 Software Engineering

Since the implemented rendering engine encloses a lot of different classes and algorithms, it is important to use common software engineering techniques to keep the system understandable, flexible and extensible. The implemented techniques are as follows:

- *UML-diagrams* are used for the global class layout.
- Software design patterns as described in Section 2.6 and Section 4.3
- Design and concept documents are provided
- Using *doxygen*-based comments, a full source code documentation can be generated.
- The MHS (as described in Section 4.1) is used to reconstruct source code changes.
- The engine uses a module approach with several layers as described in Section 3.3 to keep a clean and understandable layout of the source code.
- Coding guidelines are introduced as presented in Section 4.1.

6 Summary and Future Work

In this thesis, the rendering pipeline as an global overview concept of all graphics applications is presented. The author also explains how graphics devices can be programmed to increase flexibility. For this purpose vertex- and fragment shaders are used, which are an important topic throughout the whole thesis. For a better understanding of the layout of the implemented software framework, comparable frameworks along with their common features, structures and algorithms were presented.

The main part of this thesis was to describe the concepts and design decisions behind the implemented rendering engine. An important module of the graphics framework is the described rendering interface which wraps the underlying graphics APIs. Apart from the main layout and modules of the engine, the core of this thesis was the presentation of the effect framework. This framework gains its flexibility through splitting complicated rendering effects into simple and reusable parts. A user can then take such simple parts and combine them to a new rendering effect. The underlying system generates a vertex- and fragment program out of these parts and takes care of rendertargets and the correct order of the rendering passes. While designing the class layout, it turned out that the clean integration and automatic combination of available effects were the most difficult features to achieve. Even so, an implementation which fulfilled these requirements was presented. The next step was to provide a scenegraph which allows the developer high-level usage of the underlying effect framework and the drawgraph.

In Chapter 4, details on the coding environment and the OpenGL implementation of the rendering interface were given. To allow instant testing and usage of the implemented features, an editor application was developed.

This thesis was concluded with the confrontation of the implemented rendering engine with the requirements defined at the start of the work. It turned out that all demands were successfully met.

6.1 Future Work

As with every software project several possible improvements exist:

- The shader code generated by the effect framework can be optimized. Duplicate source code of different implementations of Cg-interfaces should be detected and avoided.
- The effect *level of detail* system as described in Subsection 3.8.7 introduces *popping artifacts* which could be reduced by the *unpopping technique* presented by Giegl and Wimmer [GieglWimmer07].

- The editor application can be extended to expose more features of YARE. E.g. a window for shader code editing and scripting would increase its usability.

A C++ Interfaces

A.1 Interfaces of the Variables Concept

```
/**
* The interface for all variables.
*/
class YARE_GRAPHICS_API IVariable : public Yare::Core::Reflection::IUnknown
{
public:

  /// An enumeration of possible lock flags, which are meant to give hints for
  /// performance optimizations.
  enum LockFlag
  {
    LOCK_READ_ONLY, ///< The data is locked for read only operations.
    LOCK_WRITE_ONLY, ///< The data is locked for write only operations.
    LOCK_READ_WRITE ///< The data is locked for read and write operations.
  };

  /// This clones the variable, and either returns a full deep copy of
  /// all data, or just a reference to the same data, in order to allow
  /// data sharing. This decision is made by checking the 'shareable' flag
  /// of the description.
  /// @return The pointer to the cloned variable.
  virtual IVariablePtr Clone() = 0;

  /// Retrieves a reference to the description of this variable.
  /// @return The variable description of this variable.
  virtual VariableDesc &GetDesc() = 0;

  /// Copies 'count' elements of this variable to a destination address.
  /// @param dest The destination pointer to copy the data to.
  /// @param count The number of variable-elements to copy.
  /// @return The number of elements copied.
  virtual uint32 CopyTo( void *dest, uint32 count ) = 0;

  /// Copies a series of variable-elements from a raw memory
  /// pointer into this variable.
```

```
  /// @param source The raw pointer to copy from.
  /// @param count The number of variable-elements to copy.
  /// @return The number of elements copied.
  virtual uint32 CopyFrom( void *source, uint32 count ) = 0;

  /// Returns a pointer to the internal data representation.
  /// After that call, the variable is locked and no other methods
  /// of this variable can be called until Unlock().
  /// @param flag Defines the type of lock to perform.
  /// @return The pointer to the data or NULL if not available.
  virtual void *Lock( const LockFlag &flag ) = 0;

  /// Unlocks this variable. The changes to the data will take effect.
  virtual void Unlock() = 0;

  /// Serializes this variable into/from a data chunk and the description
  /// of the variable into/from the attributes of the chunk.
  /// @param chunk The data chunk to serialize from/into.
  /// @param write True if the variable is written, false if its read.
  virtual void Serialize( Yare::Core::Data::ChunkPtr &chunk,
                          const bool &write = false ) = 0;

  /// Returns an individual variable-element, wrapped into
  /// a boost::any variant. Hint: This call can be very slow
  /// if the variable is e.g. a GPU-variable.
  /// @param index The index of the variable-element to return.
  /// @return A copy of the index-th element of the variable.
  virtual boost::any GetElement( uint32 index ) = 0;

  /// Sets an individual variable element, wrapped in a boost::any variant.
  /// This call can be very slow if the variable is e.g. a GPU-variable.
  /// @param index The index of the variable-element to set.
  /// @param value The wrapped value to set.
  virtual void SetElement( uint32 index, const boost::any &value ) = 0;

  /// Initializes the variable with the settings of the VariableDesc.
  /// @param desc The description for this variable.
  virtual void Initialize( const VariableDesc &desc ) = 0;
};


/**
* A description for a variable.
*/
struct YARE_GRAPHICS_API VariableDesc
{
  /// Indicates the frequency of the variable.
```

```
enum VariableFreq
{
  FREQ_VERTEX, ///< A variable that contains per vertex data.
  FREQ_PRIMITIVE, ///< A variable that constains per primitive data.
  FREQ_PRIMITIVE_GROUP, ///< A variable that contains per primitive
                        /// group data, e.g. a material.
  FREQ_GEOMETRY, ///< A variable that containts data which is used for
                 /// a whole geometry.
  FREQ_GLOBAL ///< A variable that contains global data, e.g. configuration
              /// settings of a rendercycle, e.g. Transformation matrices.
};

/// The default constructor.
VariableDesc();

/// Returns the fullname of this variable description.
/// This fullname is build from the name and the index.
/// @return The full name of this variable description.
std::string GetFullname() const;

/// Serializes this variable description into/from the attributes
/// of a data chunk.
/// @param chunk The data chunk to serialize from/into.
/// @param write True if we want to write the variable description,
///              false if we want to read it.
void Serialize(Yare::Core::Data::ChunkPtr &chunk,
               const bool &write = false);

/// Compares two variable descriptions.
/// @param v The second variable description to compare this variable
///          description with.
/// @return True if the two variable descriptions are equal,
///         false otherwise.
inline bool operator == ( const VariableDesc &v ) const;

/// Compares two variable descriptions.
/// @param v The second variable description to compare this variable
///          description with.
/// @return True if the two variable descriptions are not equal,
///         false otherwise.
inline bool operator != ( const VariableDesc & v ) const;

/// Compares two variable descriptions.
/// @param v The second variable description to compare this variable
///          description with.
/// @return True if the two variable descriptions have the same name,
```

```
  ///         type and frequency, false otherwise.
  inline bool operator ^= ( const VariableDesc & v ) const;

  uint32       count; ///< Indicates how many elements are
                      ///  in the corresponding variable.
  std::string  name;  ///< The name of the corresponding variable.
  uint32       index; ///< The usage-index of the corresponding variable,
                      /// e.g. the texturestage. Can be dynamically assigned.
  std::string  type;  ///< The cpp-name of the type of the elements of the
                      /// corresponding variable.
  VariableFreq freq;  ///< Indicates the frequency of the
                      /// corresponding variable.
  bool         dynamic; ///< Indicates whether the data of the corresponding
                        /// variable will be respecified repeatedly.
  bool         shareable; ///< Indicates whether the data of the
                          /// corresponding variable can be shared among
                          /// other variables or not.
  bool         geometryrelated; ///< Indicates whether the data of the
                               /// corresponding variable is related to
                               /// geometry (e.g. vertexnormals)
                               /// or not (e.g. CameraDirection).
  VariableAdditional additional; ///< Stores some additional information
                                 /// about the variable, e.g. if the data
                                 /// of the corresponding variable is
                                 /// stored on the GPU.
};


/**
* The interface for objects which emit variables.
* E.g. lights emit variables of the type LightDirection, LightPosition etc.
*/
class YARE_GRAPHICS_API IVariableEmitter :
                        public Yare::Core::Reflection::IUnknown
{
public:

  // The mode an object emits variables.
  enum EmitMode
  {
    EM_GLOBAL, ///< The object emits its variables to the whole 'universe'.
    EM_STRUCTURAL, ///< The object emits its variables only to other
                   /// objects which lie in the same structur.
                   /// E.g. in the same scenegraph-group.
    EM_VOLUME_OBJECT_SPACE, ///< The object emits its variables only to
                            /// objects which lie in the emit-volume.
                            /// The volume is in object space of the emitter.
```

```
    EM_VOLUME_WORLD_SPACE ///< The object emits its variables only
                          /// to objects which lie in the emit-volume.
                          /// The volume is in world space.
  };

  /// Returns the mode this object emits variables.
  /// @return The emit mode.
  virtual EmitMode GetEmitMode() = 0;

  /// Returns the range this object emits variables in.
  /// @return The volume defining the emit-range.
  virtual Yare::Core::Math::IVolumePtr GetEmitVolume() = 0;

  /// Returns the number of variable groups of this emitter.
  /// @return The number of variable groups of this emitter.
  virtual const uint32 &GetGroupCount() const = 0;

  /// Fills the provided variableset with the emitted
  /// variables of this object. Hint: The variablelist is
  /// not cleared when passing into this method.
  /// @param variables The variablelist to fill the emitted variables in.
  virtual void GetVariables( const uint32 &groupIndex,
                             std::vector<IVariable*> &variables ) = 0;

  /// Returns the name of the emitter, e.g. "Directional Light"
  /// or similar names.
  /// @return The name of the emitter.
  virtual const std::string &GetName() const = 0;
};
```

A.2 Main Rendering Interfaces

```
/** The main 3D rendering device interface.
*   There can be concrete implementations for Direct3D, OpenGL and similar.
*/
class IDevice : public Yare::Core::Reflection::IUnknown
{
public:
  /// This devices capabilities.
  /// @return The capabilites of the device.
  virtual DeviceCaps GetCaps() = 0;

  /// The device state.
  /// @return The state of the device.
  virtual IStatePtr GetState() = 0;

  /// The framebuffer of this device, used for presentation to the screen.
  /// @return The framebuffer of the device.
  virtual ITargetPtr GetFramebuffer() = 0;

  /// The shader-code compiler for this device.
  /// @return The compiler of this device.
  virtual ICompilerPtr GetCompiler() = 0;

  /// A helper object providing occlusion culling services.
  /// @return A helper object providing occlusion culling services.
  virtual IOcclusionPtr GetOcclusion() = 0;

  /// Allows to set render targets.
  /// @return The manager of the rendertargets.
  virtual ITargetManagerPtr GetTargetManager() = 0;

  /// Returns a reference to the binding manager.
  /// @return A reference to the binding manager.
  virtual IBindingManagerPtr GetBindingManager() = 0;

  /// Returns a reference to the window manager.
  /// @return A reference to the window manager.
  virtual IWindowManagerPtr GetWindowManager() = 0;

  /// Initializes the rendering device, with the given window handle.
  /// @param window The window to render to.
  /// @param initstruct Structure containing infos for
  ///                   the initialization of the device.
  virtual void Initialize( Yare::Core::Engine::WindowPtr window,
                           DeviceInit initstruct) = 0;
```

```
  /// Shuts down the rendering device.
  virtual void Shutdown() = 0;

  /// Starts a rendering frame.
  virtual void Begin() = 0;

  /// Ends a rendering frame.
  virtual void End() = 0;

  /// Flushes the device, and presents the framebuffer to the screen.
  virtual void Present() = 0;

  /// Clears the currently selected render targets.
  /// @param mask A combination of CLEAR_COLOR, CLEAR_DEPTH and CLEAR_STENCIL
  /// indicating which buffers should be cleared.
  virtual void Clear( const ClearMask &mask ) = 0;

  /// Creates a geometry buffer.
  /// @param target True if we want to use it as a render target too.
  /// @return A new geometry buffer or NULL if something went wrong.
  virtual IGeometryPtr CreateGeometry( bool target ) = 0;

  /// Creates a texture surface for rendering.
  /// @return A new texture surface or NULL if something went wrong.
  virtual ITexturePtr CreateTexture(
    TextureType type, ///< The type of texture surface to create.
    float quality, ///< The quality of the surface storage.
                   /// Can go from 0.0 to 1.0 and is clamped.
    int16 capabilities, ///< The capabilities the texture should have.
    int width, ///< The width in pixels of the surface.
    int height, ///< The height in pixels of the surface.
    TextureSampler samplerType, ///< The sampler type of the texture.
    int layers = 0 ) = 0; ///< The number of layers in the texture.
                          /// Only needed for 3D Textures.

  /// Creates a sampler for texturing.
  /// @return A new sampler or NULL if something went wrong.
  virtual ISamplerPtr CreateSampler() = 0;

  /// Returns the vendor of this graphics device.
  /// @return The vendor of this graphics device.
  virtual DeviceVendor GetDeviceVendor() = 0;
};


/** The base interface for all render targets.
```

```
*/
class ITarget : public Yare::Core::Reflection::IUnknown
{
public:
  /// The size of the target in pixels.
  /// @return The size of the target in pixels.
  virtual Vec2i GetSize() = 0;

  /// Returns a pointer to the viewport of this target.
  /// @return The pointer to the viewport of this target.
  virtual IViewportPtr GetViewport() = 0;

  /// Returns the main 2D rendering canvas that allows 2D operations
  /// to be performed on the 3D device. This can be used for HUDs,
  /// GUIs and other 2D related operations.
  /// It always operates on the current render target.
  /// @return The pointer to the canvas of this target or
  ///         NULL if the target has no canvas.
  virtual ICanvasPtr GetCanvas() = 0;

  /// Tells the system that the content of this rendertarget is not needed
  /// anymore. So the underlying resources can be reused by the system.
  virtual void Done() = 0;

  /// Returns the ID of this target.
  /// @return The ID of this target.
  virtual const Yare::Core::Engine::Identifier &GetId() = 0;
};


/** The interface of a manager for render targets.
*/
class ITargetManager : public Yare::Core::Reflection::IUnknown
{
public:
  /// Sets the active render target for a given buffer index.
  /// Setting target to NULL disables the given buffer index.
  /// @param bufferIndex The index of the buffer to bind the rendertarget to.
  /// @param target The new rendertarget.
  virtual void SetTarget(const uint8 &bufferIndex, ITargetPtr target) = 0;

  /// Allows to get the active render target.
  /// @param bufferIndex The index of the buffer to get the rendertarget from.
  /// @return The pointer to the active rendertarget.
  virtual ITargetPtr GetTarget(const uint8 &bufferIndex) = 0;

  /// Sets a render target as the only one.
```

```
  /// This binds the target to the buffer with index 0 and
  /// disables all other buffers.
  /// @param target The new rendertarget.
  virtual void SetSingleTarget(ITargetPtr target) = 0;

  /// This tells the targetmanager that setting the rendertargets is finished.
  /// Now the targetmanager ensures that the buffers are ready for rendering.
  virtual void Done() = 0;

  /// Resets the target manager.
  /// This sets the framebuffer as the only render target.
  virtual void Reset() = 0;

  /// The maximum number of simultaneous buffers supported.
  /// @return The maximum number of simultaneous buffers supported.
  virtual uint8 GetMaxBufferCount() = 0;
};


/** Texture sampler type.
*/
enum TextureSampler
{
  SAMPLER_1D, ///< A 1D sampler.
  SAMPLER_2D, ///< A 2D sampler.
  SAMPLER_3D, ///< A 3D sampler.
  SAMPLER_CUBE, ///< A cube map sampler.
  SAMPLER_RECT, ///< A texture rectangle sampler. Usually needed with
                ///  floating point textures on older nVidia hardware.
};

/** Texture filter modes.
*/
enum TextureFilter
{
  FILTER_NONE, ///< No texture filtering.
  FILTER_BILINEAR, ///< Bilinear texture filtering.
  FILTER_TRILINEAR, ///< Trilinear texture filtering.
};

/** Texture addressing modes.
*/
enum TextureAddressing
{
  ADDR_WRAP, ///< Wrap the texture. This is the default setting.
  ADDR_MIRROR, ///< Mirror the texture.
  ADDR_CLAMP, ///< Clamp the texture.
```

```
  ADDR_BORDER, ///< Set to border color.
  ADDR_MIRROR_ONCE, ///< Mirror just once.
};

/** Texture surface types.
*/
enum TextureType
{
  TYPE_INT, ///< A normal image texture map, using integer-channels.
  TYPE_FLOAT, ///< The same as above, but with floating point precision.
  TYPE_DEPTH, ///< A depth map. Usually storing 1 float per pixel.
  TYPE_SHADOWMAP, ///< A shadow map. Same as TYPE_DEPTH
                  /// but the used textureunit will be prepared
                  /// for projective texturemapping.
  TYPE_DISPLACEMENT ///< A texture that is used in a vertex shader.
};

/** Names for the faces of a cubemap.
*/
enum CubemapFaces
{
  FACE_POS_X, ///< Positive X-direction.
  FACE_NEG_X, ///< Negative X-direction.
  FACE_POS_Y, ///< Positive Y-direction.
  FACE_NEG_Y, ///< Negative Y-direction.
  FACE_POS_Z, ///< Positive Z-direction.
  FACE_NEG_Z, ///< Negative Z-direction.
};

/** Capabilities of a texture. Usually combined into a int16 value.
*/
enum Capability
{
  CAP_NONE = 0, ///< The texture has no special capabilities.
  CAP_ALPHACHANNEL = 1, ///< Alpha channel is provided by this texture.
  CAP_RENDERTARGET = 2, ///< The texture is a rendertarget as well.
  CAP_DYNAMICUPDATE = 4, ///< The content of this texture
                         /// can be updated dynamically.
  CAP_MIPMAPPING = 8, ///< This texture supports mipmapping.
  CAP_MOST_COMPATIBLE = 16, ///< The texture is created to be most
                            ///  compatible to all graphics devices.
};


/** The base interface for all texture samplers.
*/
class ISampler : public Yare::Core::Reflection::IUnknown
```

```
{
public:
  /// Returns the filter method for the texture.
  /// Default value is FILTER_BILINEAR.
  /// @return The filter method for the texture.
  virtual TextureFilter GetFilter() = 0;

  /// Sets the filter method for the texture.
  /// @param filter The filter method.
  virtual void SetFilter(const TextureFilter &filter) = 0;

  /// Sets the anisotropy for the texture. Set to 1 to disable.
  /// Default value is 1.
  /// @param anisotropy The anisotropy of the texture.
  virtual void SetAnisotropy(float anisotropy) = 0;

  /// Returns the anisotropy of the texture.
  /// @return The anisotropy of the texture.
  virtual float GetAnisotropy() = 0;

  /// Returns true if mipmapping is used for this sampler, false otherwise.
  /// Default value is false.
  /// @return True if mipmapping is used for this sampler, false otherwise.
  virtual bool GetMipmap() = 0;

  /// Enables or disables mipmapping.
  /// @param mipmap Use true to enable mipmapping, false otherwise.
  virtual void SetMipmap(const bool &mipmap) = 0;

  /// Returns the U texture addressing mode. Default value is ADDR_WRAP.
  /// @return The U texture addressing mode.
  virtual TextureAddressing GetUAddressing() = 0;

  /// Sets the U texture addressing mode.
  /// @param addressMode The addressing mode to use.
  virtual void SetUAddressing(const TextureAddressing &addressMode) = 0;

  /// Returns the V texture addressing mode. Default value is ADDR_WRAP.
  /// @return The V texture addressing mode.
  virtual TextureAddressing GetVAddressing() = 0;

  /// Sets the V texture addressing mode.
  /// @param addressMode The addressing mode to use.
  virtual void SetVAddressing(const TextureAddressing &addressMode) = 0;

  /// Returns the W texture addressing mode. Default value is ADDR_WRAP.
```

```
  /// @return The W texture addressing mode.
  virtual TextureAddressing GetWAddressing() = 0;

  /// Sets the W texture addressing mode.
  /// @param addressMode The addressing mode to use.
  virtual void SetWAddressing(const TextureAddressing &addressMode) = 0;

  /// Returns the border color used if the Border addressing mode is
  /// used for U,V or W coords. Default value is (0,0,0,0).
  /// @return The border color.
  virtual Color4f GetBorder() = 0;

  /// Sets the border color.
  /// @param color The color to use.
  virtual void SetBorder(const Color4f &color) = 0;

  /// Returns the texture associated with this sampler.
  /// @return The texture associated with this sampler.
  virtual ITexturePtr GetTexture() = 0;

  /// Sets the texture for this sampler.
  /// @param texture The texture to associate with this sampler.
  virtual void SetTexture(ITexturePtr texture) = 0;

  /// Binds a texture to a given texturestage index. Filter options are
  /// taken from the sampler. This method is only needed when using the
  /// fixed function pipeline. When using shaders, the binding of
  /// textures is done by the sampler variable of the fragment program.
  /// @param texture The texture to bind.
  /// @param textureStageIndex The index of the texture stage
  ///                          to bind the texture to.
  virtual void Bind(const ITexturePtr &texture,
                    const uint16 &textureStageIndex) = 0;

  /// Unbinds a texture from a given texture stage.
  /// @param texture The texture to unbind.
  /// @param textureStageIndex The index of the texture stage
  ///                          to unbind the texture from.
  virtual void UnBind(const ITexturePtr &texture,
                      const uint16 &textureStageIndex) = 0;
};


/** The base interface for texture surfaces.
*/
class ITexture : public ITarget
{
```

```
public:
  /// The type of sampler this texture is compatible with.
  /// @return The type of sampler this texture is compatible with.
  virtual TextureSampler GetSampler() = 0;

  /// The type of this texture.
  /// @return The type of this texture.
  virtual const TextureType &GetTextureType() = 0;

  /// Gets the capabilities of this texture.
  /// This is a combination of Capability-enum values.
  /// @return The capabilities of this texture.
  virtual uint16 GetCapability() = 0;

  /// Sets the face that is selected for rendering to
  /// when used as a render target and the texture is
  /// a cube map. Otherwise this call has no effect.
  /// @param face The active face for the cubemap to render to.
  virtual void SetFace(CubemapFaces face) = 0;

  /// Uploads the given surfaces into the texture.
  /// For cubemaps the order of the layers for the faces is:
  /// Pos_X, Neg_X, Pos_Y, Neg_Y, Pos_Z, Neg_Z
  /// @param start The start layer to start uploading from.
  /// @param layers The layers to upload into the texture.
  virtual void Upload(const uint32 &start,
                const std::vector< ISurfacePtr > &layers) = 0;

  /// Uploads the given surfaces as mipmaps into the texture (and
  /// overwrites automatically generated mipmaps).
  /// The outer index into the vector is the mipmap level.
  /// The inner index is the layer.
  /// @param mipmaps The mipmaps to upload.
  virtual void UploadMipMap(
    const std::vector< std::vector< ISurfacePtr > > &mipmaps) = 0;
};


/** Sorting order for geometry.
*/
enum SortOrder
{
  SORT_NONE, ///< No special sorting method.
  SORT_FRONT_TO_BACK, ///< Front to back sorting.
  SORT_BACK_TO_FRONT, ///< Back to front sorting.
  SORT_BACK_TO_FRONT_FOR_TRANSPARENCY ///< Back to front sorting with
                                      /// optimizations for transparency.
```

```
};

/** Geometry buffer optimization modes.
*/
enum OptimizeFlag
{
  /// Binds the current device state to the geometry buffer,
  /// thus all further draw calls with that state and this
  /// geometry buffer should be optimized.
  OPTIMIZE_STATE_BIND,
  /// Binds the currently activated shader programs to the geometry buffer,
  /// thus all further draw calls with these shaders and this geometry
  /// buffer should be optimized.
  OPTIMIZE_PROGRAM_BIND,
  /// Optimizes vertex cache locality for the current hardware.
  OPTIMIZE_VERTEX_CACHE,
  /// Optimizes all per geometry variables, so that they can
  /// be used as per vertex variable.
  OPTIMIZE_FREQUENCY
};

/** The rendermode a IGeometry uses for displaying the data.
This flag provides performance feedback from the graphics driver to the user.
*/
enum RenderMode
{
  /// The rendermode of a geometry is unknown if no renderable
  /// data has been added to it.
  RENDERMODE_UNKNOWN,
  /// The geometry is rendered using a lot of graphics commands
  /// which also provide the data. Immediate mode rendering is more
  /// flexible than retained mode, but not that fast.
  RENDERMODE_IMMEDIATE,
  /// The geometry is rendered using a few graphics commands which
  /// only the trigger the rendering of data which are already
  /// stored on the graphics device.
  RENDERMODE_RETAINED
};

/** Interface for geometry buffers for rendering.
*/
class IGeometry : public Yare::Core::Engine::Actor
{
public:
  /// Adds a given variable to the geometry buffer.
```

```
  /// @param variable The variable to add.
  virtual void Add( const Variables::IVariablePtr &variable ) = 0;

  /// Adds all variables of a given variableset to the geometry buffer.
  /// @param variables The variables to add.
  virtual void Add( const Variables::VariableSetPtr &variables ) = 0;

  /// Adds all variables of a given variablelist to the geometry buffer.
  /// @param variables The variables to add.
  virtual void Add( const Variables::IVariableList &variables ) = 0;

  /// Returns the list of add variables.
  /// @return The list of add variables.
  virtual const Variables::IVariableList &GetVariables() const = 0;

  /// Draws all primitives from this geometry buffer.
  virtual void Draw() = 0;

  /// Sorts the contained polygons according to the specified parameters.
  /// @param order The sorting order to sort by.
  /// @param location Reference location to sort for.
  virtual void Sort(const SortOrder &order, const Vec3f &location ) = 0;

  /// Optimizes the geometry buffer according to the specified mode.
  /// This method can be called once for every mode.
  /// @param flags The ored-together flags indicating
  ///              how to optimize the geometry buffer.
  /// @param parameter An additional parameter. Set it to a valid IProgramPtr
  ///                  if you use the OPTIMIZE_PROGRAM_BIND mode.
  virtual void Optimize(const OptimizeFlag &flag,
                        const boost::any &parameter) = 0;

  /// Returns the rendermode this geometry uses for rendering its data.
  /// @return The rendermode this geometry uses for rendering its data.
  virtual RenderMode GetRenderMode() = 0;
};


/** The different hardware program types.
*/
enum ProgramType
{
  PRG_VERTEX, ///< A vertex program.
  PRG_FRAGMENT, ///< A fragment program.
  PRG_GEOMETRY, ///< A geometry program.
  PRG_VERTEX_AND_FRAGMENT ///< A combination of a vertex and fragment program.
};
```

```
/** This struct defines the properties of a shader input parameter.
*/
struct VertexShaderVariableDesc
{
  std::string name; ///< The name of the parameter.
  VertexVariableBindingSemantic semantic; ///< The semantic of the parameter.
  std::string datatype; ///< The name of the datatype of this parameter.
};


/** A programmable hardware shader program representation.
*/
class IProgram : public Yare::Core::Reflection::IUnknown
{
public:
  /// Returns the type this program.
  /// @return The type this program.
  virtual ProgramType GetType() = 0;

  /// The ammount of passes this program uses/needs.
  /// @return The ammount of passes this program uses/needs.
  virtual uint16 GetPasses() = 0;

  /// The uniform input parameters wrapped as variables.
  /// @return The uniform input parameters wrapped as variables.
  virtual Variables::VariableSetPtr GetParameters() = 0;

  /// This method returns a variable descriptions for
  /// each vertex input parameter in the program.
  /// @param vertexElements A list to receive the parameter descriptions.
  virtual void GetVertexElementDescriptions(
    std::vector< VertexShaderVariableDesc > &vertexElements) = 0;

  /// Sets the current technique to use for rendering.
  /// This is used e.g. for CgFX programs or other backends which support
  /// multiple techniques. If no technique is specified for a program, the
  /// first valid technique in the program is used by default.
  /// @param techniqueName The name of the technique to use.
  virtual void SetTechnique(const std::string &techniqueName) = 0;

  /// Returns all render techniques this program supports.
  /// @return The names of the render techniques.
  virtual std::vector<std::string> GetTechniques() = 0;

  /// Begins a given rendering pass.
  /// @param pass The pass to begin with.
  virtual void Begin( const uint16 &pass ) = 0;
```

```
  /// Ends the previous rendering pass.
  virtual void End() = 0;
};
```

A.3 Interfaces of the Effect Framework

```
/** Interface for all effects.
*/
class YARE_GRAPHICS_API IEffect: public Yare::Core::Engine::Actor
{
public:
  /// Returns the identifier of this effect.
  /// @return The identifier of this effect.
  virtual const Yare::Core::Engine::Identifier &GetIdentifier() = 0;

  /// Returns the list of all techniques for this effect.
  /// @return The list of techniques.
  virtual const ITechniqueList &GetTechniques() = 0;

  /// Returns an ITechnique which is supported on the current hardware
  /// and is suitable for the given LOD. Usually the LOD value is the
  /// distance from the camera position to the center of the bounding
  /// volume of the rendered geometry. If more than one technique is
  /// available for a given LOD the algorithm tries to detect which
  /// technique will be used next.
  /// @param effectLOD The LOD of the technique wanted. Use a negative number
  ///                  for the most detailed technique.
  /// @return The ITechnique, or NULL if no technique is
  ///         supported on the current hardware.
  virtual ITechniquePtr GetTechnique(const float effectLOD) = 0;

  /// Sets a function which gets called whenever this effect changes.
  /// @param callback The callback function.
  virtual void SetOnChange(const OnChangeCallback &callback) = 0;

  /// Indicates if this effect changes the bounding
  /// volume of an object if applied to it.
  /// @return True if this effect changes the bounding volume,
  ///         false otherwise.
  virtual bool ChangesBoundingVolume() const = 0;

  /// Returns the changed boundingvolume.
  /// @param volume The current bounding volume of the object in world space.
  /// @return The changed boundingvolume.
  virtual Yare::Core::Math::IVolumePtr ChangeBoundingVolume(
    const Yare::Core::Math::IVolumePtr &volume) = 0;
};


/**
* A technique implements an effect in a special way.
```

```
*/
class YARE_GRAPHICS_API ITechnique: public Yare::Core::Reflection::IUnknown
{
public:
  /// Returns the identifier of this technique.
  /// @return The identifier of this technique.
  virtual const Yare::Core::Engine::Identifier &GetIdentifier() = 0;

  /// Returns the list of parts this technique consists of.
  /// @return The list of technique parts.
  virtual const ITechniquePartList &GetParts() = 0;

  /// Returns the LOD range for this technique.
  /// @param minLOD The minimum LOD value for this technique.
  /// @param maxLOD The maximum LOD value for this technique.
  virtual void GetLODRange(float &minLOD, float &maxLOD) const = 0;

  /// Sets the LOD range for this technique.
  /// @param minLOD The minimum LOD value for this technique.
  /// @param maxLOD The maximum LOD value for this technique.
  virtual void SetLOD(const float &minLOD, const float &maxLOD) = 0;

  /// Checks if this technique is supported on the current hardware.
  /// @return True if it is supported, False otherwise.
  virtual bool IsSupported() = 0;

  /// Returns the list of inputs of all parts of this technique.
  /// @return The list of inputs.
  virtual const IInputList &GetInputs() = 0;

  /// Returns an input with the specified name or NULL if no input with
  /// that name was found. If more than one input with that name exists,
  /// only the first one will be returned.
  /// @param name The name of the input.
  /// @return The input.
  virtual IInputPtr GetInput(const std::string &name) = 0;

  /// Sets the specified value to ALL inputs with the given name.
  /// @param name The name of the input.
  /// @param value The value to set.
  /// @return True if any input with that name was found, False otherwise.
  virtual bool SetInputValue(const std::string &name,
                             const boost::any &value) = 0;
};

/**
```

```
* Interface for all parts of a technique.
*/
class YARE_GRAPHICS_API ITechniquePart:
                        public Yare::Core::Reflection::IUnknown
{
public:
  /// An enumeration of combinable filters.
  enum Filter
  {
    TPF_DEPTH_CHANGING = 1, ///< Only technique parts match this filter
                            ///  if they change the depth value of fragments.
  };

  /// Returns the identifier of this technique part.
  /// @return The identifier of this technique part.
  virtual const Yare::Core::Engine::Identifier &GetIdentifier() = 0;

  /// Returns the list of inputs for this technique part.
  /// @return The list of inputs.
  virtual const IInputList &GetInputs() = 0;

  /// Returns the list of outputs for this technique part.
  /// @return The list of outputs.
  virtual const IOutputList &GetOutputs() = 0;

  /// Returns the group index of this technique part. Group indices can be
  /// used to put different technique parts into different rendering passes.
  /// The default group index is 0.
  /// @return The group index of this technique part.
  virtual const uint32 &GetGroupIndex() const = 0;

  /// Returns the list of state dependencies for this technique part.
  /// If the state dependencies are not fulfilled, this
  /// technique part is not ready for rendering.
  /// @return The list of state dependencies.
  virtual const StateDependencyList &GetStateDependencies() = 0;

  /// Returns the list of state changes this technique part will
  /// perform after rendering. Other technique parts may be
  /// dependend on this state changes.
  /// @return The list of state changes.
  virtual const StateChangeList &GetStateChanges() = 0;

  /// Returns if this technique part is equal to the given technique part.
  /// @param part The technique part to check with this technique part.
  /// @return True if the technique parts are equal, False otherwise.
```

```
  virtual bool IsEqual(const ITechniquePartPtr &part) = 0;

  /// Returns a version of this technique part which matches the given filter.
  /// @return The version of this technique part which matches the filter or
  ///         NULL if this technique part should has no matching version.
  virtual ITechniquePartPtr GetFilteredVersion(const Filter &filter) = 0;

  /// Returns the technique this part belongs to.
  /// @return The technique this part belongs to.
  virtual const ITechniquePtr &GetTechnique() const = 0;
};


/** The interface of all render passes.
*/
class YARE_GRAPHICS_API IPass: public Yare::Core::Reflection::IUnknown
{
public:
  /// Returns the identifier of this pass.
  /// @return The identifier of this pass.
  virtual const Yare::Core::Engine::Identifier &GetIdentifier() = 0;

  /// Returns the list of inputs for this pass. This will also include
  /// inputs of assigned technique parts.
  /// @return The list of inputs.
  virtual const IInputList &GetInputs() = 0;

  /// Returns the list of outputs for this pass. This will also include
  /// outputs of assigned technique parts.
  /// @return The list of outputs.
  virtual const IOutputList &GetOutputs() = 0;

  /// Returns the list of state dependencies for this pass.
  /// If this state dependencies are not fulfilled,
  /// this pass is not ready for rendering.
  /// @return The list of state dependencies.
  virtual const StateDependencyList &GetStateDependencies() = 0;

  /// Returns the list of state changes this pass will perform on
  /// rendering. Other passes may be dependend on this state changes.
  /// @return The list of state changes.
  virtual const StateChangeList &GetStateChanges() = 0;

  /// Renders the provided geometries with this pass.
  /// @param variables Variables which can contain e.g. renderstates
  ///                  or values for inputs.
  /// @param geometry A list of geometries which
```

```
  ///                 should be rendered with this pass.
  virtual void Render( const std::vector<
    std::vector<Yare::Graphics::Variables::IVariable*> > &variables,
    const std::vector<Yare::Graphics::Rendering::IGeometry*> &geometry) = 0;

  /// Returns if this pass is equal to the given pass.
  /// @param pass The pass to check with this pass.
  /// @return True if the passes are equal, False otherwise.
  virtual bool IsEqual(const IPassPtr &pass) = 0;

  /// Creates a copy of this pass.
  /// @return The created clone of this pass.
  virtual IPassPtr Clone() = 0;
};
```

A.4 Interface of the Drawgraph

```
/** The interface for all drawgraphs.
*/
class YARE_GRAPHICS_API IDrawGraph: public Yare::Core::Reflection::IUnknown
{
public:

  /// Adds a renderable to the drawgraph. If the drawgraph already
  /// contains this renderable
  /// it will not be added twice.
  /// @param renderable The renderable to add.
  virtual void AddRenderable(const RenderablePtr &renderable) = 0;

  /// Removes a renderable from the drawgraph.
  /// @param renderable The renderable to remove.
  virtual void RemoveRenderable(const RenderablePtr &renderable) = 0;

  /// Tells the drawgraph that the data of the renderable
  /// has changed (bounding volume, transformation).
  /// @param renderable The renderable that has changed.
  virtual void RenderableChanged(const RenderablePtr &renderable) = 0;

  /// Returns a list of renderables which are all visible with respect
  /// to the given list of culling objects.
  /// @param cullers The list of cullers to which the renderables
  ///                are tested to.
  /// @param renderables A reference to a list which will
  ///                    contain the result of this method.
  virtual void GetRenderables(const std::vector<ICullerPtr> &cullers,
                              std::vector<RenderablePtr> &renderables) = 0;

  /// Returns a geometry that contains debugging geometry for this drawgraph.
  /// @return The debugging geometry.
  virtual Yare::Graphics::Rendering::IGeometryPtr GetDebugGeometry() = 0;

  /// Returns a pass which is used to render the debugging geometry.
  /// @return The debugging pass.
  virtual Yare::Graphics::Effect::IPassPtr GetDebugPass() = 0;

  /// Returns a list of variables which are used to
  /// render the debugging geometry.
  /// @return The debugging variables.
  virtual Yare::Graphics::Variables::IVariableList GetDebugVariables() = 0;

  /// Optimizes the drawgraph.
```

```
  /// The implementation is free to decide what to optimize.
  virtual void Optimize() = 0;

  /// Returns a minimal bounding box containing
  /// all the renderables in this drawgraph.
  /// @return The bounding box of all renderables.
  virtual const Box3f &GetSceneBoundingBox() = 0;
};
```

B UML Diagrams

The UML diagram of the effect framework as described in Section 3.8 can be found in Figure B.1.

The UML diagram of the scenegraph as described in Section 3.10 can be found in Figure B.2.

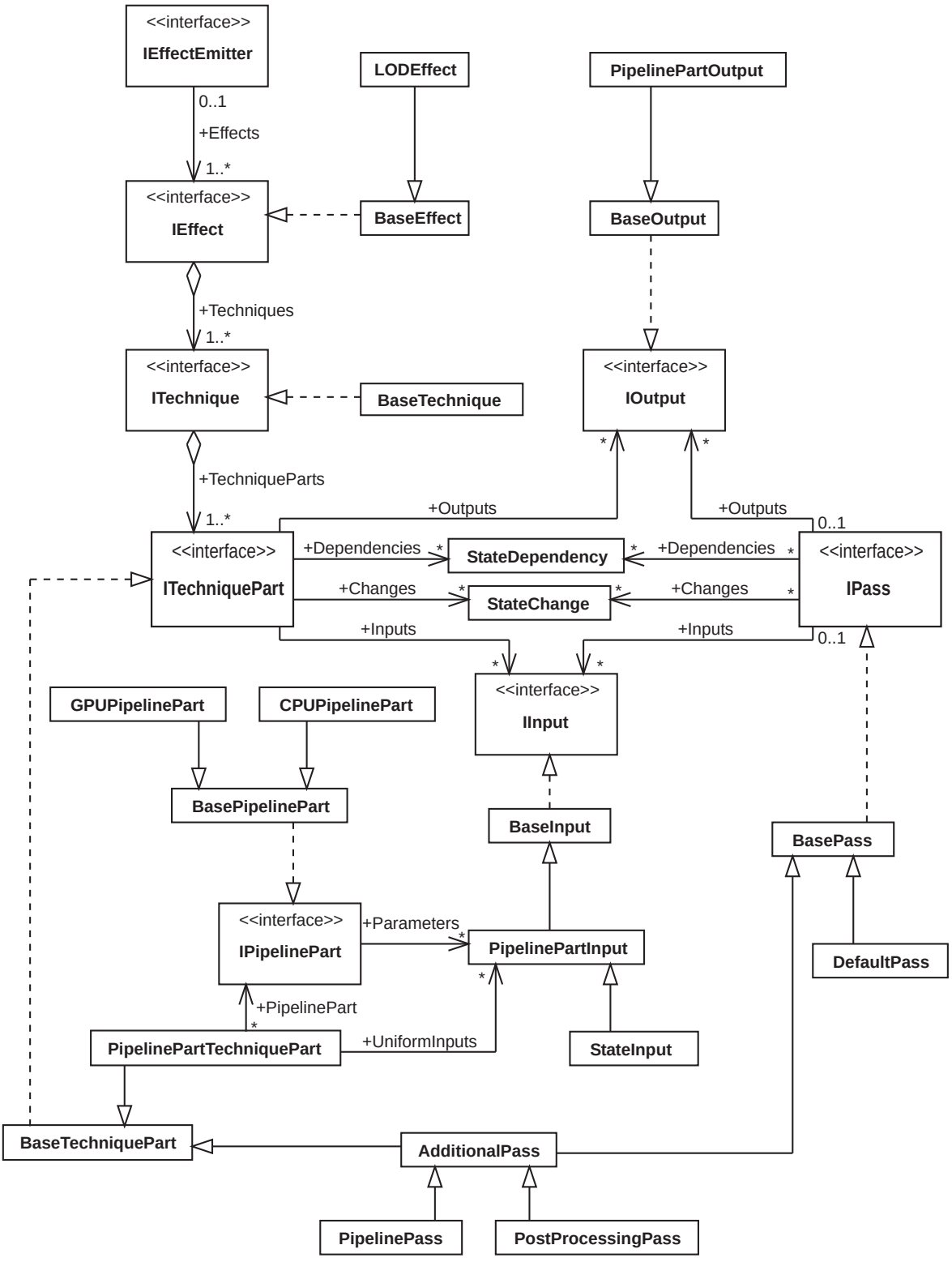

Figure B.1: The UML diagram of the effect framework

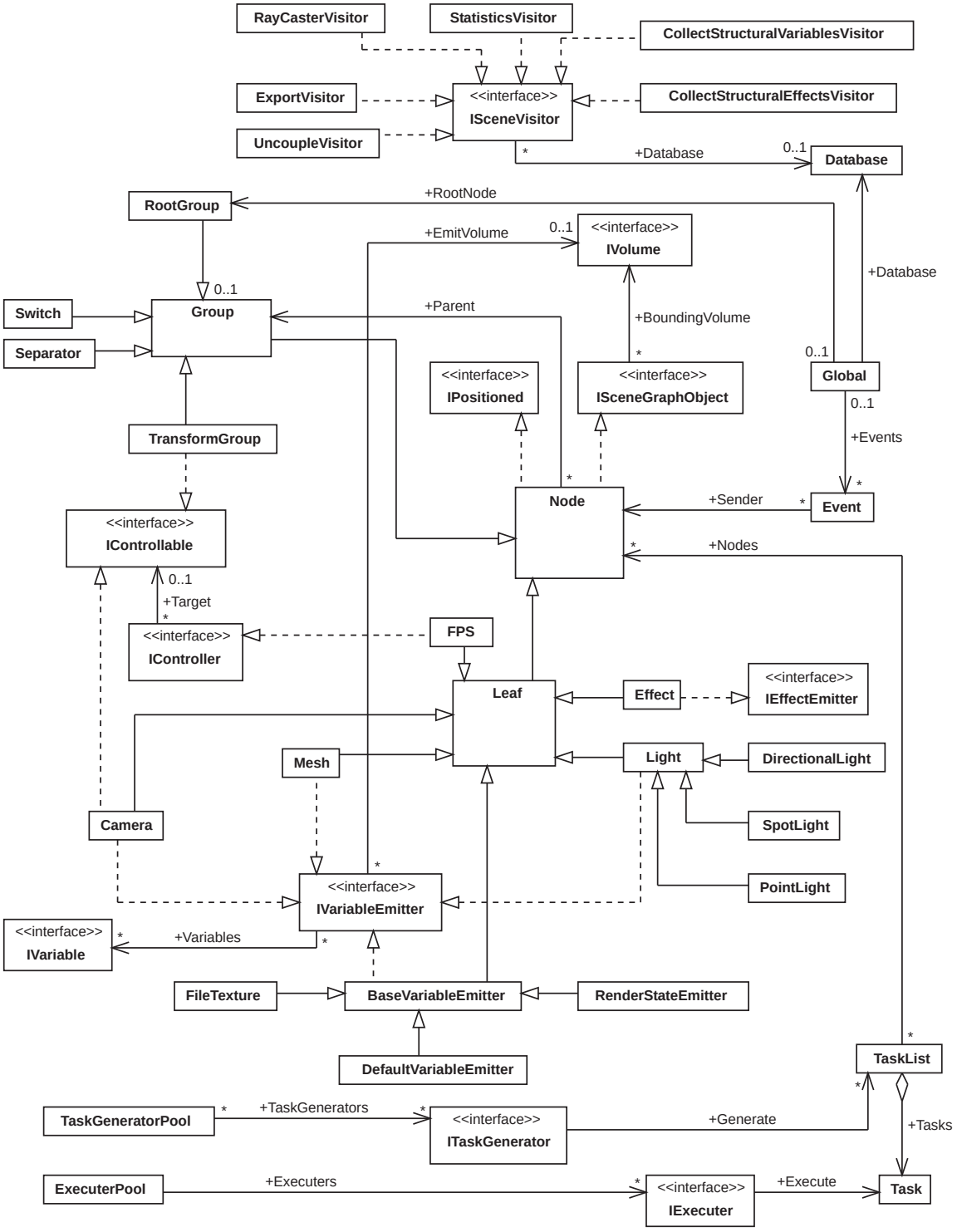

Figure B.2: The UML diagram of the scenegraph

List of Figures

1.1 Comparison real-time rendering and global illumination 1

2.1 The stages of the *graphics pipeline* 7
2.2 Transformation calculations 9
2.3 Extended GPU graphics pipeline 11
2.4 Octree containing two objects 19
2.5 kd-tree containing three objects 20
2.6 An example of a scenegraph 21
2.7 The bounding volume hierarchy of a scenegraph 23
2.8 A scenegraph containing nodes with multiple parents 23
2.9 A scenegraph using link nodes and a shared group 24
2.10 A scenegraph using node components 24

3.1 Structure of a typical application using YARE 35
3.2 Graphical representation of the *Include Guide* 36
3.3 The graphics pipeline of YARE 42
3.4 Variables forming a geometry 49
3.5 A geometry containing two triangles 50
3.6 A geometry containing one triangle starting at offset 2 50
3.7 A geometry containing one triangle and a trianglestrip 50
3.8 A geometry using an indexbuffer 51
3.9 A scene containing a mirror 60
3.10 The layout of the main components of the effect framework 61
3.11 Outline of the runtime behaviour of the effect framework 62
3.12 Usage of Cg in the effect framework 64
3.13 Illustration of the inputs and outputs of vertex and fragment programs 67
3.14 A dependency graph example 73
3.15 Example showing the effect LOD in use 74
3.16 Selecting the technique with Effect LOD 75
3.17 The passes to implement *blooming* 91
3.18 Octree Implementation in YARE 97
3.19 The complete disentangling of data and algorithms in YARE 99
3.20 Sample scenegraph of a car 104
3.21 The sequence diagram of the event processing in YARE 106
3.22 Retrieval of Renderable Objects 110
3.23 An example of a scenegraph with multiple cameras 111
3.24 A simple example of a scenegraph 112
3.25 A scenegraph setup to implement *shadow mapping* 113
3.26 The rendered image of the scenegraph from Figure 3.25 114

4.1 A screenshot of the editor 121

B.1 The UML diagram of the effect framework 153
B.2 The UML diagram of the scenegraph 154

List of Tables

3.1 The submodules of the *Core* module 34
3.2 The submodules of the *Graphics* module 35
3.3 Implemented intersection routines in the math submodule 38
3.4 Details of the *VariableDesc* structure 45
3.5 Examples for variable emitters used in the scenegraph 48
3.6 Available Cg-interfaces of the effect framework 94

4.1 External software packages used by YARE 120

Bibliography

[ARB] *OpenGL Architecture Review Board*, http://www.opengl.org/about/arb/

[BittnerEtAl04] Jiří Bittner, Michael Wimmer, Harald Piringer and Werner Purgathofer, *Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful*, EUROGRAPHICS 2004 (Grenoble, France), Computer Graphics Forum Journal (Sept. 2004, pp.615–624, ISSN 0167-7055), http://www.cg.tuwien.ac.at/research/publications/2004/Bittner-2004-CHC/

[Boost] *boost C++ libraries*, http://www.boost.org/

[BuckEtAl04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike and H. Pat, *Brook for GPUs: Stream Computing on Graphics Hardware*, Submitted to ACM Transactions on Graphics, 2004, http://graphics.stanford.edu/projects/brookgpu

[Catmull75] Edwin Catmull, *Computer Display of Curved Surfaces*, In Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures, Los Angeles, pp.11–17, May 1975.

[Cg] NVIDIA Corporation, *Cg Toolkit*, http://developer.nvidia.com/object/cg_toolkit.html

[CgFX] NVIDIA Corporation, *CgFX Overview*, http://developer.nvidia.com/object/cg_users_manual.html

[Clark76] James H. Clark, *Hierarchical Geometric Models for Visible Surface Algorithms*, Communications of the ACM, vol. 19, no. 10, 1976, pp.547–554.

[Collada] Khronos Group, *Collada - 3D Asset Exchange Schema*, http://www.khronos.org/collada/

[CohenEtAl93] Michael F. Cohen, Chris Tchou, John R. Wallace, *Radiosity and Realistic Image Synthesis*, Academic Press Professional, Boston, 1993.

[DeeringEtAl88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy and Neil Hunt, *The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics*, In Proceedings of SIGGRAPH 88 (Atlanta, Georgia, August 1-5, 1988). In Computer Graphics, v22n4, ACM SIGGRAPH, August 1988. pp.21-Ű30.

[Doxygen] Dimitri van Heesch, *Doxygen - Source code documentation generator tool*, http://www.stack.nl/ dimitri/doxygen/

[D3D] Microsoft Direct3D, http://www.microsoft.com/directx/

[Eccles00] Allen Eccles, *The Diamond Monster 3Dfx Voodoo 1*, Gamespy Hall of Fame, 2000, http://archive.gamespy.com/halloffame/october00/voodoo1/

[Ellsworth90] David Ellsworth, *Parallel Architectures and Algorithms for Real-time Synthesis of High-quality Images using Deferred Shading*, Workshop on Algorithms and Parallel VLSI Architectures (Pont-á-Mousson, France, June 12, 1990).

[Expat] James Clark, *The Expat XML Parser*, http://expat.sourceforge.net/

[Freetype] *The FreeType Project*, http://expat.sourceforge.net/

[GieglWimmer07] Markus Giegl and Michael Wimmer, *Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transition*, Computer Graphics Forum Journal, Volume 26, Nr. 1, March 2007, ISSN 0167-7055, http://www.cg.tuwien.ac.at/research/publications/2007/GIEGL-2007-UNP/

[GLSL] *The OpenGL Shading Language*, http://www.opengl.org/documentation/glsl/

[GovindarajuEtAl03] Naga K. Govindaraju, Stephane Redon, Ming C. Lin and Dinesh Manocha, *Cullide: interactive collision detection between complex models in large environments using graphics hardware*, In Proceedings of ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, 25Ű32.

[Hargreaves04] Shawn Hargreaves, *Generating Shaders from HLSL Fragments*, SHADERX3 - Advanced Rendering With DirectX And OpenGL, 2004, ISBN 1-58450-357-2, Charles River Media.

[HLSL] Microsoft Developer Network, *Introduction to the DirectX 9 High-Level Shader Language*, http://msdn2.microsoft.com/en-us/library/ms810449.aspx

[Irrlicht] Nikolaus Gebhardt et al., *Irrlicht Engine - A free open source 3d engine*, http://irrlicht.sourceforge.net/

[JasPer] Michael Adams, *The JasPer Project*, http://www.ece.uvic.ca/ mdadams/jasper/

[JAVA] Sun Developer Network, *The Java Programming Language*, http://java.sun.com/

[Jpg] Independent JPEG Group, *Library for JPEG image compression*, http://www.ijg.org/

[J3D] Java 3D, *Master project for Java 3D projects*, http://java3d.dev.java.net/

[Kajiya86] James T. Kajiya, *The Rendering Equation*, In Proceedings of the SIGGRAPH 1986, August 1986, pp.143–150.

[KanekoEtAl01] Kaneko T., Takahei T., Inami M., Kawakami N., Yanagida Y., Maeda T., Tachi S., *Detailed Shape Representation with Parallax Mapping*, In Proceedings of ICAT 2001, pp.205–208.

[Libpng] Guy Eric Schalnat et al., *libpng - The official PNG reference library*, http://www.libpng.org/

[Lib3ds] *lib3ds ANSI-C library for 3ds models*, http://lib3ds.sourceforge.net/

[Lua] Pontifical Catholic University of Rio de Janeiro, *The programming language Lua*, http://www.lua.org/

[Luabind] Rasterbar Software, *Luabind library*, http://www.rasterbar.com/products/luabind.html

[MacDo90] MacDonald J. and Booth K., *Heuristics for ray tracing using space subdivision*, The Visual Computer, Vol. 6, No. 3, 1990, pp.153Ű-166.

[MarkEtAl03] W. Mark, S. Glanville and K. Akeley, *Cg: A system for programming graphics hardware in a C-like language*, ACM Transactions on Graphics, August 2003, http://citeseer.ist.psu.edu/mark03cg.html

[McCoolEtAl04] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan and Kevin Moule, *Shader algebra*, ACM Trans. Graph. 2004, Volume 23, Number 3, ISSN 0730-0301, pp.787–795, ACM Press, http://citeseer.ist.psu.edu/mccool04shader.html

[McGuireEtAl06] Morgan McGuire, George Stathis, Hanspeter Pfister and Shriram Krishnamurthi, *Abstract Shade Trees*, In Proceedings of the Symposium on Interactive 3D Graphics and Games, March 2006, Redwood City, CA.

[MS] Microsoft Corporation, *Microsoft Website*, http://www.microsoft.com

[MSEF] Microsoft Corporation - MSDN Online, *Effect Reference*, http://msdn.microsoft.com/archive/en-us/directx9_c_Dec_2005/dx9_graphics_reference_effects.asp

[MSVS] Microsoft Visual Studio Developer Center, http://msdn.microsoft.com/vstudio/

[NVIDIA] NVIDIA Website, http://www.nvidia.com/

[NVSG] NVIDIA Corporation, *NVSG Homepage*, http://developer.nvidia.com/object/nvsg_home.html

[OliveiraAtAl00] Oliveira, Manuel M., Gary Bishop, David McAllister, *Relief Texture Mapping*, Proceedings of SIGGRAPH 2000 (New Orleans, La), July 23-28, 2000, pp.359–368.

[OGLExt] SGI OpenGL website, *The OpenGL Extension Registry*, http://www.opengl.org/registry/

[OGRE] OGRE 3D website, *OGRE 3D - Open source graphics engine*, http://www.ogre3d.org/

[OpenEXR] Industrial Light & Magic, *The OpenEXR high dynamic-range image file format*, http://www.openexr.com/

[OpenGL] Open Graphics Language Website *OpenGL - The Industry's Foundation for High Performance Graphics*, http://www.opengl.org/

[OpInv] SGI, *Open Inventor - An object-oriented 3D toolkit*, http://oss.sgi.com/projects/inventor/

[ORorke04] John O'Rorke, *Integrating Shaders into Applications*, GPU Gems, Addison Wesley, Boston, 2004, ISBN 0-321-22832-4, pp.601–615.

[OSG] *OpenSG - Main website*, http://opensg.vrsource.org/

[OScGr] *OpenSceneGraph - Main website*, http://www.openscenegraph.org/

[Patterns95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995, ISBN 0-201-63361-2.

[Pharr04] Matt Pharr, *An Introduction to Shader Interfaces*, GPU Gems, Addison Wesley, Boston, 2004, ISBN 0-321-22832-4, pp.537–550.

[Python] The Python Software Foundation, *The Python Programming Language*, http://www.python.org/

[Radiance] Greg Ward, *Radiance - Synthetic Imaging System*, http://radsite.lbl.gov/radiance/

[RPly] Diego Nehab, *RPly - ANSI C Library for PLY file format input and output*, http://www.cs.princeton.edu/ diego/professional/rply/

[RTR02] Tomas Akenine-Möller and Eric Haines, Real-Time Rendering, Second Edition, A K Peters, Natick, Massachusetts, 2002, pp.109–114.

[Samet89] Hanan Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, Massachusetts, 1989.

[Schlick94] Christophe Schlick, *High Dynamic Range Pixels*, Graphics Gems IV, Academic Press Professional, Boston, 1994, pp.422–430, http://citeseer.ist.psu.edu/schlick93high.html

[SpencerEtAl95] Greg Spencer, Peter Shirley, Kurt Zimmerman and Donald P. Greenberg, *Physically-Based Glare Effects for Digital Images*, Computer Graphics, Volume 29, 1995, pp.325–334.

[TumblinRushmeier93] Jack Tumblin and Holly E. Rushmeier, *Tone reproduction for realistic images*, IEEE Computer Graphics and Applications, 13(6):42.48, November 1993.

[WhittedWeimer81] T. Whitted and D. M. Weimer, *A software test-bed for the development of 3-D raster graphics systems*, Proceedings of SIGGRAPH 81 (Dallas, Texas, July 1981). In Computer Graphics, v15n3. ACM SIGGRAPH, August 1981. pp.271-Ű277.

[Williams78] Lance Williams, *Casting curved shadows on curved surfaces*, Computer Graphics, vol. 23, no. 3, 1978: pp.270–274.

[Windows] Microsoft, *Microsoft Windows Operation System*, http://www.microsoft.com/windows/default.mspx

[WimmerEtAl99] Michael Wimmer, Markus Giegl and Dieter Schmalstieg, *Fast Walkthroughs with Image Caches and Ray Casting*, Eurographics Workshop 1999 on Virtual Environments [Extended Version TR-186-2-98-30].

[WimmerRTR04] Michael Wimmer, *Lecture on Realtime Rendering*, Vienna University of Technology, Institute of Computer Graphics and Algorithms, http://www.cg.tuwien.ac.at/courses/Realtime/VU.html

[WonkaSchmalstieg99] Peter Wonka and Dieter Schmalstieg, *Occluder Shadows for Fast Walkthroughs of Urban Environments*, Computer Graphics Forum (Proc. Eurographics Š99), 18(3):51-60, September 1999.

[WxWidgets] *wxWidgets - Cross-Platform GUI library*, http://www.wxwidgets.org/

[YareCodingGuide] Matthias Bauchinger, *YARE 2.0 - Coding Guidlines*, http://www.yare.at/yare2/CodingGuidelines.pdf

[Zeller06] Cyril Zeller, *Practical Cloth Simulation on Modern GPUs*, SHADERX4 - Advanced Rendering Techniques, 2006, ISBN 1-58450-425-0, pp.17–27, Charles River Media.

[Zlib] Jean-loup Gailly and Mark Adler, *zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library*, http://zlib.net/