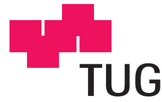

Graz University of Technology
Institute for Computer Graphics and Vision
Inffeldgasse 16, A-8010 Graz

In Cooperation With

Atronic Austria GmbH
Am Seering 13-14, A-8141 Unterpremstätten

Master's Thesis

Real-Time 3D Rendering for the Atronic EGD Framework: A Hybrid Approach

---

Bakk. techn. Christopher S. Dissauer

Graz, Austria, November 2009

Thesis supervisor
Univ.Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg
Thesis advisor
Dipl.-Ing. Lukas Gruber

Abstract

The Austrian company *Atronic* is engaged in developing and manufacturing video-based gaming machines and display systems for the world-wide casino market. Displaying graphical content on these machines is performed via the company's proprietary "EGD" scene graph framework; historically, the original design of this framework was strongly influenced by the comparatively low GPU performance of the available hardware platform, resulting in a necessarily CPU-bound 2D-only implementation. This thesis presents an approach to augment the framework by GPU-accelerated real-time 3D rendering, suitable for operation on Atronic's upcoming next-generation hardware platforms.

Necessary goals and requirements were defined; during this process, it was decided to aim for a hybrid approach that represents a novel way for seamless integration of 3D functionality into the existing 2D system. Based on these drafts, a preliminary study was carried out to ensure overall technical feasibility within the given hardware and software limits. With respect to the results of this study, a concrete design was laid out that eventually led to the implementation of a working prototype.

Corner test cases have been prepared to evaluate the hybrid prototype system with regard to extensibility, stability and performance, yielding overall satisfactory results. Nevertheless, certain inadequacies were revealed through this evaluation; most prominently, the techniques used for parallelization of CPU-bound and GPU-bound tasks showed considerable room for further optimization towards a technically mature product.

Kurzfassung

Das österreichische Unternehmen *Atronic* befasst sich mit der Entwicklung und Produktion von videobasierten Spielgeräten und Display-Systemen für den weltweiten Casino-Markt. Die grafische Darstellung von Inhalten auf diesen Geräten erfolgt über das firmeneigene proprietäre "EGD" Szenengraphen-Framework; beeinflusst durch die vergleichsweise geringe GPU-Leistung der verfügbaren Hardware-Plattform wurde dieses Framework gezwungenermaßen über einen ausschließlich CPU-basierten 2D-Ansatz konzipiert und realisiert. Diese Arbeit präsentiert einen Ansatz zur Erweiterung des bestehenden Frameworks um hardwarebeschleunigtes Echtzeit-3D-Rendering, geeignet für den Einsatz auf zukünftigen Atronic Hardware-Plattform-Generationen.

Notwendige Ziele und Anforderungen wurden definiert; während dieses Vorgangs wurde die Entscheidung gefällt, einen Hybrid-Ansatz zu verfolgen der einen neuartigen Weg zur nahtlosen Integration von 3D-Funktionalität in das bestehende 2D-System darstellt. Basierend auf diesen Definitionen wurden Untersuchungen durchgeführt, die die grundsätzliche technische Machbarkeit innerhalb gegebener Hard- und Software-Grenzen sicherstellen. Die Resultate dieser Untersuchungen führten weiters zu einem konkreten Design und schlussendlich zur Implementierung eines funktionierenden Prototypen.

Testfälle wurden definiert um das prototypische Hybrid-System hinsichtlich Erweiterbarkeit, Stabilität und Leistungsvermögen im Grenzbereich zu evaluieren, mit insgesamt zufrieden stellenden Resultaten. Nichtsdestoweniger wurden durch diese Evaluation gewisse Unzulänglichkeiten offenbart; die verwendeten Mechanismen zur Parallelisierung von CPU- und GPU-lastigen Prozessen boten hierbei die markantesten Ansatzpunkte für weitere Optimierungen in Richtung eines ausgereiften Produktes.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Acknowledgments

First and foremost, I would like to equally thank my supervisor at the Institute of Computer Graphics and Vision, Prof. Dieter Schmalstieg, my advisor, Dipl.-Ing. Lukas Gruber, and my superiors at Atronic, with department head Patrick Huiber as my direct contact person on these matters, who together offered me the possibility to write this thesis as part of my regular occupation. The challenge of finishing my university studies besides an interesting albeit demanding full-time job has never been an easy one; only through this courtesy was I able to finish this thesis in finite time.

I would also like to sincerely thank my parents, Eveline and Erwin Dissauer, for their never-ending encouragement and support towards my graduation, my ”little” siblings Jörg and Ulla for providing that little extra motivation thanks to their slightly higher studying pace, and of course my grandmother, Margarethe Knabl, for simply always being there.

And, last but not least, I would like to say thank you to all my good friends for being who they are; for all those blurry late night discussions, for all those plans to set the world on fire. For all the fun and all the skiing trips, but also for always lending me an ear in times of need.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Structure of this Thesis 2

2 Related Work **4**

2.1 Scene Graphs 4

2.2 Inventor 6

2.3 Performer 9

2.4 Atronic EGD Framework 10

3 Goals and Requirements **16**

3.1 User-Level Requirements 17

3.1.1 SDK Extension 17

3.1.2 3D Content Representation 18

3.1.3 Coordinate System Alignment 19

3.2 System-Level Requirements 19

3.2.1 Platform Scope 19

3.2.2 Performance 20

3.2.3 System Integration 21

4 A Technical Feasibility Study **22**

4.1 Pre-Conditions 24

4.1.1 2D Surface Pixel Transfer 24

4.1.2 Geometry Rendering 25

4.1.3 Offscreen Rendering 26

4.1.4 Geometry Representation 26

4.2 Graphics Driver Analysis 28

4.2.1 OpenGL API Versions 28

4.2.2 OpenGL Performance 29

4.2.3 Graphics Driver Evaluation 30

4.2.3.1 nVidia driver for GeForce/Ion chipsets 31

4.2.3.2 ATI Catalyst driver for Radeon chipsets 31

4.2.3.3 Intel GMA Driver . . . 31
4.2.3.4 Intel Embedded Graphics Driver (IEGD) . . . 32
4.2.3.5 Intel Open Source DRI drivers for Mesa3D . . . 33
4.3 Preliminary Performance Measurements . . . 33
4.4 Conclusions . . . 35

5 Concepts and Prototype Design 37
5.1 2D/3D Interaction . . . 37
5.1.1 3D Renderer Areas . . . 38
5.1.2 Texture Targets . . . 39
5.1.3 Renderer Engine . . . 40
5.1.4 Parallelization . . . 41
5.2 Materials . . . 42
5.2.1 Multi-Pass Materials . . . 43
5.2.2 GPU Shader Programs . . . 43
5.3 Node Instancing and Referencing . . . 44
5.4 Geometry Resources . . . 44

6 Prototype Implementation 46
6.1 Framework Integration . . . 46
6.2 Geometry Resources . . . 46
6.2.1 The E3D File Format . . . 47
6.2.1.1 Geometry Header Chunks . . . 47
6.2.1.2 Material Chunks . . . 48
6.2.1.3 Mesh Geometry Chunks . . . 48
6.2.2 EGD Assets . . . 48
6.3 Scripting Nodes . . . 49
6.3.1 Renderer Areas . . . 49
6.3.2 Geometric Transformations . . . 51
6.3.3 Cameras . . . 52
6.3.4 Light Sources . . . 53
6.3.5 Materials . . . 54
6.3.6 Textures . . . 56
6.3.7 Shader Programs . . . 57
6.3.8 Drawables . . . 58

7 Content Creation Workflow 61
7.1 Geometry File Conversion . . . 61
7.2 Tool Chain Extension . . . 63

8 Application Examples 65
8.1 Progressive Meter 65
8.2 Interactive Game Selection 68
8.3 Vertex and Fragment Shaders 70
8.4 The Background as a Texture 73

9 Results and Discussion 75
9.1 Performance Measurements 75
9.1.1 Surface Transfer Performance 75
9.1.2 Rendering Performance 76
9.2 Future Improvements 78
9.2.1 Improved Parallelization 78
9.2.2 Higher-Level Node Classes 79

10 Conclusion 80

Bibliography 82

List of Figures

1.1 A typical setup of a bank of six linked gaming machines up front and a large overhead display on top. 2

2.1 A simple scene graph representing a colored cube and sphere, with an additional transformation applied to the sphere. From [15] 5
2.2 A section of a scene graph representing the front and rear wheels of a bicycle, both referring to the same shape node. The path from the sub-graph's local root to the rear wheel's shape is highlighted by the heavy line. From [15] . 6
2.3 Layering of IRIS Performer's core libraries in conjunction with operating system and application. From [13] 9
2.4 An overview of EGD's software components and their inter-component relationships. Platform-related components are shown in tan color, and individual game content is highlighted in orange color. 11
2.5 Block diagram of the EGD Multimedia base software. Low-level utility and third-party libraries are omitted for clarity. 12

4.1 Texture update performance: Maximum achievable frame rate for various driver configurations. 34
4.2 Texture update performance: CPU load (single core) for various driver configurations at a fixed frame rate of 60fps. 35

5.1 An exemplary fragment of a quasi-heterogeneous scene graph consisting mainly of 2D nodes (tan color), with additional 3D nodes inserted (orange color) by means of a switcher node (`MOB_GlRendererArea`). 38
5.2 A fragment of a quasi-homogenous 3D scene graph showing how to use the 2D rendering engine for effectively generating dynamic textures by means of a `MGL_TextureDefinition` node. 39
5.3 A scene graph fragment depicting the relationship between MGL scripting nodes and their underlying renderer objects. 41
5.4 Sequence diagram showing the interaction between the common *drawing thread* and the device-specific *flip thread* for two subsequent frames. 42

6.1 Block diagram of the EGD Multimedia base software. Low-level utility and third-party libraries are omitted for clarity; the newly added MM_OPENGL library is highlighted in orange color. 47
6.2 Class hierarchy of the newly introduced OpenGL-related scripting nodes, with their connection to the existing base classes in the MML library. . . . 50
6.3 Renderer area class hierarchy. 51
6.4 Class hierarchy of available transformation scripting nodes showing the connections to their encapsulated renderer object. 52
6.5 Camera-related class hierarchy. 53
6.6 Light source-related class hierarchy. 53
6.7 Class hierarchy of material-related scripting nodes and relationship to internal renderer objects. 54
6.8 Class hierarchy of texture definition and reference nodes, together with encapsulated renderer objects. 56
6.9 Class hierarchy of shader-related classes for scripting nodes and renderer objects. 58
6.10 Class hierarchy for geometry-related scripting nodes, renderer objects and assets. 60

7.1 The main window of the *EGD Convert 3D* application, showing geometry and materials imported from a given FBX file. 62
7.2 Tool windows of the *EGD Convert 3D* application showing on-the-fly creation of a DOT3 bump map shader material. 63

8.1 A rendering of the existing two-dimensional progressive meter with default style. 65
8.2 A rendering of a three-dimensional progressive meter using the existing 2D implementation as the source for a dynamic texture. 68
8.3 A screen shot of the game selection menu screen. 68
8.4 A furry and a bump-mapped torus both rendered using vertex and fragment shaders, with the furry one using a multi-pass material to render 16 distinct shells of fur. 70
8.5 An example illustrating the possibility of integrating the 2D surface as a texture object on 3D geometry. 73

9.1 EGD texture update performance: Maximum achievable frame rate for different screen resolutions and varying load. 76
9.2 EGD texture update performance: CPU load (single core) for different screen resolutions and varying load, at a fixed frame rate of 60 fps. 76
9.3 Screen shot of the "coin flow" performance test, with 250 individually animated coins rendered with a DOT3 bump mapping shader. 77

9.4 Rendering performance: Maximum achievable frame rate for the "coin flow" test case, with varying instance count and three materials of varying complexity. 78
9.5 Rendering performance: Single core CPU load for the "coin flow" test case, with varying instance count and three materials of varying complexity, at a fixed frame rate of 60 fps. 78

Chapter 1

Introduction

1.1 Motivation

Being a well-established competitor in the casino market, the Austrian company *Atronic*[*] is known for its innovative portfolio covering a wide spectrum of individual casino-related application areas. Holding regulatory licenses for operation in more than 200 individual jurisdictions world-wide, the company offers a broad range of casino products such as stand-alone slot machines, linked gaming machines with large overhead presentation displays or back-end accounting systems. See figure 1.1 for a typical setup.

As an active developer, manufacturer and vendor of electronic gaming machines (EGMs), the company – not least due to persistent pressure from its competitors – constantly strives to raise their product value by means of various individual actions; one key factor particularly affecting the work of this thesis is the goal of impressing and attracting new players through ever increasing quality of an EGM's visual presentation.

Knowing the capabilities of modern desktop workstations and laptops in the field of real-time 3D graphics, bringing this technology also into the casino market appears to be an all-too-logical step in achieving the above mentioned goal; nevertheless, until the present day, these efforts have been hindered by Atronic's existing EGM hardware platform and its tailor-made software base not supporting any of these features.

However, having a new generation of hardware platforms featuring increased CPU and GPU capabilities ready for launch drastically changes this situation; the only remaining obstacle is formed by the software framework being restricted to CPU-bound 2D rendering only. Recognizing this situation, Atronic's management decided to take the essential steps

[*] http://www.atronic.com

Figure 1.1: A typical setup of a bank of six linked gaming machines up front and a large overhead display on top.

to remedy this shortcoming; as a result, it was agreed to carry out all necessary prototype work within the scope of this thesis, in order to provide the existing framework with advanced real-time 3D features.

1.2 Structure of this Thesis

Following this introduction, chapter 2 gives an overview of related work. Initially presenting a brief review of the general topic of *scene graphs* and a comparison of selected products implementing this concept, the chapter concludes with a closer examination of Atronic's existing framework within this context.

Chapter 3 deals with the process of determining concrete goals and requirements necessary to reach the overall goal of transforming the 2D-only framework into an efficient

2D/3D hybrid. In consideration of the results from this process, chapter 4 gives a detailed analysis of various target hardware platforms, operating systems and graphics drivers, concluding with a statement about overall project feasibility and specific constraints.

Subsequently, chapters 5 and 6 cover in detail the actual steps taken in the course of developing the hybrid prototype, guiding the reader on a path from initial concepts and design details to the final implementation. Building on this foundation, chapters 7 and 8 briefly describe the actual content creation workflow and present selected application examples, to demonstrate the prototype's actual capabilities and to provide basic insight into the functioning of the system.

Finally, chapters 9 and 10 conclude this thesis, presenting the results of a number of real-world performance tests carried out on the final prototype. Whereas chapter 9 gives a brief outline of possible future improvements, chapter 10 provides an overall summary.

Chapter 2

Related Work

During the evolution of 3D rendering hardware and algorithms towards the early 1990s, it soon became apparent that, for developing interactive applications, it is not sufficient to only concentrate on the rendering aspects of given 3D content. Specifically, the need for advanced manipulation methods – either programmatically or via user input – was given, which in turn demanded that existing content not only be represented by e.g. simple display lists, but in fact by some sort of a higher-level description, preferably in a way related to the physical equivalents of the objects it consists of. In this context, the following sections give an introductory overview of an existing solution to these demands in general, followed by an in-depth comparison of actually existing software products implementing this concept.

2.1 Scene Graphs

In 1992, Strauss and Carey [16] addressed the needs described above in their paper, in which they present an object-oriented toolkit written in C++ that is targeted towards developers of interactive 3D graphics applications. Here, they introduce the notion of a *scene graph*, which is essentially a tree or, more general, a *directed acyclic graph* [4] containing abstract representations of 3D objects, so-called *nodes*. These nodes may carry information about cameras, lights, geometries, materials or other properties used for describing a 3D scene, as well as information about dynamic behavior such as animation. Figure 2.1 shows the hierarchy of a simple scene graph.

In this simple example, the scene graph is represented by a tree hierarchy; however, Strauss and Carey state that a node may also be a child of more than one group node, which

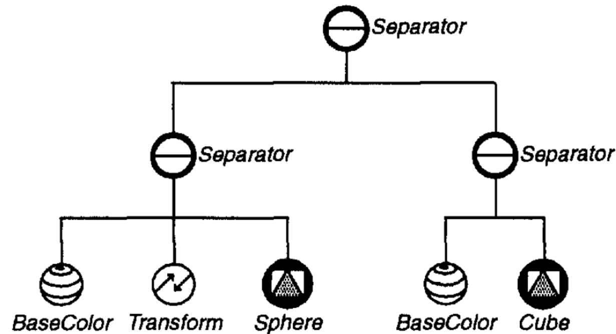

Figure 2.1: A simple scene graph representing a colored cube and sphere, with an additional transformation applied to the sphere. From [15]

allows for re-using common parts of the scene graph. Commonly known as *instancing* [1, p. 356], this technique can possibly reduce the overall complexity of a scene graph; for example, a car wheel only needs to be defined once and may then be instanced four times. Instancing requires the scene graph to be stored as a directed acyclic graph instead of a tree; this can lead to ambiguities when referring to a multiply instanced object. As a solution to this problem, they introduce the concept of a *path* object, which stores information about the traversal from a given node to a different – possibly instanced – node within its sub-graph. Figure 2.2 illustrates this technique.

In order to e.g. render the contents of a scene graph on screen or manipulate specific nodes therein, the toolkit must provide a mechanism to support traversal of the graph. For that reason, Strauss and Carey define a set of *actions* that can be applied to scene graphs, such as rendering, event handling or bounding box computation. A common way to carry out these actions is to traverse the scene graph starting at the root node and recursively process all children from left to right, i.e. in a depth-first manner.

However, while Strauss and Carey actually coined the term *scene graph* in their original paper, they do have a rather relaxed view about it in general; in fact, they do not even provide an explicit definition. Today, numerous commercial as well as non-commercial applications and frameworks exist that basically implement the hierarchical structure and traversal concepts as described, all referring to the generic term *scene graph*. Moreover, Akenine-Möller and Haines even go so far as to state:

> *"In a sense, every graphics application has some form of a scene graph, even if the graph is just a root node with a set of children to display."* [1, p. 355]

In this sense, the scene graph concept is not necessarily restricted to only handling three-

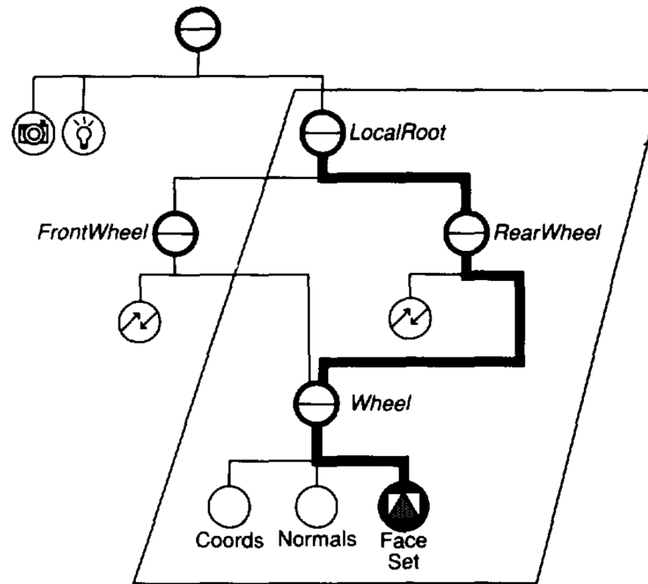

Figure 2.2: A section of a scene graph representing the front and rear wheels of a bicycle, both referring to the same shape node. The path from the sub-graph's local root to the rear wheel's shape is highlighted by the heavy line. From [15]

dimensional content; all the mechanisms described in [16] can also be applied to systems that solely perform 2D rendering e.g. via the *Painter's Algorithm*. In fact, two-dimensional rendering may as well be regarded as just a special case of 3D, restricted to image plane-aligned quadrilaterals rendered through an orthographic projection.

The following sections continue to deal with these important concepts by presenting a comparison of a number of actually available framework implementations. As a counterpart to various existing general-purpose products, also the existing Atronic EGD framework is taken into consideration, which forms the actual basis for this thesis.

2.2 Inventor

Based on Strauss' and Carey's original publication, Strauss introduced Silicon Graphics' *IRIS Inventor* in 1993 [15], which represents the first actual implementation of the concepts described above. He mentions two essential advantages of its object-oriented approach: *encapsulation* and *extensibility*. Encapsulation provides a means to safely hide complex graphical operations or algorithms from the user by creating objects designed for a specific task. Extensibility is maintained through the very nature of the object-oriented paradigm; developers are able to implement, by means of derivation, their own objects that seamlessly

fit into the existing framework.

In Inventor, a scene graph is stored as a directed acyclic graph of nodes. Strauss defines three basic types of such nodes:

- *shapes* represent geometric objects such as cubes, spheres or other primitives
- *properties* provide certain attributes to those shapes such as surface materials or drawing styles, and
- *groups* can hold multiple child nodes and provide the basis for creating hierarchies.

However, nodes in general are not restricted to implement exactly one of these types; Strauss explicitly notes the possibility of certain higher-level compound classes that may incorporate any number of these characteristics at the same time.

Inventor provides the possibility to either build up a scene graph at run-time, i.e. by directly creating and linking different types of node objects programmatically from within some other piece of code, or by loading it from an ".iv" file. See listings 2.1 and 2.2, respectively.

```
1  SoSeparator *root, *sepl, *sep2;
2  SoBaseColor *bl, l b2;
3  SoSphere    *sphere:
4  SoTransform *xf;
5
6  // Create the subgraph with the sphere
7  sep1 = new SoSeparator:
8  b1 = new SoBaseColor;
9  b1->rgb.setValue(1.0, 0.2, 0.2);
10 xf = new SoTransform;
11 xf->translation.setValue(0.0, 3.0, 0.0):
12 sphere = new SoSphere:
13 sphere->radius = 0.3;
14 sep1->addChild(b1);
15 sep1->addChild(xf);
16 sep1->addChild(sphere);
17
18 // Create the subgraph with the cube
19 sep2 = new SoSeparator:
20 b2 = new SoBaseColor;
21 b2->rgb.setValue(0.2, 0.2, 1.0);
22 sep2->addChild(b2);
23 sep2->addChild(new SoCube);
24
25 // Put them together
26 root = new SoSeparator:
27 root->ref();
28 root->addChild(sep1):
```

```
29 root->addChild(sep2);
```

Listing 2.1: A C++ code fragment for generating the scene graph depicted in Figure 2.1. From [15].

```
1  #Inventor V2.0 ascii
2  Separator {
3    Separator {
4      BaseColor {
5        rgb 1 .2 .2
6      }
7      Transform {
8        translation 0 3 0
9      }
10     Sphere {
11       radius .3
12     }
13   }
14   Separator {
15     BaseColor {
16       rgb .2 .2 1
17     }
18     Cube {}
19   }
20 }
```

Listing 2.2: An Inventor ASCII file representing the scene graph depicted in Figure 2.1. From [15].

Instancing of sub-graphs from within an IV scene graph description file is accomplished through the use of the *DEF* and *USE* keywords [19]. Adding the DEF keyword followed by a unique identifier in front of a node statement introduces a *named instance* to the scene graph, consisting of the respective node and all of its children. This named instance can later be referenced any number of times by simply writing the USE keyword, followed by the given identifier anywhere in the IV file's scene graph definition.

To implement the action concept, Strauss chose not to perform distinct method calls on node objects; instead, an action is a separate object on its own. To apply a specific type of action to a specific type of node, he introduced a multi-dispatch scheme that is based on a two-dimensional table, with each row representing one specific action, each column one specific node class, and each cell holding a pointer to a function designed for applying the respective action to the respective node.

In order to provide a consistent, easy-to-use interface for setting and retrieving individual node instance attributes, all such instance data are encapsulated in separate *field* objects, each of which in turn contains one of various common data types such as e.g. integers, floats, vectors or colors. Fields are one of Inventor's key concepts; in addition to

providing the application programmer with a straight-forward way to handle individual properties of a node instance, it is also an effective way for reading and writing scene graphs from/to a file, and it forms the basis for creating animation within a scene graph by means of *field connections* and *engines*.

With the emergence of version 2.0, Silicon Graphics decided to change the product's name from "IRIS Inventor" to "Open Inventor"; after several years of licensing the source code to third party companies, they eventually decided to release it to the Open Source community in 2000. Yet, there are still a number of companies developing commercial products based on that code, such as *Kongsberg SIM* (Coin3D)* or *VSG*†.

2.3 Performer

In 1994, also at Silicon Graphics, Rohlf and Helman introduced *IRIS Performer* [13] (today known as *OpenGL Performer*), basically in an effort to remedy a few key problems still in existence with other similar frameworks, including IRIS Inventor. As the name already suggests, the focus during the development of this framework was put on maximum performance; achieving this goal was primarily accomplished through the use of multiprocessing and heavy optimization of rendering techniques and algorithms.

The Performer toolkit basically consists of two core libraries, named *libpr* and *libpf*, which provide two different API layers to be utilized by an application programmer. Of these two layers, *libpr*, written in C, encapsulates functionality for high-performance primitive rendering, memory sharing and other basic low-level functions. The *libpf* library, written in C++ and sitting on top of *libpr*, provides scene graph/scene database functionality, multiprocessing support and techniques for ensuring real-time behavior. See Figure 2.3.

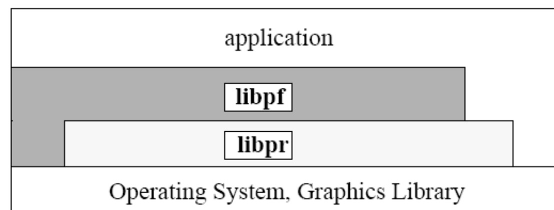

Figure 2.3: Layering of IRIS Performer's core libraries in conjunction with operating system and application. From [13]

*http://www.coin3d.org/
†http://www.vsg3d.com/vsg_prod_openinventor.php

As stated above, Performer's focus was put on optimal render performance, in contrast to the ease-of-use goal pursued in the development of Inventor. Here, a key advantage lies in Performer's ability to natively support multiprocessing. The framework provides efficient mechanisms for distributing its work load over multiple processes; usability is still maintained by effectively hiding the implementation details of these mechanisms from the user.

Another key feature of Performer is that it is designed as a real-time system; here, the most important goal was to establish mechanisms that ensure a constant frame rate, independent of the actual complexity of a scene's visible as well as invisible parts. Striving to produce a steady stream of individually rendered frames without hiccups or glitches, the framework employs techniques for *stress management* to reduce visible scene complexity through level-of-detail reduction, and *overload management* techniques for applying user-defined behavior upon failure of the former techniques.

In contrast to Inventor's extensible action concept, scene graph traversal in Performer is restricted to three basic types: *ISECT* traversals allow detecting object collisions through intersection requests, *CULL* traversals perform object culling, level-of-detail calculations, geometry sorting and preparation of drawing lists, and *DRAW* traversals carry out the actual rendering process. To allow for finer control over the individual traversal steps in this scheme, the framework provides pre- and post-traversal callbacks for each traversal type that may be installed for each node type, e.g. to support customized culling, rendering or state management. In addition, *traversal masks* may be used to selectively inhibit traversal for certain nodes, to allow for e.g. separate geometry for collision determination and rendering.

In its core, Performer represents a runtime-only programming framework; the only way to build a scene graph is by calling exposed API functions from custom code. However, an additional utility library (*pfuBuilder)* is provided to facilitate construction; this library provides individual loader modules for various geometry object and scene database file formats.

2.4 Atronic EGD Framework

In 2001, Atronic introduced the "Hi!bility" hardware platform as a basis for creating high-quality casino games. Based on a customized PC mainboard equipped with an Intel Celeron CPU running at 566 MHz and a specially developed ATI Rage Mobility graphics card supporting a dual-screen configuration, this product by far outperformed the

previous-generation Zilog Z80-based ”CashLine” platform on the presentation side. To take advantage of the platform’s hardware capabilities from the software perspective, it soon became apparent that – compared to the already existing software – also the need for a more sophisticated approach for a game developer framework or SDK was given, with the main goal of reducing content development time yet still increasing content quality.

For that reason, the company started the *Easy Game Development* (EGD) project in 2003. In the scope of this project, an important task was to do a complete re-design of the presentation subsystem software in a more portable way, so that the majority of its code base may be compiled for the target platform as well as a standard developer PC. In addition, the existing code for the control and communications subsystems – each of them running on its own separate Motorola 68k-based CPU board – also had to be ported to the x86 architecture, in order to provide a full simulation of a real target machine on a developer workstation. Figure 2.4 shows a block diagram of the affected software components.

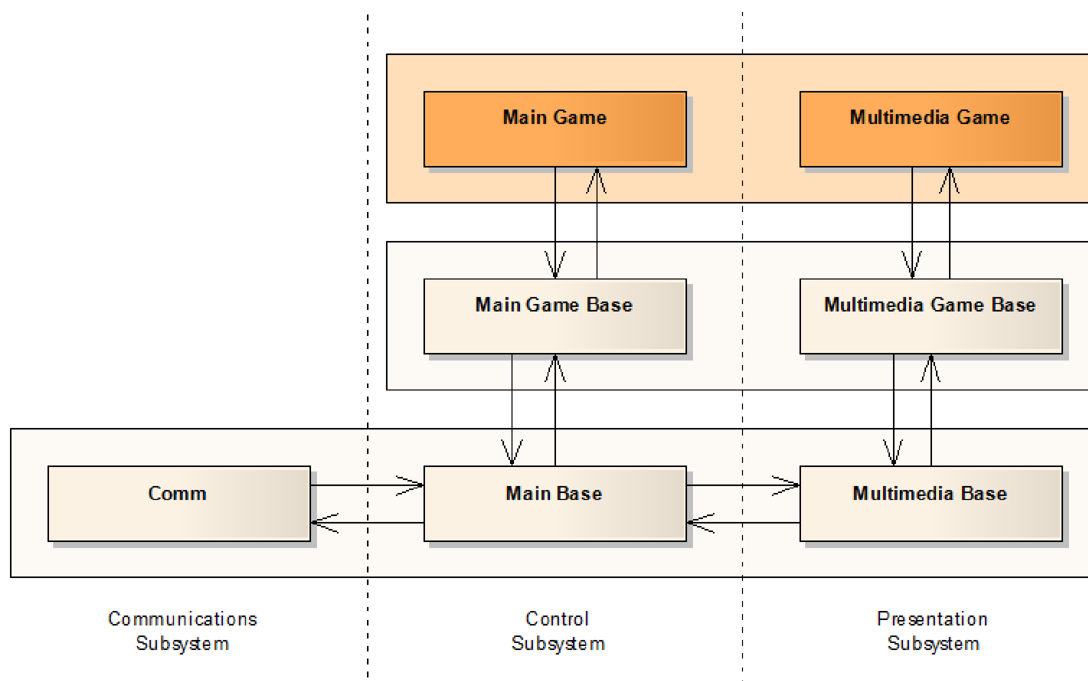

Figure 2.4: An overview of EGD’s software components and their inter-component relationships. Platform-related components are shown in tan color, and individual game content is highlighted in orange color.

Due to various requirements for casino market approval in certain jurisdictions, choosing Windows CE as the target operating system was the only option at that time. How-

ever, the major drawback of this decision was the lack of a Windows CE graphics driver supporting proper hardware acceleration for the platform's GPU; as a consequence, the framework's presentation subsystem – more precisely, the platform component known as the "Multimedia Base" – was conceived as a software-only 2D rendering system, with all necessary low-level functionality such as image blitting or alpha blending written in highly optimized assembly code, hereby utilizing x86-specific SIMD CPU features such as the MMX and SSE instruction set extensions[12].

Influenced by the concepts introduced in Strauss and Carey's original publication and the actual implementation of various available scene graph frameworks, the presentation subsystem was designed as a hierarchical 2D-only scene graph framework, employing various concepts such as different traversal types, instancing and graph paths; however, certain mechanisms were adapted or simplified to account for performance limits and specific needs in the casino gaming market. See Figure 2.5 for a simplified overview of the "Multimedia Base" component as depicted in [9].

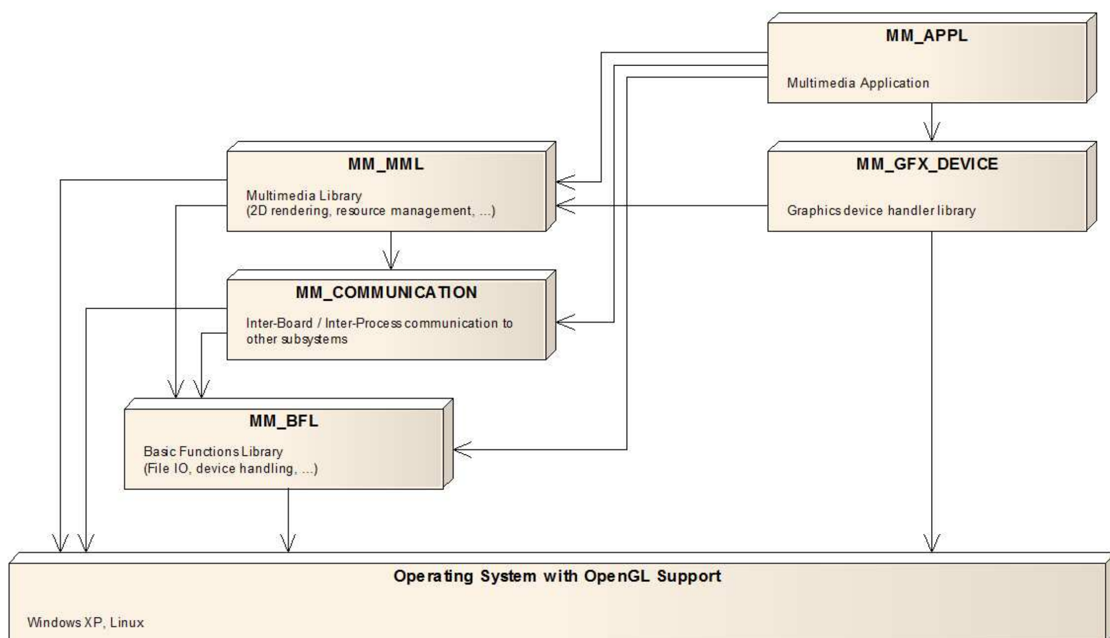

Figure 2.5: Block diagram of the EGD Multimedia base software. Low-level utility and third-party libraries are omitted for clarity.

Binary resources in the form of images, animations, sounds or multi-language text can be accessed via EGD's built-in *resource manager*[18]. All such resources needed for a specific content project must be specified within one or more *resource definition* (RD)

files; the framework supports various individual loader modules for each type of resource, such as PNG or BMP images, WAV and OGG sounds, and a number of proprietary animation formats such as MPNG or DZY[7]. During the development it is possible to directly load resources into the manager from their locations specified in given *resource definition* files; upon delivery, each of these definition files containing individual resources must be compiled into a corresponding *resource package* file that represents a one-to-one mapping of these individual resource files' system memory footprints. This compilation process is carried out by means of the *EGD Resource Packer*, an external command line tool that itself makes use of the resource loading scheme present in the EGD development framework runtime.

Similar to Inventor, the EGD framework allows for building a scene (sub-)graph either from a loadable scene file or via direct instancing from C++ code; reading and parsing of scene files is facilitated through the use of XML. See Listings 2.3 and 2.4 for simple examples.

```
1 <!-- ======== Definition of a content scene ======== -->
2 <BMC_Scene
3   name="BASEGAME SCENE"
4   interface_id="MMI_BaseObjectIds::REELSLOT_SCENE"
5 >
6   <!-- ======== The physical screen where to display ======== -->
7   <MOB_Screen
8     name="BASEGAME SCENE MAIN SCREEN"
9     device_id="GFX_DeviceType::MAIN"
10  >
11    <!-- ======== Instancing of a common object via a reference ======== -->
12    <MOB_ObjectReference
13      shared_object="\SHARED_SCENE\SHARED_SCREEN\SAMPLE_OBJECT"
14    />
15    <!-- ======= Create a new sub-graph instance via an XML entity ======= -->
16    &BaseGameObjects;
17    <!-- ===== Display an animation loaded from the resource manager ===== -->
18    <MOB_AnimationSingle
19      x="0" y="0"
20      animation="resource('sample_animation.dzy')"
21    />
22  </MOB_Screen>
23 </BMC_Scene>
```

Listing 2.3: An XML fragment representing a simple scene to be displayed by the EGD framework

```
1 uint32 MOB_SimpleExample::InitSelf()
2 {
3     uint32 status = MOB_Object::InitSelf();
4     if (status & STATUS_ERROR)
```

```
5         return STATUS_ERROR;
6 
7     // Retrieve an animation asset from the resource manager
8     ASSET_Animation* anim_asset = RES_Manager::GetInstance()->GetAnimation(
9         L"sample_animation.dzy"));
10 
11     // Create an animation node and assign the asset
12     MOB_Animation* anim_node = dynamic_cast<MOB_AnimationSingle*>(
13         BMC_Object::Create(L"MOB_AnimationSingle"));
14     anim_node->SetAnimation(anim_asset);
15 
16     // Create a text object node and define its text
17     MOB_Text* text_node = dynamic_cast<MOB_Text*>(
18         BMC_Object::Create(L"MOB_Text"));
19     text_node->AddText(L"Sample Text");
20 
21     // Create a container node, add the two nodes created as its children
22     // and add the container itself as a child of the current instance
23     MOB_Object* container_node = dynamic_cast<MOB_Object*>(
24         BMC_Object::Create(L"MOB_Object"));
25     container_node->AddChild(anim_node);
26     container_node->AddChild(text_node);
27     this->AddChild(container_node);
28 
29     return status | STATUS_OK;
30 }
```

Listing 2.4: A simple C++ example showing how to create scene graph nodes and link them into a hierarchy.

Analogous to *IRIS Performer*, scene graph traversal in the EGD framework is restricted to three basic traversal types due to performance reasons, carried out in the following order:

- *UpdateContext* traversals update the internal state of all nodes in the scene graph, based on commands sent by the platform's control subsystem via the *Multimedia Interface* (MMI)[17] as well as user input via the touch screen attached to the presentation subsystem.
- *UpdateAnimation* traversals act only on currently active scene graph nodes; here, nodes update their internal state according to given input values for current time and time elapsed since the last iteration.
- *Draw* traversals perform compilation, sorting and rendering of a *draw list* for visible nodes, thereby considering a set of modified drawing regions to update only necessary screen portions.

Sorting the draw list is performed in a back-to-front order, rendering this list is carried out by means of the *Painter's Algorithm*. For that reason, despite being a 2D-only scene

graph framework, coordinates in the EGD framework are internally represented as a three-dimensional vector with its Z coordinate representing the actual drawing depth; objects "closer" to the viewer are represented by higher Z values.

In contrast to Inventor's mechanism for flexible manipulation of node instance properties via field objects and interconnections, the EGD framework implements a stricter scheme also for performance reasons; programmatic node instance manipulation in the EGD framework is performed via specialized *controller object* classes that can be attached to graph nodes implementing a controller-specific class interface. The base implementation provides a number of such controller classes for modifying various node instance properties, e.g. visibility, position, scale factor etc.; however, users are free to implement their own controller classes and node interfaces for more specific purposes.

Eventually, at the time of the first EGD-based platform software and content release in late 2006, the re-written presentation software had proven to be a reliable, extensible and easy-to-use product; up to the current day, all further Atronic platform software releases still build upon that foundation. Moreover, a specially adapted version of the same code base also found its way into back-end casino display systems, hereby replacing any previous solutions that – in direct comparison – suffered from by far inferior usability and performance.

Chapter 3

Goals and Requirements

In mid-2008, Atronic's *Platform Development* management team drew together a number of key-persons, to form a group in charge of identifying and discussing possible future improvements for Atronic's well-established EGD software platform described in chapter 2. Consisting of representatives from a broad range of teams such as various members of the *System Architecture and Requirements* (SAR), *Software Architecture* (SWA) and *Software Development* (SWD) teams within the *Platform Development* department, and members from *Content Development* responsible for game design, graphic design and game implementation, this group soon realized, with the upcoming new gaming machine hardware platform in mind, the importance of providing some kind of hardware-accelerated real-time 3D graphics from a platform software perspective; based on this conclusion, the company decided to actually start the project of extending the EGD framework in that direction. However, with the field of real-time 3D graphics being virgin territory for Atronic, the decision was made to carry out this project in form of an experimental prototype within the scope of this thesis.

To obtain a clear view of the project's goal, the first step in this work was to collect all necessary requirements for further design and implementation, in collaboration with the members of the platform improvements group as well as individual members of the company's design and development teams. The following two sections describe these requirements in detail; with the first section handling user-level requirements mainly collected via input from the actual users of the framework (i.e. *Content Development* members), followed by a section listing system-level requirements that are chiefly influenced by statements from the platform architecture and development teams.

3.1 User-Level Requirements

3.1.1 SDK Extension

Analyzing the past and current situation regarding the use of the EGD framework as a basis for content development, it quickly becomes clear that, for the time being, the framework's SDK is well understood and well established among in-house content developers; however, this achievement was not attained easily after EGD was first introduced. Concerned about any major changes within or new functionality added to the current EGD framework that might lead to a similarly steep learning curve, the whole group agreed on requirements to demand a minimum-invasive approach for any changes or additions to the framework API:

- New functionality shall be provided to the user in a way similar to all existing functionality. More specifically, newly introduced XML scripting nodes shall seamlessly fit into existing scene graphs or easily replace certain parts of it, and the established mechanism of creating higher-level nodes through derivation or encapsulation must be preserved.
- Any necessary changes within existing node class interfaces or class hierarchy shall be transparent to the user; i.e. any user-specific higher-level node classes derived from such classes must be compatible to these changes.

On a more generalized scale, this demand also led to the definition of a fundamental goal in the scope of this work. Contemplating the vast functionality already present in the existing CPU-based 2D implementation for performing basic tasks (e.g. image decompression, animation and video playback, font rendering, or more complex implementations such as *meter objects*), it was decided to aim for a more integral approach instead of considering 3D extensions as an isolated entity:

- The 3D prototype shall make extensive and effective use of functionality already existing in the EGD framework, hereby establishing a combined 2D/3D hybrid scene graph framework.

Aside from the actual framework API, the EGD SDK also provides the user with a number of individual tools to support e.g. image conversion, movie clip editing and resource management and viewing; especially representatives of the latter, such as the *EGD Resource Editor* and *EGD Viewer*, play an important role in the way a developer

provides the framework's central resource management system with any type of binary resource. To take into account any possible new resource types that have to be introduced in the course of developing the prototype, the following requirements were established:

- All new types of binary resource shall be integrated into the existing resource handling mechanism; in addition, resource management tools shall be updated to also incorporate these new resource types.
- The SDK shall provide necessary tools in form of exporters, converters etc. as part of the existing SDK tool chain.

3.1.2 3D Content Representation

Currently, besides the traditional way of creating 2D graphics and animation via imaging and animation tools such as *Adobe Photoshop*, graphic artists make heavy use of 3D modeling applications to create (animated) content scenery, and to render this content in an off-line step to create 2D resources for a specific game. With the artist department being familiar with these applications, specifically *Autodesk** *3D Studio Max* and *Autodesk Maya*, it is an agreed goal to also use them as a means for creating real-time capable 3D content, not at least to avoid costly licensing or trainings. Discussing this goal led to the definition of the following requirements, with additional considerations regarding the actual implementation in EGD-based content:

- The prototype shall be able to handle 3D geometry (in the form of individual objects or whole scenes) exported from any of the modeling applications in use by the Graphic Art department.
- It shall further be possible to also play back any animation stored within such geometry, hereby supporting key-frame animated hierarchical transformations as well as key-frame animated mesh deformations.
- Controlling animation of such objects shall be possible via the framework's already existing animation controllers designed for two-dimensional animation resources.

Besides the existing possibility of creating dynamic 2D content by means of programmatically playing back animation sequences, EGD as a scene graph framework also permits

*http://usa.autodesk.com/

manipulating individual properties such as node visibility, position, rotation etc. in a hierarchical context. The following requirements were defined in order to permit analogous operations also on 3D content, and to provide a common basis for related scene graph controller classes:

- For arbitrary placement and orientation of 3D geometry in scene-space, the prototype shall provide scripting nodes that allow for translation, rotation and scaling of encapsulated geometry objects.
- To allow for programmatic animation, these transformations shall be accessible via the transformation controllers already in existence for moving, rotating and scaling two-dimensional objects.

3.1.3 Coordinate System Alignment

By design, the EGD framework employs a left-handed coordinate system for display: The two-dimensional point of origin is placed in the upper left corner of the screen with the positive X and Y axes running to the right and down directions, respectively; an additional Z axis represents an object's "depth" or painting order in the 2D environment, with its origin in the background image plane and increasing values towards the viewer along a virtual line perpendicular to the screen. To facilitate integration of 3D content into a 2D scene, it is necessary to align the 3D subsystem's coordinate system to that of the 2D renderer.

3.2 System-Level Requirements

3.2.1 Platform Scope

On the system level, the hybrid framework is destined to operate on different hardware platforms and in various fields of application. Taking the following configurations into account, the prototype shall be designed and implemented in a way that allows it to be operated on these platforms without feature restrictions:

- For stand-alone casino gaming machines, the *Synergy* hardware platform running Linux shall be used; technologically, this platform represents the lowest-end target configuration required.

- Casino back-end display systems shall either use the *Synergy* platform or any other suitable off-the-shelf embedded x86 system with same or higher performance, running Windows XP.
- Standard desktop PCs or Laptops with same or higher performance running Windows XP shall be used as developer workstations.

Older hardware platforms, or platforms that do not support hardware-accelerated 3D rendering, need not be considered as a target for operation; nevertheless, on such platforms the hybrid prototype must still be functional in principle, however without support for any of the newly introduced features.

3.2.2 Performance

In direct comparison to off-the-shelf consumer desktop and laptop computers, the embedded *Synergy* platform merely ranks at the lower end of the possible performance scale; this makes it necessary to direct deliberate attention to any issues connected with various aspects that are commonly covered by the umbrella term "performance". Here, one aspect particularly important regarding the end-user – i.e. the casino player – is the system's ability to reliably update the screen contents within a given time interval to display smooth animations without perceptible motion glitches:

- On the *Synergy* platform, the framework prototype shall be able to maintain a constant frame rate of 60 fps during normal operation; this excludes corner cases such as switching between scenes, or rendering scenes with a high load of text objects rendered with high-quality fonts.

To account for the fact that the platform's single CPU board must also run both the control and communication subsystems in parallel to the presentation engine, and in consideration of keeping a reasonable amount of resources in reserve for future extensions, the following requirements were formulated:

- The load on a single CPU core shall not exceed the 50 percent mark; again considering the corner cases mentioned above.
- The prototype's hybrid rendering system shall make optimal use of CPU and GPU resources, effectively implementing a parallel CPU/GPU rendering scheme as far as possible.

3.2.3 System Integration

An important goal in developing the prototype is not only to smoothly fit in the extended functionality into the framework's API at the user level, but also to maintain the consistency of the framework's logical structure in terms of modularization and strictly defined interfaces and dependencies between these modules on the system level:

- All of the newly introduced functionality dealing with hardware-accelerated rendering shall be encapsulated into a separate loadable module; as stated above, on any platforms not supporting the mechanisms used therein, the hybrid prototype shall still be able to function without loading that module.

Having successfully narrowed down the scope of the project to a manageable extent as a result of this thorough definition of necessary requirements, the next step was to ensure overall technical feasibility within that scope. The next chapter presents a brief study on this topic, depicting preliminary performance measurement results for a number of important corner cases; these results then form another key factor for further design and implementation work.

Chapter 4

A Technical Feasibility Study

As stated in chapter 2, the EGD framework evolved from the need for a software base suiting the more advanced capabilities of the then-new "Hi!bility" hardware platform. Before the EGD framework was available, the only way for developers to write software for the platform was to run a cross-compiler on a PC, upload the compiled binaries to a Windows CE target machine and execute them there. As this proved to be quite a cumbersome and time-consuming process, it was decided to design the framework in such a way that there is an easy possibility to develop and run a simulation of the "real" software on a Windows PC.

Having laid the foundation for portability by making this design decision, the framework has actually been ported to several different other platforms, comprising various hardware architectures and operating systems:

- The original "Hi!bility" platform, with two 68k-based circuit boards for game control and external communications running the OSE operating system, and a customized PC mainboard for presentation running Windows CE, equipped with an Intel Celeron CPU running at 566MHz and an ATI Radeon Mobility M6 AGP graphics card.
- The "Oxygen" platform, which is also based on a multi-board architecture, with two PowerPC-based boards for game control and communications, and a custom presentation board based on the Intel i855GM chipset with integrated GPU, equipped with a Pentium M CPU at 1.6GHz. All three boards use Linux as their operating system.
- The "Sensys" platform, which represents Atronic's first platform based on a single-board architecture; the custom all-purpose board is running under Linux

and equipped with an Intel i915GM chipset with integrated GPU and an Intel Core2Duo CPU at 2.2GHz.

- The "Synergy" platform, being the direct successor of "Sensys", with the most noticeable difference in the more modern i965GME chipset used.

- Any standard PC running Windows 2000 or Windows XP.

Whereas the first four platforms primarily operate in stand-alone casino gaming machines, the standard PC option is not only used for development purposes, but also the choice for back-end systems driving large overhead displays for e.g. linked gaming or jackpot solutions [8]. Here, a broad range of off-the-shelf industrial or embedded x86 systems is utilized, with different CPU and GPU configurations varying from low-cost to high-end.

For bringing graphical output on screen, the presentation engine of the existing framework makes use of specific implementations for each operating system it is compiled to be executed on; Windows 2000/XP versions make use of *DirectDraw* surfaces, whereas the Linux implementation is based on accessing the *Framebuffer Device* native to the X Window system.

Clearly, the goal of extending the EGD framework to support hardware-accelerated 3D rendering primarily on the *Synergy* platform also demands a system-level API that does support hardware acceleration on all operating systems in question. The fact that there are actually only two candidates in existence that come into consideration for full support of consumer-level hardware and operating systems – *Direct3D* as part of Microsoft's *DirectX*, and OpenGL – led to the inevitable decision to go with OpenGL , as DirectX is proprietary to Microsoft operating systems, and OpenGL is a well-established, well-supported and well-performing standard across all required operating systems.

The following section describes necessary pre-conditions that have to be met to allow for efficient hardware-accelerated rendering in the given 2D/3D hybrid environment. Section 4.2 gives an overview of certain limitations and problems of the given platforms and OpenGL / graphics driver implementations that were encountered during the evaluation phase, which may have impacts on overall performance. Finally, section 4.3 presents a summary of preliminary performance measurements carried out on the "Synergy" platform and conclusions drawn from these results.

4.1 Pre-Conditions

4.1.1 2D Surface Pixel Transfer

When displaying two-dimensional content made up of a fair number of animations, the Atronic EGD framework with its software-only rendering system produces high amounts of 32bpp RGBA pixel data in system memory for each successive display frame. In order to achieve an acceptable display frame rate, the platform must provide a fast method for transferring these amounts of system-memory surface data to GPU-accessible memory. In the present implementation of the Atronic EGD framework, this task is accomplished by transferring pixel data directly to the back buffer of the primary screen, using *DirectDraw* surfaces under Windows and the *Framebuffer Device* under Linux. This functionality is also provided by OpenGL, since version 1.2 direct RGBA frame buffer writes are supported using the API function `glDrawPixels()`.

For conventional 2D-only content, drawing pixel data directly to the frame buffer proves to be fairly adequate. For hybrid content however, this may not be sufficient; once the pixel data end up in the frame buffer, no spatial transforms can be applied anymore. Considering possible visual effects such as zooming, tilting or otherwise manipulating the software-rendered background surface – effectively making it a *part* of a three-dimensional scene – requires the surface to first be loaded into texture memory and afterwards be drawn to the frame buffer via a textured mesh object; using a screen-aligned quadrilateral here yields the same effect as a direct frame buffer write. Since version 1.2, OpenGL has allowed for the creation and update of two-dimensional RGBA textures, using the API functions `glTexImage2D()` and `glTexSubImage2D()`, respectively.

In the attempt of reusing as many parts of the existing framework implementation as possible also for rendering 3D content, it is an obvious technique for any 2D image generated by a scene sub-graph to be used as a 2D texture applied to a 3D object, whether it be a simple static image, a movie clip or a more complex animation made up of a composition of objects. Whereas static images do not necessarily require highly optimized data transfer to a video memory texture, this is an essential requirement for the latter two examples due to their highly dynamic nature.

Since version 2.1, the OpenGL specification [10] has provided an additional mechanism that may be used – depending on the actual graphics driver implementation – to speed up data transfer to the GPU's frame buffer or texture memory, commonly known as *Pixel Buffer Objects* (PBO). This mechanism allows creating and allocating memory buffers

directly in GPU-local video RAM; these buffers may then be temporarily mapped to client memory for easy read/write access by the CPU, possibly taking advantage of hardware-accelerated memory transfer (DMA-Blits).

4.1.2 Geometry Rendering

One crucial aspect of creating visually attractive output in a three-dimensional real-time environment is the system's capability of effectively handling high amounts of geometric data, i.e. individual triangles or -strips, standard geometric shapes such as cubes or spheres, or more complex geometry arranged in a mesh. In addition, it is also desirable not to restrict rendering only to static geometry; the system should also be capable of handling e.g. key-frame animated mesh objects. Traditionally, as stated by Strauss and Carey [16] and observable from various existing scene graph frameworks, two approaches to feeding geometry to the rendering pipeline exist:

- In *Retained Mode* frameworks, prior to rendering, each geometry object is compiled into an internal data structure or buffer (such as a *display list*) that is kept in parallel to the original geometry data in the underlying data base; this way, the rendering subsystem may gain access to the data to be rendered in a way that allows for optimal rendering performance. This approach works well for static geometry; on the other hand however, highly dynamic data sets such as animated meshes require either a high number of extra resources (e.g. one display list for each animation frame), or the – very costly – effort of repeatedly re-creating or updating a single retained buffer, at worst once per frame. The issue of having to manage an additional copy of the geometry data, and having to keep those copies in sync, is also commonly referred to as the *Duplicate Database Problem*.

- *Immediate Mode* implementations render geometry by submitting geometric primitives (triangles, triangle strips or -fans) from the data base individually to the pipeline. The nature of this approach makes it relatively easy to handle dynamic data sets between individual display frames; as no internal copy of the geometry is kept, possibly a high amount of resources can be saved for other purposes, and there is no need to update a retained buffer prior to drawing. On the downside, as the rendering subsystem usually has no direct access to the geometry data base, this approach imposes a high workload on the CPU feeding individual primitives and thus usually bypasses any advanced mechanisms present in the GPU for high-

performance geometry rendering, resulting in the CPU having less resources to perform other (possibly also time-critical) tasks, and an all-to-low maximum polygon count per frame.

From a performance perspective, taking into consideration the overall capabilities and limitations of the target platforms in question, a retained mode approach appears to be the lesser of those two evils in the design of the Atronic EGD framework extensions, despite the drawbacks mentioned above.

Fortunately, as of version 1.5, OpenGL provides an additional means for performance-critical geometry rendering; the concept of *Vertex Buffer Objects* (VBOs) – analogous to Pixel Buffer Objects – introduces the possibility to manage data buffers to hold geometry data in GPU-local memory, additionally offering a method to dynamically update previously created buffers with minimal overhead. Although making use of this concept does not actually eliminate the duplicate database problem, it at least facilitates mitigating the problems of a retained mode approach when dealing with dynamic geometry.

4.1.3 Offscreen Rendering

Based on the above described technique of using the output of a two-dimensional subgraph as a texture on a 3D object, it is also desirable to not restrict texture targets only to software-rendered surfaces, but also to allow the hardware-accelerated renderer itself to generate images to be used as textures, or possibly again a hybrid of both mechanisms. For that purpose, it is necessary that the GPU is able to render its output to an off-screen color buffer with attached depth buffer, which in turn gets uploaded into a texture object.

OpenGL versions prior to 3.0 do not directly provide an efficient platform-independent way to handle such situations. However, there exists the widely implemented `GL_EXT_framebuffer_object` [14] extension, which introduces so-called *Frame Buffer Objects* (FBOs) that serve as alternative off-screen targets. By binding a *Render Buffer* to an FBO's depth component and a texture object to its color component, this extension allows to directly perform the desired off-screen rendering task.

4.1.4 Geometry Representation

The necessity of dealing with three-dimensional content designed in a modeling application brings up the problem of exporting these data to a common file format, without sacrificing essential features that are often also unique to a specific application. Based on

the requirements defined in chapter 3, an adequate file format must support at least the following features:

- The format shall allow to hierarchically store multiple geometric objects within one file, to be able to either represent single objects or whole scenes.
- Material properties assigned to these objects at design-time shall be preserved, with the possibility to uniquely identify each material and to group together identical ones.
- The file format shall be able to store normal vectors as well as tangent vectors on a per-vertex basis, to allow for state-of-the-art shading techniques such as *DOT3 Bump Mapping*.
- Animation shall be possible, allowing for key-framed parent-child transformations as well as key-framed mesh deformation; for the latter, vertex coordinates, normal vectors and tangent vectors must be considered.

Among the vast amount of available file formats for representing 3D content, the two most promising candidates soon emerged during the analysis of the two modeling applications mainly used at Atronic – *3D Studio Max* and *Maya* – regarding their range of supported export formats: *VRML97* [3] and *FBX* [2].

Essentially based on *Inventor*'s original human-readable ASCII scene graph definition syntax, *VRML97* was conceived as a standard file format for describing 3D objects and scenes with a publicly available specification; reading and parsing of these files is facilitated through a number of freely available parsers that may easily be integrated into existing software projects on the source code level. Offering a comprehensive set of features, *VRML97* fulfills almost all of the above mentioned requirements, with the exception of storing tangent vectors. Due to the open and extensible nature of file format and parsers though, this problem may be remedied by adding custom extensions to the file format; however, the applications' plug-ins are only available as compiled binaries and thus do not allow for adding these extensions to the exporter side.

FBX in contrast was defined by *Autodesk* as a proprietary file format in an attempt to unify and simplify the usage of 3D data exported from their aforementioned products; for that purpose, the company offers a freely available suite of plug-ins for these applications that allow for exporting scene and model data to *FBX* files. Although this file format in its binary form is not publicly documented, reading and parsing of such files by a third-party

application is rendered possible through the use of the associated *FBX SDK*, which is also provided by Autodesk on a free basis in the form of pre-compiled binaries for Windows and Linux operating systems. This format also fulfills all the requirements above except tangent vector storage; here, due to the closed-source nature of the SDK and individual plug-ins, it is nearly impossible to enhance the file format by this feature.

For both described file formats, the missing tangent vector storage possibility must be considered in the prototype design, together with a number of other drawbacks that manifest themselves in a practical environment despite the formats' decent set of features:

- *VRML97* files and ASCII *FBX* files may become inconveniently large, especially considering complex geometry with detailed key-frame animation.
- Loading and parsing of these files may take a considerable amount of time, hereby imposing noticeable delays when a player switches between individual games.

4.2 Graphics Driver Analysis

Naturally, when dealing with a number of different hardware platforms (incorporating various different GPU brands and models) dedicated to run a certain software application on a variety of operating systems, it is also necessary to take into account the availability and suitability of graphics drivers supporting any of those hardware/OS combinations. Here, especially in the light of an application in the field of embedded computing, such drivers basically have to fulfill two essential requirements: *stability* and *performance*. In addition, to allow for a common application code base and identical behavior on all platforms, all of these drivers in question ought to have a common OpenGL API version to rely on.

4.2.1 OpenGL API Versions

Regarding the plurality of available graphics drivers for the various hardware platforms in question, the diversity of supported OpenGL API versions among these drivers poses a significant issue. While the most recent driver versions for mid-range to high-end GPUs such as the *nVidia GeForce* or *ATI Radeon* series provide – even under Linux – a fully compliant OpenGL 2.1 interface to rely on, this situation becomes more complicated when dealing with Intel chipsets with integrated GPUs such as the i915 or i965. Typically built into low-end desktop workstations and laptops, these latter chipsets have seldom

been noted for their 3D graphics capabilities, even though especially the i965 and its successors do provide at least acceptable performance in that domain. Only recently, with the increasing demand for hardware-accelerated 3D features also on operating system level (e.g. *Windows Aero*[*] or *Compiz Fusion*[†] desktop effects), driver development has started to catch up to support these chipsets' features more thoroughly.

Despite these efforts, the drivers in question still have not reached the desired OpenGL version level of 2.1 at the time of writing, even though certain essential API functions mentioned in section 4.1 are at least available through OpenGL's extension mechanism on some of the drivers. As a consequence, a comprehensive design supporting all desired platforms must provide sufficient abstraction between higher-level functionality, such as texture handling or geometry rendering, and low-level OpenGL API calls; missing or incomplete OpenGL functionality must either – transparent to the user – be emulated using other algorithms or techniques or made unavailable on certain platforms.

4.2.2 OpenGL Performance

Looking at the OpenGL API documentation in theory, performing the operations mentioned in section 4.1 seems to be a straight-forward task. In practice however, there are a number of issues to be dealt with, which may have more or less severe impacts on overall performance. These issues may arise from incomplete or sub-optimal driver implementation as well as functionality not supported in the underlying graphics hardware being emulated in software:

- The OpenGL API defines three distinct 32bpp RGBA modes with different byte ordering: `GL_RGBA`, `GL_BGRA_EXT` and `GL_ABGR_EXT`, with `GL_BGRA_EXT` matching the pixel format native to the Atronic EGD framework. Choosing the "wrong" OpenGL pixel format during the analysis resulted in performance drops on various hardware configurations; this indicates that the driver must carry out a per-pixel format conversion during transfer. Fortunately, on all evaluated drivers, the framework's native format yields optimal results.
- Although all target platforms in question support fast *direct memory access* (DMA) transfers from system memory to video memory via the AGP or PCIe interface, not all drivers make use of this efficient mechanism. Instead, on these drivers, all data transfers are done by the CPU, effectively preventing it – or at least one core

[*] http://www.microsoft.com/
[†] http://www.compiz-fusion.org/

if dealing with a multi-core CPU – from executing other tasks for a considerable amount of time.

- Some driver implementations overcome this problem by allowing DMA-accelerated access to OpenGL Pixel Buffer Objects (PBO), which get temporarily mapped to client memory space so that the application may directly update a region in video memory. In the attempt of efficiently using such a PBO as the source for an OpenGL texture, the driver may choose to directly use this memory region as the actual texture object without any data transfer; however ("zero-copy upload"), such a scheme is generally prone to stalling the GPU pipeline when the CPU is trying to update a buffer during rendering. Implementing a double-buffered scheme for PBO texture updates solves this problem, however at the cost of doubled PBO memory consumption.

- When updating an OpenGL texture, certain drivers show a significant performance discrepancy between a call to `glTexImage2D()`, which always updates a texture as a whole, and `glTexSubImage2D()`, which may be used to perform also a partial update via a sub-rectangle within the texture surface area. Specifically, the latter sometimes shows far worse performance on some drivers than the former due to a CPU-intensive software fallback implementation.

Again, regarding the diverse behavior of the various GPU/driver setups, a comprehensive design must provide a well-implemented abstraction layer to hide internal OpenGL call details from the user; specific code paths for handling texture updates in an optimal way – with or without PBOs, either using `glTexImage2D()` or `glTexSubImage2D()` – must be considered depending on each driver's capabilities.

4.2.3 Graphics Driver Evaluation

In practice, available graphics drivers for a specific hardware/OS combination often do not only differ in their unique version history. Specifically regarding platforms equipped with an Intel chipset with integrated graphics such as the i915 or i965, research on this topic brought up a number of options among which the most suitable driver implementation had to be chosen. Following below is a detailed record of all these drivers that were evaluated with regard to OpenGL API version, performance and stability.

4.2.3.1 nVidia driver for GeForce/Ion chipsets

As a vendor of GPUs in the highly competitive PC gamer market, which is primarily defined by high-performance and high-quality 3D graphics, nVidia[‡] is not only under constant pressure to develop new leading-edge hardware products, but is also forced to provide adequate drivers that support these products' hardware capabilities in an optimal way. With full support of OpenGL version 2.1, evaluation of the current version of the driver on various supported GPU models has shown that it fulfills all necessary pre-conditions listed in section 4.1; it also provides the stability and performance necessary for unrestricted operation. In addition, due to the driver's common closed-source code base, the same driver quality is maintained over both operating systems, Windows and Linux, with the Linux version being only one or two months behind in the release schedule.

4.2.3.2 ATI Catalyst driver for Radeon chipsets

Being the primary competing company to nVidia, AMD/ATI[§] face the same situation. Evaluation of the current driver for their "Radeon" chipset family also yielded optimal results regarding OpenGL support, stability and performance; both Windows and Linux operating systems are supported through a common code base.

4.2.3.3 Intel GMA Driver

The *Intel Graphics Media Accelerator Driver*[¶] is generally the first choice to operate off-the-shelf desktop workstations and laptops equipped with an Intel-based GPU running Windows XP or any of its successors. As stated earlier, 3D graphics applications on such systems have somewhat been neglected in the past; in supporting only the OpenGL 1.4 API, the GMA driver more than slightly lags behind the most recent drivers from nVidia or ATI, which commonly support OpenGL 2.1. Being at least able to support efficient geometry rendering using VBOs through the OpenGL `ARB_vertex_buffer` extension, the driver still lacks an efficient mechanism for pixel data transfer; texture upload performance measurements via `glTexImage2D()` and `glTexSubImage2D()` showed sub-optimal performance with a high CPU load, indicating that the driver does not make use of DMA transfers. Being a well-tested and well-established product, the GMA driver provides good long-term stability; however, it is only available on Microsoft Windows operating systems.

[‡] http://www.nvidia.com/
[§] http://ati.amd.com/
[¶] http://www.intel.com/support/graphics/

4.2.3.4 Intel Embedded Graphics Driver (IEGD)

As an alternative to the Windows-only desktop driver, Intel offers a second closed-source driver package for their GPU families especially for applications in the field of embedded computing. Known as the *Intel Embedded Graphics Driver* (IEGD)[‖], this driver is available for a variety of different operating systems, such as Windows XP, Windows XP Embedded, Windows CE and Linux. Being a particularly interesting option for the "Synergy" platform running Linux, this platform/OS combination was the main focus during the evaluation of this driver; however, several OpenGL-related drawbacks were encountered throughout this evaluation:

- Although, for the platform's i965 chipset, texture upload via `glTexImage2D()` and `glTexSubImage2D()` shows acceptable transfer throughput from system memory to the OpenGL subsystem, these operations impose a heavy load on the CPU and thus appear to not make use of the chipset's DMA feature. As the OpenGL support is basically limited to API version 1.5, and there is no support for PBOs through an OpenGL extension, there is also no further option to remedy this issue.
- The alternative fall-back mechanism of transferring a software-rendered surface directly to the primary screen's back buffer via `glDrawPixels()` proved to be more efficient; however, due to an obviously erroneous driver implementation, the driver performs any other OpenGL rendering operations *in parallel* to the transfer operation, resulting in an overall corrupted output image. Furthermore, all attempts to resolve this problem by systematically inserting `glFlush()` or `glFinish()` calls to synchronize the rendering pipeline were unsuccessful.
- According to the specifications, using VBOs for rendering geometry objects is fully supported. In practice however, the driver proved to be extremely unstable when doing so; overloading the rendering pipeline with too many VBO drawing calls per frame frequently resulted in crashes of the entire X Window system.

In addition to these severe defects, the underlying 2D driver also tends to lose synchronization to the *vertical blank* interrupt, resulting in a rather jolting display appearance. Altogether, this driver – in its current 9.1.1 version – does neither provide sufficient stability nor performance for undisturbed operation.

[‖] http://edc.intel.com/Software/Downloads/IEGD/

4.2.3.5 Intel Open Source DRI drivers for Mesa3D

Starting out in 1993 as a software-only implementation of the OpenGL API, the *Mesa*[**] 3D graphics library emerged into an open-source cross-platform product that nowadays also provides hardware-accelerated graphics for various GPU vendors and models, especially through connection with the *Direct Rendering Infrastructure* (DRI)[††] API that is natively provided by Linux and other operating systems. Particularly after Intel has released the full hardware documentation for their line of GPU products to the public, these chipsets have been well supported in Mesa through specific driver module implementations.

Although Mesa – as of version 7.4.4 – fully implements the OpenGL 2.1 API using its built-in software renderer, the availability of certain 2.1 features still depends on the actual existing driver for a specific GPU. Regarding Intel's i965 chipset, evaluation of this driver has shown that it does support both the VBO and PBO mechanisms; however, it is necessary to utilize a double-buffered scheme here to prevent the GPU pipeline from stalling when updating such buffer objects. Texture updates via `glTexImage2D()` in conjunction with PBOs perform well with sufficiently low CPU overhead; as a downside, partial updates via `glTexSubImage2D()` have shown to be not hardware-accelerated at all, whether PBOs are used or not. However, this drawback is somewhat mitigated by the fact that the driver does support the use of non-power-of-two texture sizes on the API level.

Evaluation in terms of stability proved to be sufficient as well; a two-week duration test on five target machines produced no crashes and no unexpected behavior.

4.3 Preliminary Performance Measurements

For the purpose of measuring pixel data transfer throughput, a small GLUT-based application was developed for Windows and Linux that is able to repeatedly (once per display frame) fill a system memory region with a changing color pattern and to update a previously generated 2D texture in video memory by sourcing this memory region. Transfers were measured for both `glTexImage2D()` and `glTexSubImage2D()`; either function was analyzed with respect to its performance depending on three different PBO configurations:

- No PBO used: Texture transfers are carried out directly.

[**] http://mesa3d.org/
[††] http://dri.freedesktop.org/wiki/

- Single-buffered PBO: The PBO used for texture update is directly used for rendering immediately after the transfer.
- Double-buffered PBOs: Texture update is performed in the back-buffer PBO; rendering is also performed immediately after the transfer, however using the texture object holding the transferred data from the previous frame now in the front-buffer PBO. This way, the driver is granted extra time for e.g. performing a delayed transfer of the actual data. Upon a frame buffer swap also these double PBOs are swapped.

It should be noted that, as this texture is also rendered on screen after its update, and due to possible internal overhead in the graphics driver and/or windowing subsystem, the results of these measurements do not directly reflect the maximum possible transfer rate; instead, the resulting FPS number represents a fairly reasonable transfer speed indicator in a practical environment.

Measurement results for a 32bpp RGBA texture at a resolution of 1024x1024 texels displayed in a full-screen window at a resolution of 800x600 pixels can be seen in Figures 4.1 and 4.2; the former displays the maximum achievable frame rate for various graphics driver configurations on the *Synergy* platform, whereas the latter shows the load percentage of a single CPU core when clamping the frame rate to 60 FPS. The configurations shown include all three relevant drivers for the i965 chipset; for comparative purposes, also the results of the ATI Catalyst driver running on a Mobility Radeon HD 2600 PCIe laptop graphics card are shown.

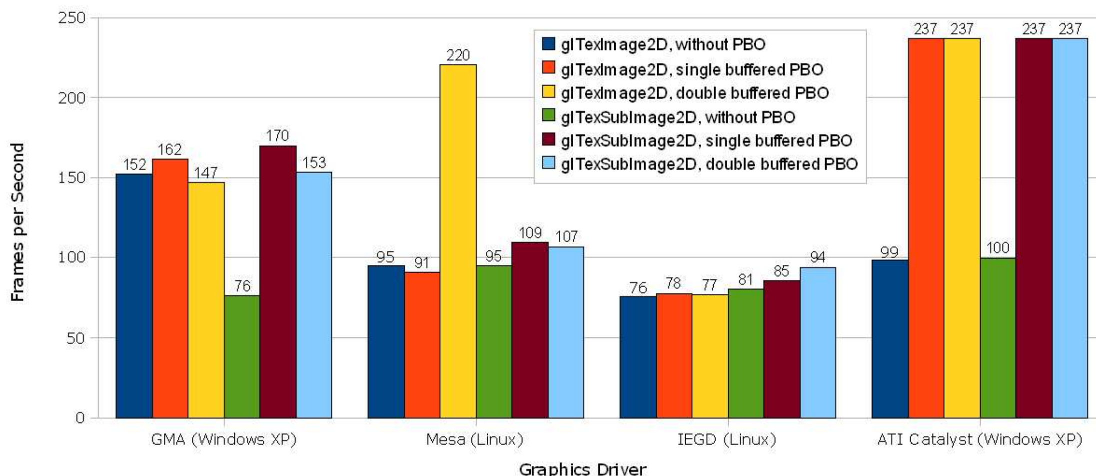

Figure 4.1: Texture update performance: Maximum achievable frame rate for various driver configurations.

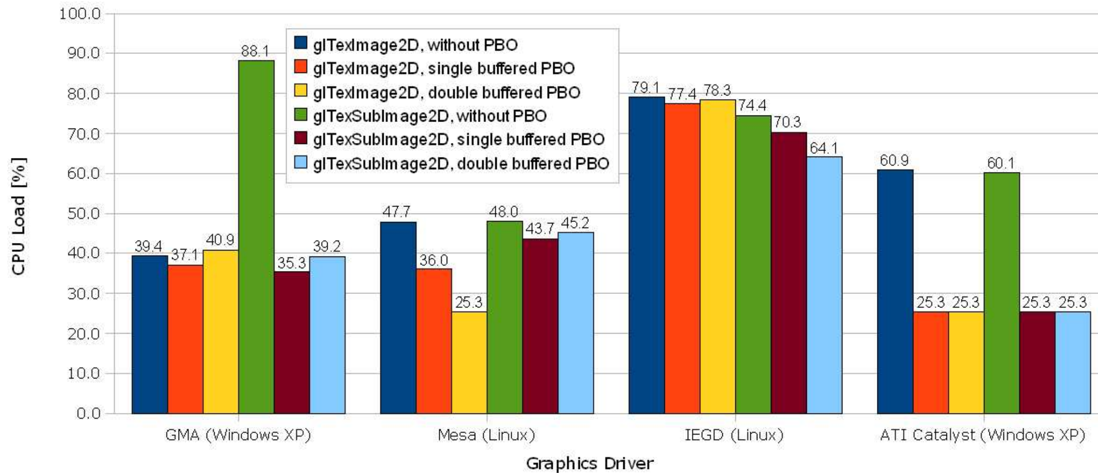

Figure 4.2: Texture update performance: CPU load (single core) for various driver configurations at a fixed frame rate of 60fps.

4.4 Conclusions

Evaluation and performance measurements of the various available graphics drivers showed that the desired target platform/OS configurations more or less fulfill the necessary preconditions from section 4.1, and thus it becomes feasible to design and implement an efficient hybrid 2D/3D system on top of the EGD framework, following the requirements from chapter 3. Nevertheless, due to the partly sub-optimal OpenGL support and certain quirks especially in the graphics drivers available for Intel GPUs, such a system must be designed to abstract away these matters:

- The OpenGL API version common to all evaluated drivers is 1.5; the system must be able to detect and use all higher-level functionality via extensions.
- Pixel data transfer to an OpenGL texture must be considered to either operate directly or by using a single-buffered or double-buffered PBO scheme, depending on the available graphics driver detected at run-time
- Texture updates must also take into consideration the performance discrepancies between the API functions `glTexImage2D()` and `glTexSubImage2D()`; again based on a run-time driver detection

Considering the various target platforms supposed to run the extended EGD framework, the following conclusions can be drawn from the driver evaluation results:

- For the ”Sensys” and ”Synergy” platforms running Linux, the only feasible option is the open-source Mesa 3D library together with its compatible i965 DRI driver.
- Developer workstations and overhead display controllers running under Windows should be equipped with nVidia or ATI GPUs; Intel i965-based platforms with the standard GMA driver currently available are bound to severe restrictions.

Chapter 5

Concepts and Prototype Design

Having laid the foundation for transforming EGD into an efficient hybrid 2D/3D scene graph framework through the goals and requirements drafted in chapter 3 and the preliminary results of the graphics driver performance and stability evaluation in chapter 4, this chapter describes the necessary concept work and design decisions made, as a basis for the actual implementation.

5.1 2D/3D Interaction

In the Atronic EGD framework, a scene graph basically consists of a tree of `BMC_Node` instances or nodes derived from that class. All visual parts of the scene graph must be defined within the sub-graph of a `MOB_Screen` instance, which restricts all its children to be instances of the `MOB_Object` class or any derivative. Through this restriction, a number of architectural options emerged during the design process for how to integrate three-dimensional content into such a graph.

The first approach that was taken into consideration was to keep the aforementioned child restriction, and to satisfy it by deriving any OpenGL-related classes also from `MOB_Object`. Analyzing the influence of this approach on the overall architecture brought up the problem that any of these derived classes would inherit all methods and members of the `MOB_Object` base class, among which a high number is not applicable to OpenGL-based rendering, such as 2D object metrics and handling of modified regions on a 2D surface. This would result in a significant undesired waste of resources given a high number of OpenGL-related nodes in a scene graph, therefore this approach was discarded.

The second approach was to remove the restriction – or at least mitigate it by restricting

to only a common base class, namely BMC_Node – and thus allow a fully heterogeneous scene graph in terms of mixing MOB nodes and OpenGL-related nodes. The downside of this approach however would be that on the one hand the existing safety mechanism in effect through that very restriction would be nulled, and on the other hand it would pose a significant intrusion on the existing framework implementation. Being classified as too risky, this approach was also discarded.

5.1.1 3D Renderer Areas

In the final approach, which is the basis for this work, the concept of a *renderer area* for rendering 3D content has been developed (not to be confused with *Inventor*'s *RenderArea*[15]). A *renderer area* in the Atronic EGD framework basically represents a "window" or "viewport" to a three-dimensional scene – i.e. a sub-graph consisting of OpenGL-related nodes – within the hierarchy of a two-dimensional scene graph built of distinct MOB_Objects. Unlike the traditional understanding of a viewport, which usually defines the area where a scene is rendered from a windowing subsystem's point of view, a MOB_RendererArea exists *within* the scene graph hierarchy and thus directly inherits various properties from its parent, such as visibility, 2D position, drawing order etc. Generally speaking, a renderer area represents some sort of a "switcher node", which allows switching from a 2D parent node to a 3D sub-graph within an otherwise homogenous scene graph. The child restriction on MOB_Object instances remains intact; additionally, a similar restriction is enforced on any newly introduced OpenGL-related classes to also ensure a homogenous sub-graph. See Figure 5.1 for a simple example.

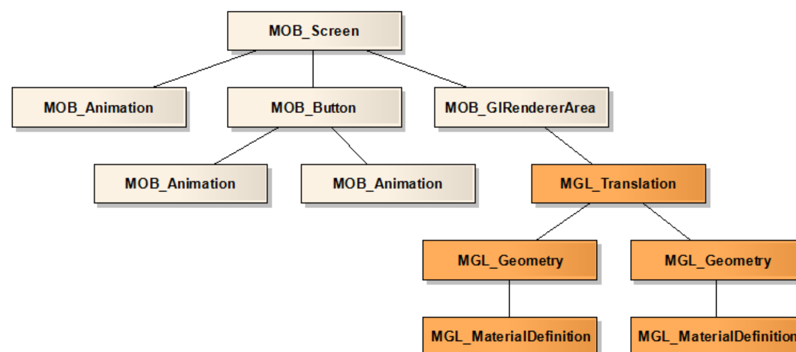

Figure 5.1: An exemplary fragment of a quasi-heterogeneous scene graph consisting mainly of 2D nodes (tan color), with additional 3D nodes inserted (orange color) by means of a switcher node (MOB_GlRendererArea).

Analogous to the existing hierarchy of scripting nodes, with all higher-level nodes derived from the `MOB_Object` base class, these newly introduced OpenGL-related node classes are also derived from a common base class, `MGL_Object`; all of its descendants can be identified by their "MGL" prefix. A specialized implementation of the abstract `MOB_RendererArea` class, `MOB_GlRendererArea` provides the connection between "MOB" nodes and "MGL" nodes in the MM_OPENGL library.

5.1.2 Texture Targets

As can be seen in Figure 5.2, the switcher node concept can also be applied in the opposite direction, i.e. switching back from 3D to 2D within the scene graph. One obvious field of application for such a switcher node is a *texture target*, which is essentially an OpenGL-related node encapsulating a reference to an OpenGL texture object that may be drawn to using the existing framework's 2D rendering engine. Making use of the framework's existing 2D rendering engine and its internal mechanism for handling modified surface regions, this provides an easy way to not only load static image resources into a texture, but also to perform a per-frame dynamic texture update for playing movie clips or for rendering more complex two-dimensional sub-graphs that represent application- or user-controlled animation.

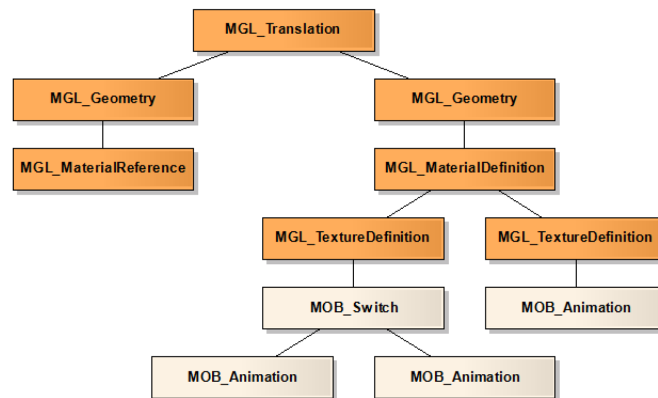

Figure 5.2: A fragment of a quasi-homogenous 3D scene graph showing how to use the 2D rendering engine for effectively generating dynamic textures by means of a `MGL_TextureDefinition` node.

5.1.3 Renderer Engine

Unlike in e.g. a *Binary Space Partitioning tree* (BSP tree), a scene graph is not organized in a way that allows to implicitly sort its geometry during traversal. In addition, traversal of a scene graph in the Atronic EGD framework is generally done in a depth-first manner, i.e. the root node is processed first, followed recursively by traversing its children from left to right. As a result, when a scene graph's affected MGL scripting nodes were rendered directly upon traversal, it would not be possible to perform any sorting, grouping or certain other pre-processing steps. Depth sorting, however, is required to maintain a correct back-to-front drawing order for rendering (semi-)transparent objects with alpha blending enabled; low-performance GPUs may also benefit from a front-to-back drawing order for rendering opaque geometry in an attempt to minimize overdraw, or from grouping together geometry sharing the same material to minimize internal state changes.

IRIS Performer overcomes this limitation by introducing a "DRAW" traversal that does not operate on the actual scene graph; instead, it processes a simple display list of state changes and geometry objects that is generated by a preceding "CULL" traversal, where any culling, depth sorting and state sorting takes place. A similar approach was chosen for extending the Atronic EGD framework: Here, MGL scripting nodes may contain any number of internal renderer objects that are derived from a common base class (`OGL_Object`); upon drawing traversal, an MGL node's internal objects are enqueued to the respective renderer area for later processing. See Figure 5.3 for a simplified illustration.

A central role in creating, managing and destroying these internal renderer objects is played by the `OGL_RendererEngine` class; each physical `GFX_Device` instance registers its own individual renderer engine upon initialization. Through this design, all `MOB_Screen` instances in a given scene graph that share the same device also gain access to the same renderer engine. The central renderer engine class also provides an interface for creation and destruction of individual renderer areas defined within those screens; this inter-connection makes it possible to share and reference renderer objects across multiple renderer areas.

In general, referencing of renderer areas and objects is accomplished through the use of *named instances*. Creation of such a named instance can be performed in a scene graph via an according *definition node*, which allows specifying a unique identifier for that node's underlying renderer object. A specific object may then be referenced via this unique ID by specifying a *reference node* in the scene graph below.

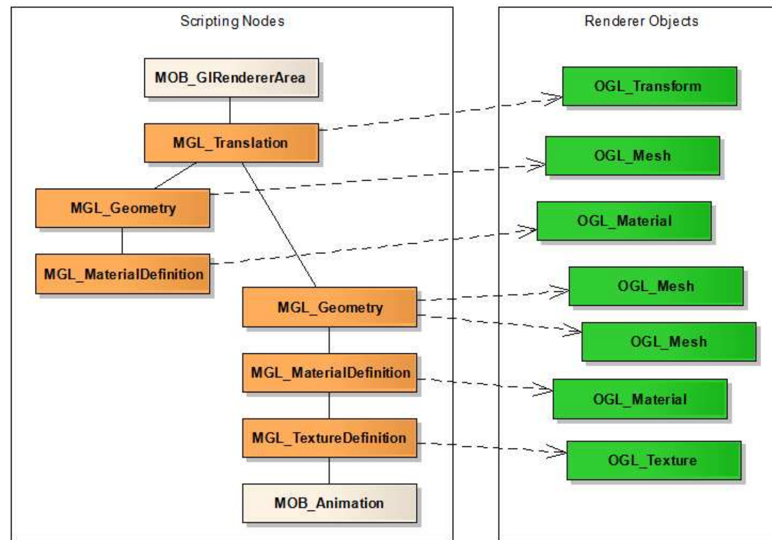

Figure 5.3: A scene graph fragment depicting the relationship between MGL scripting nodes and their underlying renderer objects.

5.1.4 Parallelization

Although the 2D rendering process in the EGD framework's *drawing thread* main loop with its three subsequent scene graph traversals `UpdateContext()`, `UpdateAnimation()` and `Draw()` essentially follows a single-threaded scheme, the actual screen update is performed in a separate *flip thread*; this way a clean separation is established between the common drawing thread in the MML library and specific flip thread implementations for each available graphics device in the GFX_DEVICE library. Synchronization between these threads is performed via events; upon finishing the drawing of all relevant objects into a system-memory surface, the drawing thread sends a `DrawSceneDone` notification to the flip thread, which in turn sends back a `FlipDone` notification after the surface contents have been transferred to video memory and made visible by flipping the front and back buffers.

As described in the previous subsection, for each frame the renderer engine inside a `GFX_Device` instance receives all renderer objects to be drawn during the scene graph's `Draw()` traversal. The actual rendering process is then initiated by the `GFX_Device` via its active renderer areas after receiving a `DrawSceneDone` notification, prior to performing a buffer swap. Figure 5.4 illustrates this interaction between the two threads.

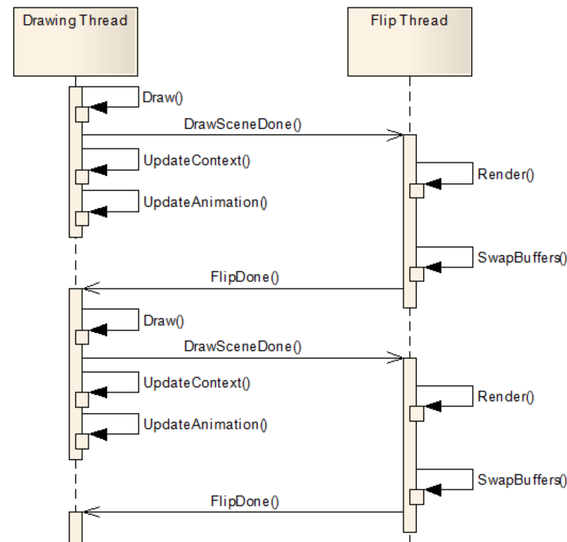

Figure 5.4: Sequence diagram showing the interaction between the common *drawing thread* and the device-specific *flip thread* for two subsequent frames.

5.2 Materials

In the OpenGL specification, the ”material” term is referred to exclusively in conjunction with specifying values for the ambient, diffuse, specular and emissive (self-illuminating) color components and a single value for the specular exponent (referred to as ”shininess”) used in lighting calculations. In contrast, commercial 3D modeling applications such as *Autodesk Maya* or *3D Studio Max* define materials on a higher level, which more closely resembles the real-world notion of a material. Here, the complexity of such a material may range from simple uniformly colored surfaces to a combination of multiple different texture maps each affecting surface properties such as color, translucency, bumpiness, reflectivity, etc. These modeling applications allow for creating and assigning different materials to individual geometric objects or even parts of geometric objects on a per-triangle basis.

When applied, complex materials usually produce convincingly realistic output using the modeling application’s built-in offline renderer. Ideally, extracting such a complex material automatically from a given modeler file and rendering this material on a real-time system would yield the same result as the output of the modeling application’s renderer; in practice however, the differences are more or less noticeable, not least due to the inherently different nature of the rendering methods used. Additionally, in the power range of the required target hardware platforms, there exist a number of difficulties in achieving such results without degrading overall performance:

- Emulation of complex materials comprising a high number of map channels for individual surface properties in real-time requires the use of efficient shader code. Writing a common shader that is able to handle all possible channel combinations might suffer from a large overhead.
- The evaluated Intel GPU-based platforms do not allow for more than 8 simultaneous texture channels; materials using more channels have to be rendered in more than one pass.
- More complex lighting models (incorporating e.g. anisotropic reflectivity) might require more complex shader code.

For these reasons, certain trade-offs have to be made when designing 3D models or scenes for the given platforms; in general, simplicity is the key. Graphic artists should try to keep overall material map channel counts low and aim for simpler lighting models.

5.2.1 Multi-Pass Materials

Even with high-end graphics hardware featuring a fair number of texture units and allowing for complex shader programs there exist a number of different rendering techniques that still require rendering in more than just a single pass. In general, this means that some geometry must be rendered multiple times after another, with a different material state set at each of these iterations.

5.2.2 GPU Shader Programs

As mentioned above, *Shaders* play a key role in producing a convincing appearance of real-time rendered 3D content on state-of-the-art graphics hardware. Through the use of the high-level *GL Shading Language* (GLSL), OpenGL provides access to GPU shader execution functionality for both *Vertex Shaders* and Fragment Shaders; to actually enable such shaders for rendering, OpenGL provides a means to create *program objects* to which those shaders can be attached to.

Shader source code must be provided to OpenGL as a human-readable ASCII string; the concept of run-time compilation by the driver allows to directly integrate shader code as part of a scene graph via adequate scripting nodes. Directly mapping OpenGL's mechanism of handling shaders, the EGD framework extensions provide scripting nodes for the definition of a shader program, which in turn accepts shader code nodes as direct children.

To allow for parametrization of a shader – e.g. for supplying different parameter values for each pass of a multi-pass material, or to define actual texture lookup channels – OpenGL provides a mechanism to retrieve the locations of user-defined variables defined in given shader code, and to set or query the actual values at these locations. The EGD extensions encapsulate this mechanism into a `program parameters` scripting node, which must be supplied in parallel to a material's attached shader program.

5.3 Node Instancing and Referencing

As stated in chapter 2, *node instancing* is an important concept for sharing and re-using parts of a scene graph to reduce memory consumption and increase manageability [15, 16]. Standard instancing in the extended EGD framework can be accomplished by simply defining one named `MGL_Geometry` node and defining several `MGL_GeometryReference` nodes with the same identifier elsewhere in the scene graph. This way, the referencing node inherits *all* properties of the node it refers to, such as visibility or internal state.

When dealing only with static geometry, there exists also a second possibility to perform instancing. As with all currently existing scripting nodes in the EGD framework, also `MGL_Geometry` nodes do not actually contain any renderable data. Analogous to existing 2D scripting nodes, MGL nodes always retrieve pointers to content via the framework's *resource manager*, which holds collections of individual read-only binary resources (so-called *assets*), such as 3D meshes, animation data or audio files. In the case when individual `MGL_Geometry` nodes refer to the same static geometry asset taken from the resource manager, that asset only needs to allocate and fill its internal vertex and index buffers once, and can pass this buffer directly back to the drawable object for rendering. Especially for higher-level nodes derived from `MGL_Geometry`, this method may be convenient as it supports individual node properties at the same time as maintaining better performance due to sharing a geometry asset's internal buffers.

5.4 Geometry Resources

Resource management in the EGD framework, as described in chapter 2, is based on the concept of *resource definition* and *resource package* (RP) files; each individual binary resource – or *asset* – inside an RP file is stored in a way that allows it to be loaded into system memory most efficiently, and to be used directly without the need for a complex parsing or initialization process. Applying this concept also to the suitable geometry file

formats evaluated in chapter 4 requires the design and implementation of additional loader modules for each supported format; these modules are then responsible for loading and parsing individual *VRML97* or *FBX* files, and for creating a binary representation of these files' content in system memory.

Regarding both these file formats, the parsing process can become fairly time-consuming when dealing with a large number of individual resource files; this may have a direct impact on the content development time due to an extremely increased application startup duration. In addition, the attempt to conceal the lack of per-vertex tangent vector storage by performing tangent vector estimation after loading may increase this duration even further.

These facts eventually led to the decision to introduce a new, EGD-specific binary file format that accounts for data storage most efficient regarding the EGD framework's resource management, to be further known as the *EGD 3D* (E3D) file format. With the respective requirement from chapter 3 coming into effect through this decision, it was necessary to extend the present tool chain by providing a converting tool that fills the gap between modeling applications and actual framework. E3D files were conceived to directly represent a loaded geometry file's actual footprint in system memory; in contrast to the *EGD Resource Packer* generating complete resource packages from any number of assets, the newly conceived *EGD Convert3D* application only writes out a single E3D file from a geometry file loaded to system memory, however by making use of the identical loader mechanism. As a result, for an E3D file inserted into a *resource definition* file, the complexity of the specific E3D loader module is limited to loading a given input file directly to memory and performing a simple sanity check on the file contents.

Chapter 6

Prototype Implementation

Based on the concepts and prototype design modeled in chapter 5, this chapter describes the actual implementation of the framework extensions. More specifically, it deals with how the extended functionality is actually integrated into the existing structure, followed by a detailed description of data structures used, and newly introduced scripting node classes and their related internals.

6.1 Framework Integration

All of the OpenGL functionality added to the Atronic EGD framework is encapsulated in a separate loadable module (MM_OPENGL), which sits on top of the already existing *Basic Functions Library* (MM_BFL), *Multimedia Library* (MM_MML) and various other shared libraries providing e.g. threading support, string conversion or common utility functions as well as third-party components such as audio/video decoders or data compression. This module is explicitly loaded via the newly introduced OpenGL graphics device implementations residing in the MM_GFX_DEVICE library, one for each specific hardware/operating system combination. Figure 6.1 shows a block diagram of the EGD Multimedia application's library components and their dependencies.

6.2 Geometry Resources

As described in the previous chapter, geometry resource files can be integrated into a content project in the form of *VRML97* and *FBX* files as well as files in the newly introduced E3D format, by means of their respective loader modules. Internally preparing data

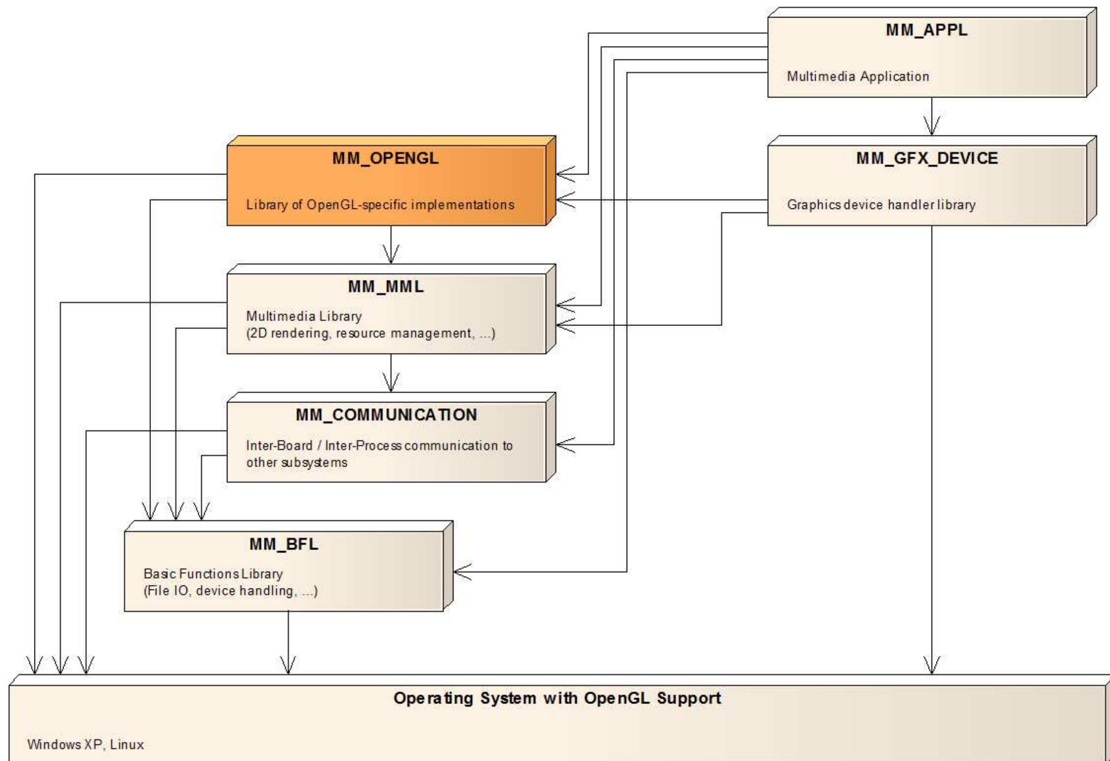

Figure 6.1: Block diagram of the EGD Multimedia base software. Low-level utility and third-party libraries are omitted for clarity; the newly added MM_OPENGL library is highlighted in orange color.

structures designed for efficient handling by the rendering system, both the *VRML97* and *FBX* hereby actually produce a system memory footprint equal to an E3D file on disk.

6.2.1 The E3D File Format

The E3D format itself was designed to be conformant to the *RIFF* standard [5] conceived by Microsoft and IBM in 1991, consisting of a number of so-called data *chunks* each identified by a specific 4-byte character code ("FOURCC") holding individual pieces of content data. An illustrated description of this concept can be found in [6]; following below is a list of data chunks necessary to build up a valid E3D file.

6.2.1.1 Geometry Header Chunks

Global specifics of the geometry data in the file are stored in a *Geometry Header* ("GEOM") chunk, holding information about the file format version, number of

animation frames, the desired playback frame rate and the size and location of the geometry's overall bounding box. A valid file must contain exactly one such chunk, and it must be defined prior to any other chunks described below.

6.2.1.2 Material Chunks

Material ("MATL") chunks simply consist of a unique integer ID and an assigned material name; their specific purpose is to provide a mapping between internally used material IDs and human-readable names. A valid file must contain exactly one material chunk for each integer ID specified in a mesh object defined below.

6.2.1.3 Mesh Geometry Chunks

Mesh Geometry ("MESH") chunks represent actual geometry for rendering, in the form of a mesh consisting of either individual triangles or triangle strips. Each mesh chunk holds its own unique integer ID; hierarchical representation is possible by specifying a non-negative number in the chunk's parent mesh ID field. Each mesh chunk may also have a specific material assigned, if the chunk's material ID field holds a non-negative value. In addition, each chunk also holds an OpenGL-conformant model-view matrix to allow for hierarchical transformations.

Geometry data in a mesh chunk are arranged in a way so that they may act directly as a source for OpenGL vertex and index buffers for efficient rendering; each vertex entry in the chunk's data area must at least contain that vertex's position, optionally also normal vectors, tangent vectors, colors and up to eight individual texture coordinates may be present on a per-vertex basis. If no actual vertex or triangle data are specified or no material is assigned for a given chunk, the node represented by this chunk can act as a transform-only node within a geometry's hierarchy.

Key-frame animated transformations are stored with individual model-view matrices for each animation frame; interpolation between two subsequent key frames is provided by additional relative translation vectors and rotation quaternions for each transformation key frame. Animated mesh deformations are supported through the use of individual vertex position, normal vector and tangent vector fields for each key frame.

6.2.2 EGD Assets

Once loaded into the EGD framework, individual assets may be retrieved via the framework's resource manager; for the newly introduced geometry assets, the 3D prototype

offers the OpenGL-specific `ASSET_GlGeometry` type, which directly encapsulates an E3D asset in system memory. Due to the efficient memory layout, the initialization effort for these assets is greatly reduced; necessary steps merely consist of pointer list creation for meshes and materials by skipping over individual file chunks.

After retrieving an `ASSET_GlGeometry` object from the resource manager, the class interface provides methods to retrieve pointers to individual `ASSET_GlMesh` instances contained therein, which directly map to corresponding mesh geometry chunks.

6.3 Scripting Nodes

The current implementation of the MML library already offers a mechanism to automatically add user-defined scripting nodes derived from a common base class (BMC_Node). In fact, one preferred way to implement content for the framework is to derive from common script node base classes and export these specialized nodes so that they may be instantiated via an XML scene graph file.

The OPENGL library also makes use of this mechanism to register its own, OpenGL-related, nodes with the system. A new type of node was introduced, *MGL_Object*, with the "MGL" prefix (short for *Multimedia GL*) in analogy to the 2D *Multimedia Objects* ("MOB"); this class represents the base class for all OpenGL -related implementations in the extended framework.

Additionally, an abstract base class was introduced to the MML library, `MOB_RendererArea`, which is the base for any – not restricted to OpenGL – hardware-accelerated rendering area; a specialized `MOB_GlRendererArea` that performs OpenGL specific rendering is also provided. See Figure 6.2 for a simplified overview of the extended framework's class hierarchy.

As stated earlier, the actual MGL scripting nodes do not directly contain any OpenGL function calls or state information; instead, an MGL node, depending on its specific function, holds pointers to one or more renderer objects, which are derived from a common base class, `OGL_Object`. The following sub-sections give an overview of the existing scripting nodes and renderer object classes, and their specific purpose and interaction.

6.3.1 Renderer Areas

Derived from the abstract `MOB_RendererArea` node class, the `MOB_GlRendererArea` class provides the front-end to the content developer through which OpenGL3D content in

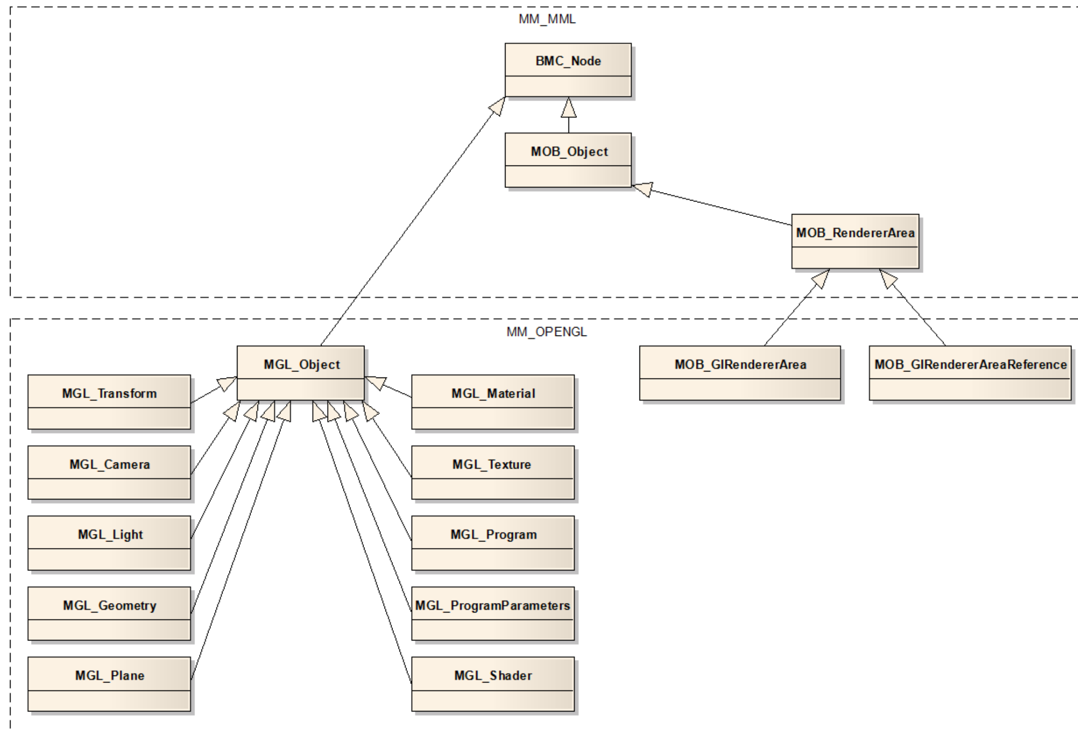

Figure 6.2: Class hierarchy of the newly introduced OpenGL-related scripting nodes, with their connection to the existing base classes in the MML library.

the form of MGL scripting nodes can be inserted into a 2D MOB scene graph at any point. By specifying a unique string identifier for the `id` attribute, it is possible to create a named instance of a `MOB_GlRendererArea` that may later be referenced via a `MOB_GlRendererAreaReference` instance carrying the same ID.

Internally, both of these scripting nodes hold a pointer to an `OGL_RendererArea` instance, which performs the actual work of OGL object creation and destruction, depth sorting (to ensure a correct back-to-front drawing order for semi-transparent primitives) and rendering of enqueued objects, and maintaining internal transformation and rendering states. This class implements the newly introduced `GFX_RendererArea` interface, which provides implementation-independent method declarations related to renderer area handling (see Figure 6.3).

Upon initialization and destruction, an `MOB_GlRendererArea` creates and destroys its encapsulated `OGL_RendererArea` instance via the `CreateRendererArea()` and `DestroyRendererArea()` methods of its associated `MOB_Screen`, respectively; an `MOB_GlRendererAreaReference` retrieves its reference by passing the given identifier

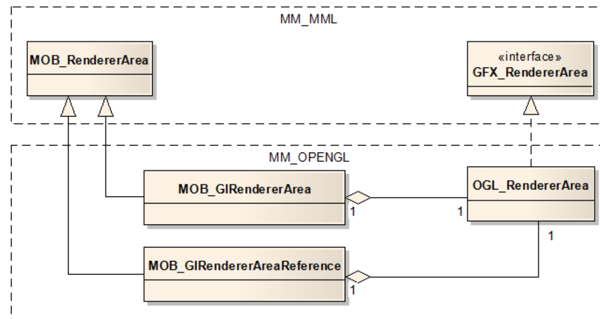

Figure 6.3: Renderer area class hierarchy.

when calling the `GetRendererArea()` method. All calls to these methods newly introduced to the `MOB_Screen` interface are routed to the `GFX_Device` attached to that screen, and in turn re-routed to the device's underlying renderer engine.

For all MGL scripting nodes contained within a given renderer area, creation and destruction of internal "OGL" renderer objects is performed upon their initialization and deinitialization, respectively. For that reason, a reference to the `OGL_RendererArea` corresponding to the containing `MOB_GlRendererArea` is passed to each individual node; calls to the methods `CreateRenderObject()` and `DestroyRenderObject()` exhibited by its interface are also relayed to the underlying `OGL_RendererEngine`.

6.3.2 Geometric Transformations

The generic `OGL_Transform` object encapsulates a 4x4 transformation matrix stored in column-major order to directly reflect matrices in OpenGL. Specific operations such as translation, rotation or scaling have no direct equivalent here; instead, these are presented to the user via corresponding MGL scripting nodes that fill in the appropriate values into their underlying `OGL_Transform` objects. Upon traversing the scene graph, the MGL nodes push and pop their transform objects to their assigned renderer area whenever they are entered and left, respectively; any scripting nodes containing geometry objects then enqueue their underlying drawables to the renderer area using the currently active transformation *in* the area. Actual transformation states are managed by the renderer area itself during the render process and loaded into OpenGL via `glLoadMatrix()` whenever necessary.

To the user, the framework provides four specialized scripting nodes: `MGL_FreeTransform`, `MGL_Translate`, `MGL_Rotate` and `MGL_Scale`. `MGL_FreeTransform` directly exposes the encapsulated general-purpose 4x4 matrix for arbitrary

transformations, whereas the latter three each implement their respective transformation interface from the MML library to perform translation, rotation and scaling, respectively (see Figure 6.4). Rotations must be specified via a main rotation axis vector and an angle scalar, following a right hand rule. Scale factors may be specified independently for each axis; if done so however, the `normalize_mode` attribute of the material used must be set to either `MGL_NormalizeMode::STANDARD` or `MGL_NormalizeMode::RESCALE` to preserve correct normal vectors for lighting calculations.

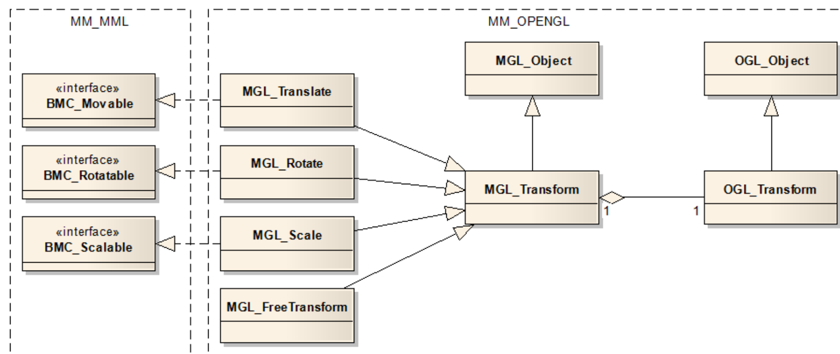

Figure 6.4: Class hierarchy of available transformation scripting nodes showing the connections to their encapsulated renderer object.

The transformation interfaces themselves are also part of the work for the extended framework; to allow for a common handling of 2D and 3D objects with regard to transformations, the existing implementation of `MOB_Object`, `MOB_RotatableObject` and `MOB_ScalableObject` was modified by extracting the respective transformation functionality from these nodes (translate, rotate and scale) and realizing these newly created interfaces for these classes as well.

6.3.3 Cameras

From the renderer's point of view, there is no such thing as an explicit camera object. Instead, MGL camera script nodes (`MGL_OrthoCamera` and `MGL_PerspectiveCamera`) directly apply their respective projection matrix to the renderer area once per frame when they are encountered during scene-graph traversal. Thus, the camera is set up in a way so that the main viewing axis runs along the negative Z-axis; to account for camera moves and rotations, it is up to the content developer to specify necessary transformations to counter-rotate the whole sub-scene around the fixed camera. See Figure 6.5.

Camera nodes are closely tied to the renderer areas they are defined within. Every

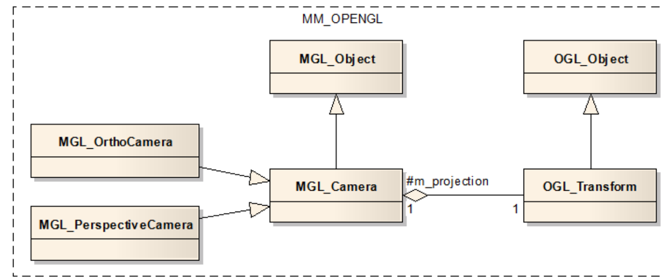

Figure 6.5: Camera-related class hierarchy.

MOB_GlRendererArea can hold exactly one camera instance that must be defined as a direct child of the area; if no camera is explicitly specified, an orthographic camera is used with default settings.

6.3.4 Light Sources

Similar to the OGL_Transform object, the OGL_Light object acts as a generic container for an OpenGL light source whose actual behavior is defined via its encapsulating MGL scripting node. Here, MGL_DirectionalLight provides a light source emitting parallel light rays at infinite distance (comparable to a sunlight system), whereas MGL_PointLight radially emits light rays from a central point, similar to a naked light bulb. See Figure 6.6.

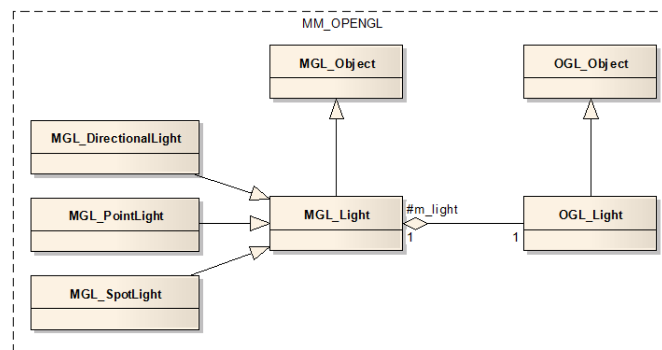

Figure 6.6: Light source-related class hierarchy.

By default, these specialized light source node classes point towards the negative Z axis; orienting a light source in a different direction must be performed by specifying an MGL_Rotation parent that performs the necessary transformation. Analogous to its orientation, the position of a light source must be specified via an MGL_Translation parent;

with the exception of MGL_DirectionalLight, for which positioning has no effect.

6.3.5 Materials

Materials in the renderer are represented by instances of the OGL_Material class, which act as a compound for various individual properties. The mandatory OGL_Surface child object defines the material's overall surface appearance; it encapsulates the actual OpenGL material parameters such as ambient, diffuse, specular and emissive color components and specular exponent, as well as other OpenGL parameters such as lighting behavior, alpha blending functions and depth buffer access. Additionally, a material may carry an optional OGL_TextureSet containing references to up to 8 individual texture maps, and an optional OGL_Program together with an OGL_ProgramParameters object that define a specific vertex/fragment shader program and related parameters (see below). Materials can also define a specific sorting order for the drawables they are bound to; setting the alpha blending parameters so that any geometry rendered with this material appears semi-transparent usually requires the sorting order to be set to "back-to-front" instead of the default "don't care" setting. See Figure 6.7.

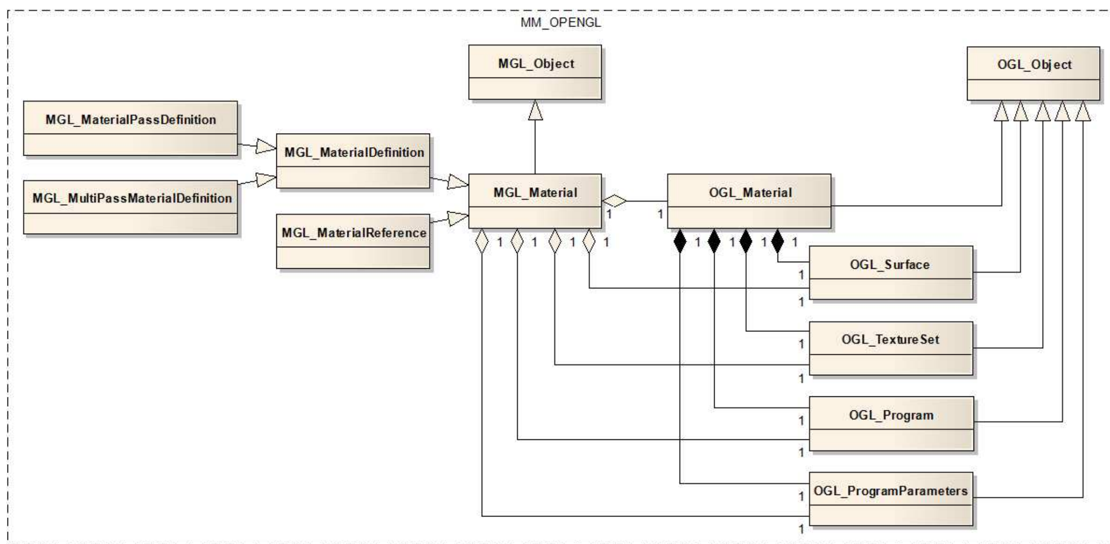

Figure 6.7: Class hierarchy of material-related scripting nodes and relationship to internal renderer objects.

Defining a standard single-pass material within the scene graph can be performed via an MGL_MaterialDefinition node, which exposes a number of attributes that define the actual appearance:

- `ambient_color`, `diffuse_color`, `specular_color` and `emissive_color` control the individual material color parameters used in OpenGL's lighting equations; each of these values may be specified either as a string of comma-separated individual brightness values for the alpha, red, green and blue color components, each in the range from 0 to 255, or as a single hexadecimal string of the form "0xAARRGGBB".

- `shininess` defines the specular exponent used in OpenGL's lighting model for the specular color parameter, in the range from 0 to 128.

- `z_buffer_usage` defines OpenGL's depth-buffer access policy; by specifying one of the `MGL_ZBufferUsage` enumeration's four member values `NONE`, `READ_ONLY`, `WRITE_ONLY` and `READ_AND_WRITE`, it is possible to individually enable depth buffer comparisons and writes.

- `visible_faces` controls culling of front-facing and back-facing primitives via the `MGL_VisibleFaces` enumeration; possible values are `FRONT`, `BACK` and `FRONT_AND_BACK`.

- `alpha_blending`, when set to `MGL_AlphaBlending::ON`, allows to globally enable blending in OpenGL; in this case, the additional `src_blend_factor` and `dst_blend_factor` attributes specify the actual pixel arithmetic used in the blending equation. Both of these attributes accept members of the `MGL_BlendFactor` enumeration; the symbolic constants here match those of the OpenGL specification for `glBlendFunc()` without the "GL_" prefix, e.g. `MGL_BlendFactor::SRC_ALPHA` instead of `GL_SRC_ALPHA`.

- `face_lighting` defines OpenGL's polygon lighting behavior. Specifying the `MGL_FaceLighting::OFF` enumeration value completely disables all lighting calculations for that material; using the `ONE_SIDE` member considers only front-facing polygons, whereas `TWO_SIDE` performs lighting calculations on both front- and back-facing ones.

- `sorting` specifies the sorting order for all geometries utilizing this material, with possible values of the accepted `MGL_Sorting` enumeration are `DONT_CARE`, `BACK_TO_FRONT` and `FRONT_TO_BACK`.

Multi-pass materials can be defined via the `MGL_MultiPassMaterialDefinition` container class. Individual passes must then be specified as direct children of that container

via multiple MGL_MaterialPassDefinition instances; both of these classes inherit their attributes from MGL_MaterialDefinition. Setting the container's attributes defines the common behavior over all individual passes; these common values can then be overridden by each individual pass.

Referencing a previously defined material is made possible through the use of an MGL_MaterialReference node; however, only named instances can be referenced. Specifying a unique string identifier for the id attribute of an MGL_MaterialDefinition or MGL_MultiPassMaterialDefinition node creates such a named instance to be later referenced by a reference node with the same ID.

6.3.6 Textures

An individual texture object is represented by an OGL_Texture instance, which is responsible for creation, destruction and update of an underlying OpenGL texture object; if available, it makes use of an OpenGL *pixel buffer object* to improve image data transfer performance. Additionally, it is possible to specify the type of mapping to be performed, either planar or spherical environment mapping. To apply one or more textures to a material, an OGL_TextureSet object is needed, which provides eight possible texture slots for holding individual texture objects; these slots directly map to the corresponding texture units in OpenGL. See Figure 6.8.

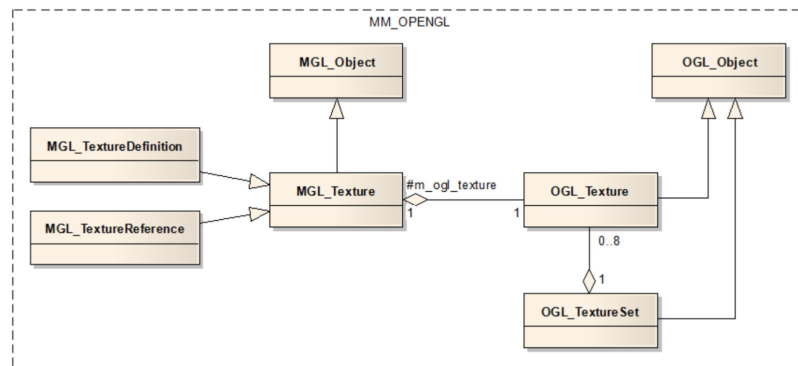

Figure 6.8: Class hierarchy of texture definition and reference nodes, together with encapsulated renderer objects.

Defining a texture container in the scene graph is made possible through the use of the MGL_TextureDefinition scripting node. To define the texture's actual dimension, both the w and h attributes must be specified; both of these values must be a power-of-two integer number to account for restrictions in certain OpenGL implementations. By default,

the texture is applied using planar mapping; using it as a spherical environment map is permitted by setting the `mapping_type` attribute to `MGL_MappingType::SPHERICAL`.

Analogous to material nodes, referencing a previously defined texture container needs a named instance of an `MGL_TextureDefinition` node, and an `MGL_TextureReference` with the same ID. Both definition and reference nodes allow to independently set their blending mode and texture slot, i.e. textures may be defined in a material using a specific slot and later be referenced using a different slot. The `unit` attribute defines the slot to be used, as a number between 0 and 7. The `mode` attribute accepts one of several members of the `MGL_TextureMode` enumeration whose members match the blending modes defined in the OpenGL specification: `REPLACE`, `MODULATE`, `DECAL`, `ADD`, `BLEND` and `COMBINE`. Whereas `MGL_TextureDefinition` nodes may be defined anywhere within an MGL sub-graph, the actual binding of a texture definition or reference to a specific material is accomplished by defining the respective node as a direct child of a material node.

6.3.7 Shader Programs

As stated in chapter 5, OpenGL's shader mechanism is directly wrapped by suitable renderer objects internally; OpenGL vertex and fragment shader objects are encapsulated by instances of the `OGL_Shader` class and linked together into an OpenGL program object held by an `OGL_Program` instance. Creation of a shader program in the scene graph is accomplished via an `MGL_ProgramDefinition` node, as a direct child of a specific `MGL_MaterialDefinition` node for which the shader program is to be applied. Within this `MGL_ProgramDefinition` node, individual `MGL_ShaderDefinition` or `MGL_ShaderReference` nodes may be specified that define the actual code objects. To specify the type of an `MGL_ShaderDefinition` node, this class exports the `type` attribute accepting either `MGL_ShaderType::FRAGMENT` or `MGL_ShaderType::VERTEX` as input; the actual shader code is defined via the `source` attribute that directly accepts GLSL code.

Program and shader nodes employ the same definition/reference mechanism as material and texture nodes; the node classes `MGL_ShaderDefinition` and `MGL_ShaderReference` derived from `MGL_Shader`, and `MGL_ProgramDefinition` and `MGL_ProgramReference` derived from `MGL_Program` serve these specific purposes. See Figure 6.9.

Accessing any custom input parameters defined in the GLSL shader code can be performed via an `MGL_ProgramParameters` node, which internally holds a pointer to an `OGL_ProgramParameters` object directly acting on the actual program object it is bound to. `MGL_ProgramParameters` must also be specified as a direct child of a spe-

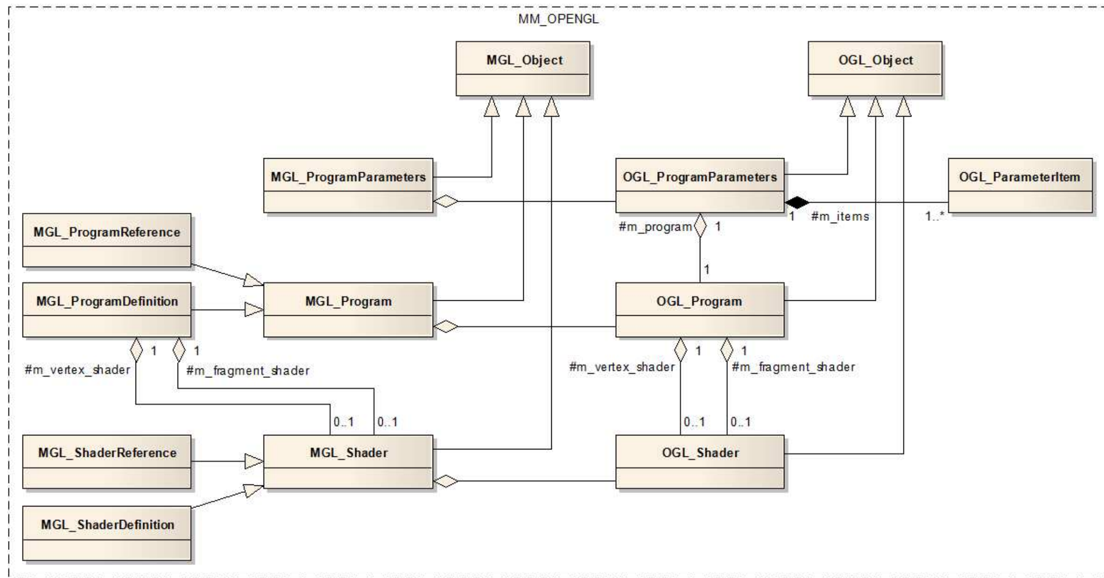

Figure 6.9: Class hierarchy of shader-related classes for scripting nodes and renderer objects.

cific MGL_MaterialDefinition node in the scene graph, as a sibling to the MGL_Program node it is supposed to act on; the actual parameters are specified via the params attribute that accepts a string of individual semicolon-separated parameter assignment statements.

Such statements must be of the form "⟨data type⟩ ⟨variable name⟩ = ⟨value⟩;", where data type accepts any of the following native GLSL types: int, ivec2, ivec3, ivec4, float, vec2, vec3, vec4, bool, bvec2, bvec3, bvec4, sampler1D, sampler2D, sampler3D, sampler1DShadow, sampler2DShadow or samplerCube; variable name, together with data type must match the name and type of an attribute or uniform variable declaration in the GLSL shader to attach to. For single-value data types such as float or sampler2D, value simply accepts the desired value; vectorized data types must be specified by their vector constructor form, e.g. "vec2 myValue = vec2(1.0, 0.5);".

6.3.8 Drawables

The OGL_Drawable interface provides the base class for any geometrical objects to be rendered; currently there are four classes that implement this interface.

OGL_SimpleMesh provides a means for creating geometry by directly passing individual arrays of vertices, normal vectors and texture coordinates together with corresponding triangle index arrays. As the class name suggests, this should be used for rather simple (i.e.

low triangle count and/or static) geometries; internally, an OpenGL display list is created that is filled up with individual calls to `glNormal()`, `glTexCoord()` and `glVertex()` to allow for independent index arrays for these attributes. To the user, this type of geometry is available through the scripting node classes `MGL_Plane` and `MGL_Cube`. `MGL_Plane` creates a square plane in the global X/Y plane with a side length of one; if a textured material is applied, it is possible to specify the texture coordinate range by setting the exported coordinate attributes `tx1`, `ty1`, `tx2` and `ty2`. `MGL_Cube` provides a simple unit-length cube.

`OGL_Mesh` on the other hand does not provide a means to programmatically create geometry in a simple way; instead it is able to make efficient use of the framework's resource management system by directly accepting `ASSET_GlMesh` objects from the read-only resource collections. For drawing performance reasons however, it is still necessary to allocate an OpenGL *vertex buffer object* for each mesh contained in the asset; an additional system memory buffer is also needed when dealing with animated sets of vertices and normal vectors. This type of geometric object can be instanced via `MGL_Geometry` scripting nodes; each node instance must have exactly one `ASSET_GlGeometry` resource specified via the `geometry` attribute. Upon initialization, this resource object is parsed, and for each individual `ASSET_GlMesh` contained within, exactly one corresponding `OGL_Mesh` object is created. Analogous to the existing `MOB_Animation` class, an optional animation controller may be attached via the `animation_controller` attribute to allow for playback of key-frame animation contained in the geometry resource specified. Analogous to textures, materials and shaders, this geometry type may also be defined as a named instance using the `id` attribute. Later referencing this named instance is accomplished via an `MGL_GeometryReference` node specifying the same identifier. See Figure 6.10.

In order to correctly display geometry objects, it is necessary to establish a connection to one or more specific materials. For the above mentioned scripting nodes, this is accomplished by specifying such a material (`MGL_MaterialDefinition` or `MGL_MaterialReference`) as a child of a given geometry node; here, `MGL_Plane` and `MGL_Cube` nodes are restricted to expect *exactly one* material child. `MGL_Geometry` nodes on the other hand may have a geometry resource attached that contains more than one actual mesh; in this case, it is necessary to specify multiple material children. To correctly identify and assign these materials to the meshes contained in the resource, the user must specify the `link` attribute for each material node; the link identifier of each material must then match the corresponding identifier stored in the `ASSET_GlGeometry` resource.

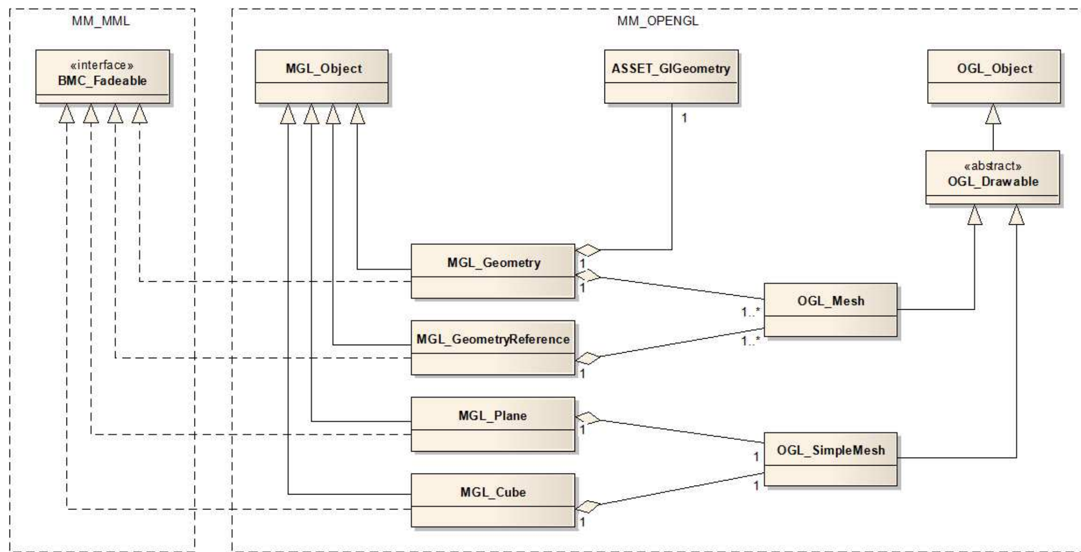

Figure 6.10: Class hierarchy for geometry-related scripting nodes, renderer objects and assets.

As all geometry-related scripting nodes implement the `BMC_Fadeable` interface, it is possible to individually specify an alpha value per node, either via setting the exported **`alpha`** attribute or via the interface's `SetAlpha()` method.

Chapter 7

Content Creation Workflow

To be able to actually make efficient use of the extended framework prototype from an end-user's perspective, it is necessary to provide additional means to support the content creation workflow from graphic artist to developer; the following sections give a brief insight into that workflow.

7.1 Geometry File Conversion

As already stated in chapter 5, the decision to define a proprietary geometry file format makes it necessary to also extend the present tool chain; for that reason the *EGD Convert 3D* application was created that allows for importing given FBX scene files and converting them to the newly introduced E3D format. A typical screen shot of this application is presented in figure 7.1. As indicated by the depicted user interface, the application provides additional functionality beyond the basic task of converting files:

- Imported geometry is presented in a preview window besides its complete hierarchical structure; it is possible to manipulate certain properties of each node – such as visibility, transformation parameters, material assignment etc. – prior to the conversion process. Key-frame animation within individual geometries can be inspected via a time-line slider.
- Fine-tuning of material properties can be achieved via a simple material editor that directly reflects the actual parameters that can be specified for an XML material node.
- Simple image manipulation controls are provided for imported textures; it is possible

to resize the texture and to individually adjust brightness and contrast of the image, additionally the image may be distorted to create a suitable texture for applying spherical environment mapping.

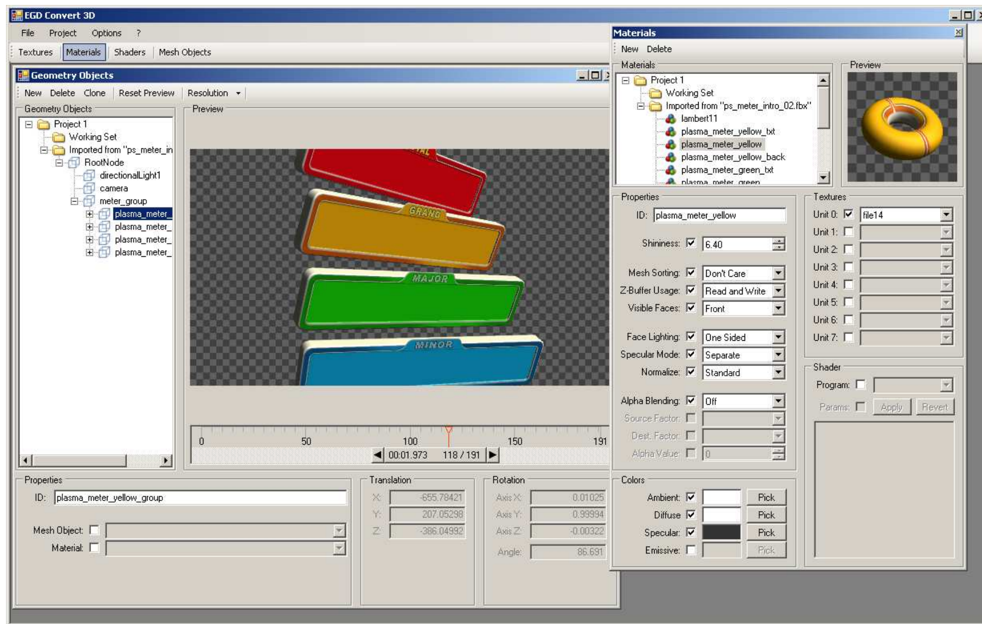

Figure 7.1: The main window of the *EGD Convert 3D* application, showing geometry and materials imported from a given FBX file.

In addition to the possibility of manipulating imported materials, applying advanced material effects is facilitated through a simple code editor for on-the-fly creation of GPU shader programs written in GLSL. Vertex and fragment shader code can be typed or copied into the respective tool window, clicking the "Apply" button triggers the graphics driver's internal GLSL compiler to create an OpenGL GPU program object. Correct functioning can then be verified via the compiler's log output, and by assigning the shader object to a specific material together with a user-defineable set of program parameters. Figure 7.2 shows the preview of a DOT3 bump map shader in the converting application's environment.

When the actual conversion process is started by the user, each geometry root node including all its child nodes is converted to a separate E3D file storing geometry, animation and material bindings. Necessary textures are converted to individual PNG image files, and a resource definition file is created that includes references to all these files. For the actual definitions of texture objects, shader code and materials, individual XML scene

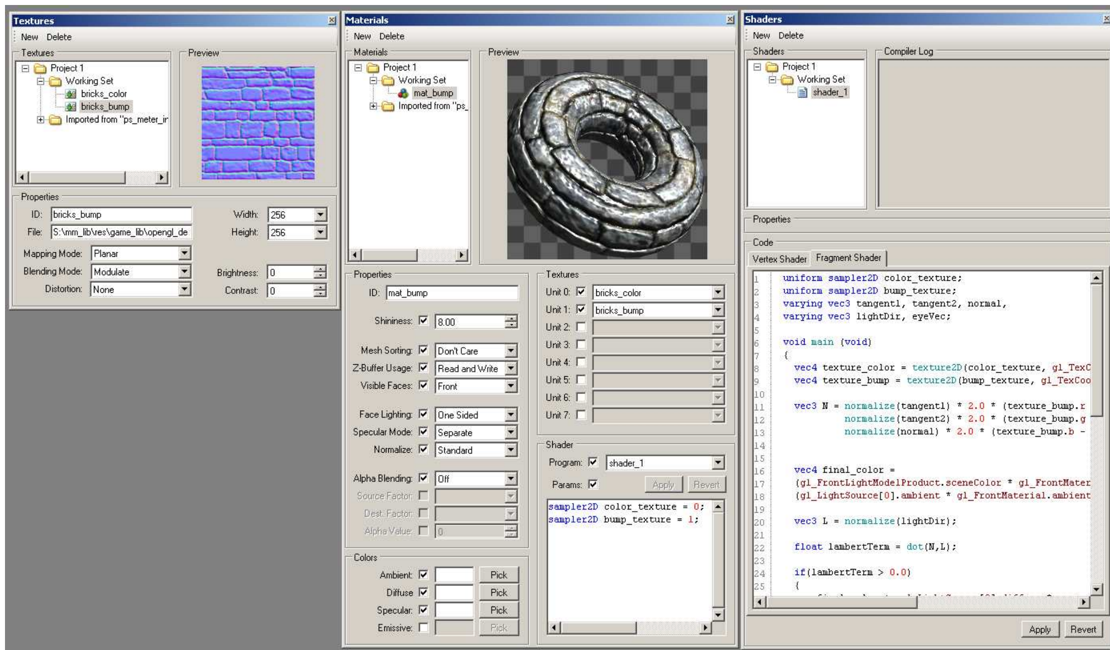

Figure 7.2: Tool windows of the *EGD Convert 3D* application showing on-the-fly creation of a DOT3 bump map shader material.

graph fragments are generated that can be directly included into a full graph by means of XML entities.

By utilizing the Autodesk FBX SDK for importing files, and by actually integrating the hybrid framework for the purpose of rendering material and geometry previews, it was possible to greatly reduce the actual implementation effort for the converting application. Additionally, the approach of directly using the framework for preview rendering offers the particularly interesting possibility to actually view imported scenes as they would be shown on an EGM, thus making it easier to spot possible deficiencies in any stage of the workflow.

7.2 Tool Chain Extension

With the *EGD Convert 3D* application playing a central role in the content creation tool chain, the actual flow can be most accurately summarized as follows:

- Creating three-dimensional objects or scenes is done via the available modeling applications, currently either *3D Studio Max* or *Maya*, and saved in these applications' respective native file format.

- Exporting these objects or scenes to the FBX file format is then performed through the use of the appropriate FBX exporter plug-in provided by the application vendor.
- In the next step, the FBX files are imported to the *EGD Convert 3D* application; at this point it is possible to verify the proper appearance of the imported data with the help of the available render preview windows before actually saving the converted version to disk. On occasion, these first three steps may be carried out repeatedly until satisfactory results are achieved, without the need to prematurely integrate unfinished data into a specific content project for in situ verification.
- The converted data, comprising E3D geometry files, PNG texture images, resource definition files and XML scene graph fragments, are then available for integration into a content project; from this point on, handling these resources is accomplished via the EGD SDK's already existing set of tools (resource editor, resource packer etc.).

Having such a set of converted 3D data at hand, a content developer is now ready to construct his or her specific content project. With an acceptable set of available scripting nodes covering the full range of the hybrid prototype's 3D capabilities, it is already possible to create highly sophisticated content through the mere use of XML scene graph description files. However, as was already observed during the use of the existing 2D-only framework over the past few years, certain tasks require advanced node handling, which may be accomplished more easily through the creation of specific higher-level C++ classes, by means of derivation or encapsulation of existing scripting nodes.

Which one of these techniques to use is up to the content developer and must be decided as the case arises; there is no universal rule for making such a decision. The next chapter illustrates a number of selected application examples, with a brief glance at both methods.

Chapter 8

Application Examples

To ensure correct operation of the hybrid prototype developed in the course of this thesis, a number of test cases were prepared that cover various aspects of the implementation; this chapter describes a carefully selected subset of these cases, with the additional goal of demonstrating the actual usage of the extended features in a practical environment.

8.1 Progressive Meter

In casino EGMs, the term *Progressive Meter* refers to some sort of display that shows the current amount of money that can be won by a player when he or she e.g. hits a jackpot. Such a display is either present as a physical stand-alone device, or in virtual form shown on an EGM's video screen, possibly displaying visually nice animation effects to attract players passing by. In the EGD framework there already exists a 2D implementation of such a virtual progressive meter; here, the digits of the ever increasing money amount are cycling in from right to left similar to a classic mechanical odometer known from a car's dashboard. See Figure 8.1 and Listing 8.1.

Figure 8.1: A rendering of the existing two-dimensional progressive meter with default style.

```
1 <!-- ======== Define a regular container object ======== -->
2 <MOB_Object w = "770" h = "98">
3   <!-- ======== The background image including frame ======== -->
4   <MOB_AnimationSingle
5     animation="resource('bg_progressive_meter_frame.png')"/>
6   <!-- ======== The actual meter object ======== -->
7   <MOB_ProgressiveMeterDefault
8     x = "50"  y = "10" z = "2"
9     w = "670" h = "79"
10    font = "resource('progressive_meter_display_big.font')"
11    font_size = "79"
12    controllers.1 = "MCO_ProgressiveAtakaProtocol()"/>
13 </MOB_Object>
```

Listing 8.1: A code snippet of the existing progressive meter sub-graph.

The following example presents a way of re-using this meter object and encapsulating it into a three-dimensional geometric object for additional visual effects; this implementation was chosen for a number of reasons:

- Re-using of existing implementations: The current meter object is the result of months (if not years) of development, fine-tuning and adaptation to ever evolving game design needs; it is well tested and accepted and should therefore be left unchanged.
- Dynamic texture generation: Spinning the digits of the money amount is internally more complex than just playing a video file; here, the EGD framework's mechanism of re-drawing only modified parts of a 2D image comes into play.
- Sealed-off rendering area: Often, such a virtual *In-Machine Display* is not considered part of a real content scene, but is simply overlaid over any other content present on screen. This makes it a good candidate to demonstrate the use of separate renderer areas in the 3D implementation.
- No side effects on existing games: The EGD framework provides an XML file defining a sub-graph of the scene only containing a progressive meter object and image objects that define a common appearance; a game may simply include this file to provide a progressive meter with default style. Changing this sub-graph to show the meter in 3D effectively enables a three-dimensional progressive meter for all existing games without any additional work to be done.

In addition, this example shows how to create 3D content by only using ”stock” node classes; it does not require the developer to implement any derived classes in C++. Listing

8.2 shows how the 2D object may be embedded into a 3D sub-scene. Note the re-using of the original implementation in lines 11 to 18, with only the background image being replaced with a different one without a border frame. Figure 8.2 shows the visual output of that code snippet.

```
1 <!-- ======== Define a separate renderer area ======== -->
2 <MOB_GlRendererArea id="progressive_meter_context" w="800" h="200">
3   <!-- ======== Setup camera ======== -->
4   <MGL_PerspectiveCamera center_x="400" center_y="100"/>
5   <!-- ======== Setup lighting ======== -->
6   <MGL_Rotate axis_x="0.5" axis_y="1.0" axis_z="0.0" angle="60">
7     <MGL_DirectionalLight/>
8   </MGL_Rotate>
9   <!-- ======== Textures ======== -->
10  <MGL_TextureDefinition id="progressive_meter_odo" w="512" h="128">
11    <!-- ======== The background image ======== -->
12    <MOB_AnimationSingle
13      animation="resource('progressive_meter_background.png')"/>
14    <!-- ======== The actual meter object ======== -->
15    <MOB_ProgressiveMeterDefault
16      w="512" h="128"
17      font="resource('progressive_meter_display_big.font')"
18      font_size="79"
19      controllers.1="MCO_ProgressiveAtakaProtocol()"/>
20  </MGL_TextureDefinition>
21  <!-- ======== Geometry ======== -->
22  <MGL_Translate x="400" y="100" z="50">
23    <MGL_Geometry geometry="resource('progressive_meter.ase')">
24      <!-- ======== Implicitly define materials here ======== -->
25      <MGL_MaterialDefinition
26        link="MatCase"
27        ambient_color="250, 186, 13"
28        diffuse_color="250, 186, 13"
29        specular_color="255, 255, 255"
30        shininess="127"
31      />
32      <MGL_MaterialDefinition
33        link="MatOdometer"
34        ambient_color="255, 255, 255"
35        diffuse_color="255, 255, 255"
36        specular_color="0, 0, 0"
37      >
38        <!-- ======== Reference texture previously defined ======== -->
39        <MGL_TextureReference id="progressive_meter_odo"/>
40      </MGL_MaterialDefinition>
41    </MGL_Geometry>
42  </MGL_Translate>
43 </MOB_GlRendererArea>
```

Listing 8.2: Example code for a 3D progressive meter using the existing 2D implementation as the source for a dynamic texture.

Figure 8.2: A rendering of a three-dimensional progressive meter using the existing 2D implementation as the source for a dynamic texture.

8.2 Interactive Game Selection

One of Atronic's future goals is to offer gaming machines with more than one game installed simultaneously, and to let the player choose which one of them he or she wants to play. Naturally, such a game selection screen shall be as visually attractive as possible and also offer an intuitive user interface. Utilizing the possibility of the EGM's built-in touch screen, it was chosen to implement a simple selection menu that allows the player to leaf through a set of title screens of available games, similar to the style of the "Cover Flow" known from Apple's iPod. See figure 8.3.

Figure 8.3: A screen shot of the game selection menu screen.

This menu example actually consists of two separate node classes:

- The `MOB_ImageFlow` class is derived from the `MOB_GlRendererArea` class, with its interface extended to allow the actual menu scroll position to be controlled externally.
- The `MOB_ImageFlowButton` class is derived from a standard `MOB_Button`, which is

used to capture ”up”, ”down” and ”move” events from the touch screen. The derived button is used to calculate the actual menu scroll position from these input events and pass them on to the MOB_ImageFlow node it is linked to.

Furthermore, the MOB_ImageFlow class is designed to relieve the content developer from defining all the transformation and geometry nodes necessary for bringing the menu contents on screen. All he or she has to do is specify an MGL_TextureDefinition child node with a valid ID for each image that is supposed to appear in the menu; the initialization routine of the MOB_ImageFlow class then automatically adds all other objects necessary to perform its task, as can be seen in Listing 8.3.

```
1 // Find all relevant textures containing a menu entry
2 for (uint32 i = 0; i < GetNumChildren(); i++)
3 {
4   BMC_ScriptObject* child = BMC_ScriptObject::GetChild(i);
5   MGL_TextureDefinition* texture = dynamic_cast<MGL_TextureDefinition*>(child)
      ;
6   if (texture != 0)
7   {
8     m_image_ids.push_back(texture->GetId());
9   }
10 }
11
12 // Create necessary nodes for each entry
13 for (uint32 i = 0; i < m_image_ids.size(); i++)
14 {
15   MGL_TextureReference* texture =
16     dynamic_cast<MGL_TextureReference*>(
17       BMC_Object::Create(L"MGL_TextureReference"));
18   texture->SetId(m_image_ids[i]);
19
20   MGL_MaterialDefinition* material =
21     dynamic_cast<MGL_MaterialDefinition*>(
22       BMC_Object::Create(L"MGL_MaterialDefinition"));
23   material->SetSorting(
24     BMC_EnumBaseStrList::GetValue(L"MGL_Sorting::BACK_TO_FRONT", 0));
25   material->SetAlphaBlending(
26     BMC_EnumBaseStrList::GetValue(L"MGL_AlphaBlending::ON", 0),
27     BMC_EnumBaseStrList::GetValue(L"MGL_BlendFactor::SRC_ALPHA", 0),
28     BMC_EnumBaseStrList::GetValue(L"MGL_BlendFactor::ONE_MINUS_SRC_ALPHA", 0))
        ;
29   material->AddChild(texture);
30
31   MGL_Plane* plane =
32     dynamic_cast<MGL_Plane*>(
33       BMC_Object::Create(L"MGL_Plane"));
34   plane->AddChild(material);
35
36   MGL_Scale* scale =
37     dynamic_cast<MGL_Scale*>(
```

```
38        BMC_Object::Create(L"MGL_Scale"));
39    scale->SetScaleFactor(300.0, 300.0, 300.0);
40    scale->AddChild(plane);
41
42    MGL_Rotate* rotate =
43      dynamic_cast<MGL_Rotate*>(
44        BMC_Object::Create(L"MGL_Rotate"));
45    rotate->SetRotationAxis(0.0, 1.0, 0.0);
46    rotate->AddChild(scale);
47
48    MGL_Translate* translate =
49      dynamic_cast<MGL_Translate*>(
50        BMC_Object::Create(L"MGL_Translate"));
51    translate->SetPosition(-400.0, 300.0, 50.0);
52    translate->AddChild(rotate);
53
54    BMC_Node::AddChild(translate);
55  }
```

Listing 8.3: A code fragment of the `MOB_ImageFlow` initialization routine used for programmatically creating a sub-graph depending on a given set of input textures.

8.3 Vertex and Fragment Shaders

The following example describes how GLSL vertex and fragment shaders can be directly specified from the scene graph XML file. It further demonstrates the application of multi-pass materials with varying parameters for each individual pass; an implementation of Lengyel's algorithm for rendering real-time hair [11] serves as a visually attractive example. See Figure 8.4 and Listing 8.4.

Figure 8.4: A furry and a bump-mapped torus both rendered using vertex and fragment shaders, with the furry one using a multi-pass material to render 16 distinct shells of fur.

```
1 <!-- Shader Program Definition -->
2 <MGL_ProgramDefinition id="fur_prog">
3   <MGL_ShaderDefinition type="MGL_ShaderType::FRAGMENT"
4     source="
5       uniform float lightness;
6       uniform float distance;
7       uniform float scale;
8       uniform sampler2D fur_texture;
9       uniform sampler2D color_texture;
10      varying vec3 tangent1, tangent2, normal, lightDir, eyeVec;
11
12      void main (void)
13      {
14        vec3 N = normalize(normal);
15        vec3 L = normalize(lightDir);
16        vec3 gravity = vec3(-1.0, 0.0, 0.0);
17        float dtu = dot(tangent2, gravity);
18        float dtv = dot(tangent1, gravity);
19        vec2 texcoord = gl_TexCoord[0].xy * 1.0;
20        texcoord.x += dtu * distance * distance* 0.00035;
21        texcoord.y += dtv * distance * distance* 0.00035;
22        vec4 tex_color = texture2D(fur_texture, texcoord);
23        tex_color.rgb *= texture2D(color_texture, texcoord).rgb;
24        vec4 final_color =
25          (gl_FrontLightModelProduct.sceneColor * gl_FrontMaterial.ambient
26            * tex_color) +
27          (gl_LightSource[0].ambient * gl_FrontMaterial.ambient * tex_color);
28        float lambertTerm = dot(N,L);
29        if (lambertTerm > 0.0)
30        {
31          final_color += gl_LightSource[0].diffuse *
32                         gl_FrontMaterial.diffuse * tex_color *
33                         lambertTerm;
34          vec3 E = normalize(eyeVec);
35          vec3 R = reflect(-L, N);
36          float specular = pow( max(dot(R, E), 0.0),
37                           gl_FrontMaterial.shininess );
38          final_color += gl_LightSource[0].specular *
39                         gl_FrontMaterial.specular *
40                         specular;
41        }
42        final_color.a = tex_color.a * (1.0 - (distance / 24.0));
43        final_color.rgb = final_color.rgb * lightness;
44        gl_FragColor = final_color;
45      }"/>
46   <MGL_ShaderDefinition type="MGL_ShaderType::VERTEX"
47     source="
48       uniform float lightness;
49       uniform float distance;
50       uniform float scale;
51       varying vec3 tangent1, tangent2, normal, lightDir, eyeVec;
52
53       void main()
54       {
55         normal = gl_NormalMatrix * gl_Normal;
```

```
56         tangent1 = gl_NormalMatrix * gl_MultiTexCoord7.xyz;
57         tangent2 = cross(tangent1, normal);
58         vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
59         lightDir = gl_LightSource[0].position.xyz;
60         eyeVec = -vVertex;
61         vec4 n;
62         n.xyz = gl_Normal;
63         n.w = 0.0;
64         gl_TexCoord[0] = gl_MultiTexCoord0;
65         gl_Position = gl_ModelViewProjectionMatrix *
66           (gl_Vertex + n * distance * scale);
67       }"/>
68 </MGL_ProgramDefinition>
69
70 <!-- Material Definition -->
71 <MGL_MultiPassMaterialDefinition
72   id="mat_fur"
73   diffuse_color="255, 255, 255"
74   ambient_color="255, 255, 255"
75   specular_color="25, 25, 0"
76   shininess="10"
77   multi_pass_type="MGL_MultiPassType::PER_OBJECT"
78   alpha_blending="MGL_AlphaBlending::ON"
79   sorting="MGL_Sorting::BACK_TO_FRONT"
80 >
81   <MGL_ProgramReference id="fur_prog"/>
82   <MGL_TextureReference id="map_fur" unit="0"/>
83   <MGL_TextureReference id="map_leopard" unit="1"/>
84   <MGL_ProgramParameters
85     params="sampler2D fur_texture = 0;
86             sampler2D color_texture = 1;
87             float scale = 0.4;
88             float lightness = 1.0;
89             float distance = 0.0;"/>
90   <MGL_MaterialPassDefinition alpha_blending="MGL_AlphaBlending::OFF">
91     <MGL_ProgramParameters params="float distance = 0.0;"/>
92   </MGL_MaterialPassDefinition>
93   <MGL_MaterialPassDefinition>
94     <MGL_ProgramParameters params="float distance = 1.0;"/>
95   </MGL_MaterialPassDefinition>
96   <MGL_MaterialPassDefinition>
97     <MGL_ProgramParameters params="float distance = 2.0;"/>
98   </MGL_MaterialPassDefinition>
99   ...
100  <MGL_MaterialPassDefinition>
101    <MGL_ProgramParameters params="float distance = 15.0;"/>
102  </MGL_MaterialPassDefinition>
103 </MGL_MultiPassMaterialDefinition>
```

Listing 8.4: An XML fragment describing the definition of a vertex/fragment shader pair for rendering a single shell of fur followed by the definition of a multi-pass material applying this shader program to 16 distinct shells.

8.4 The Background as a Texture

Another interesting way of combining software-rendered 2D and OpenGLrendered 3D content is to use the whole background surface as a texture on an arbitrary geometric object, e.g. for applying distortion effects on the background or virtually zooming away. For this reason, a `MOB_Scene` node now accepts an additional attribute named `disable_background_display` that, when set to one, forces the renderer engine to suppress rendering of its screen-aligned background quad whenever that scene is active. It is then up to that scene to fill the screen with 3D-only content. However, the background surface and its underlying texture are still constantly updated, and can be accessed via an `MGL_TextureReference` with the predefined ID ”BACKGROUND_TEXTURE”. The example shows how this mechanism can be used to smooth-ly zoom out when switching between two scenes; see Figure 8.5 and Listing 8.5.

Figure 8.5: An example illustrating the possibility of integrating the 2D surface as a texture object on 3D geometry.

```
1 <!-- Scene definition -->
2 <BMC_Scene name="FREEGAME SCENE">
3   <MOB_Screen name="FREEGAME SCREEN"
4     device_id="GFX_DeviceType::MAIN"
5     disable_background_display="1"
6   >
7     <MOB_Object name="BOTTOM SCREEN" x="0" y="600" z="0" w="800" h="600">
8       <MOB_AnimationSingle animation = "resource('screen_background.png')"/>
```

```
 9        <!-- Regular screen contents -->
10        ...
11      </MOB_Object>
12      <!-- 3D representation -->
13      <MOB_GlRendererArea id ="freegame_context" x="0" y="0" w ="800" h="1200">
14        <!-- Materials -->
15        <MGL_MaterialDefinition id="bg_mat"
16          ambient_color="255, 255, 255"
17          diffuse_color="255, 255, 255"
18          specular_color="255, 255, 255"
19          face_lighting="MGL_FaceLighting::OFF"
20        >
21          <MGL_TextureReference id="BACKGROUND TEXTURE"/>
22        </MGL_MaterialDefinition>
23        <!-- Geometry -->
24        <MGL_EffectLoop
25          name="rotate_freegamescene_effectloop"
26          start_on_activate_screen="1"
27          execution_behaviour = "EXECUTION_BEHAVIOUR::SEQUENTIAL"
28          number_of_loops="1"
29          controllers.1 = "MCO_EffectRotateObject(Object=../
              translate_freegamescene/rotate_freegamescene,AnimationTime=0,
              ToAngle=0)"
30          controllers.2 = "MCO_EffectRotateObject(Object=../
              translate_freegamescene/rotate_freegamescene,AnimationTime=3000,
              ToAngle=25)"
31        />
32        <MGL_EffectLoop
33          name="translate_freegamescene_effectloop"
34          start_on_activate_screen="1"
35          number_of_loops="1"
36          controllers.1 = "MCO_EffectMoveObject(Object=../
              translate_freegamescene,AnimationTime=3000,FromX=400,ToX=260,FromY
              =600,ToY=500,FromZ=0,ToZ=-180)"
37        />
38        <MGL_Translate name="translate_freegamescene" x="400" y="600" z="0">
39          <MGL_Rotate
40            name ="rotate_freegamescene" angle="0"
41            axis_x="0.6" axis_y="-1.0" axis_z="0.0">
42            <MGL_Scale x="800" y="1200" z="1">
43              <MGL_Plane tx1="0.0" ty1="0.0" tx2="0.78125" ty2="0.5859375">
44                <MGL_MaterialReference id="bg_mat"/>
45              </MGL_Plane>
46            </MGL_Scale>
47          </MGL_Rotate>
48        </MGL_Translate>
49      </MOB_GlRendererArea>
50    </MOB_Screen>
51  </BMC_Scene>
```

Listing 8.5: An XML fragment showing how to disable regular background rendering for a given content scene and integrate it into custom 3D geometry.

Chapter 9

Results and Discussion

Having presented all the necessary steps – from initial considerations to the final implementation – for transforming the Atronic EGD framework into a hybrid 2D/3D rendering system in the course of this thesis so far, this chapter concludes with an evaluation of the developed prototype with respect to daily-use suitability, followed by a discussion about possible future improvements to leave behind the prototype stage towards a mature product.

9.1 Performance Measurements

Considering the actual primary field of operation for the hybrid EGD prototype, the following sub-sections describe the results of a number of performance measurements carried out on the *Synergy* platform with the *Mesa* graphics driver under Linux; these measurements were performed in a "real-world" scenario running an already existing game, with the EGD framework's default screen resolution of 800x1200 pixels for 4:3 displays (800x600 pixels in a dual-screen configuration) and an extended wide-screen resolution of 960x1200 pixels for 16:10 displays.

9.1.1 Surface Transfer Performance

On the *Synergy* platform using the open-source *Mesa* DRI driver, as can be seen from the preliminary performance measurements carried out in chapter 4, transfer of the background surface to video memory shows the best performance when performing a full update of the target texture via `glTexImage2D()` utilizing a double-buffered PBO scheme.

Regarding the two target screen resolutions of 800x1200 pixels and 960x1200 pixels,

the actual amount of pixel data to transfer is roughly of the same order of magnitude as the preliminary test application's 1024x1024 resolution. Figures 9.1 and 9.2 show a comparison of maximum achievable frame rates and CPU load, respectively; for each of the two resolutions, measurements were performed in "idle mode" with no actual 2D rendering taking place, and during "reelspin" with roughly 30 percent of the whole screen being constantly updated.

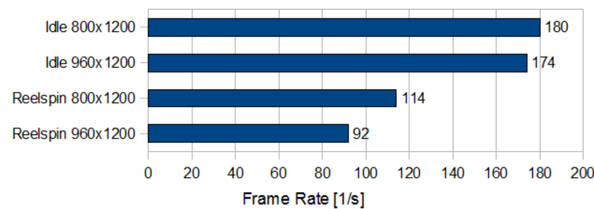

Figure 9.1: EGD texture update performance: Maximum achievable frame rate for different screen resolutions and varying load.

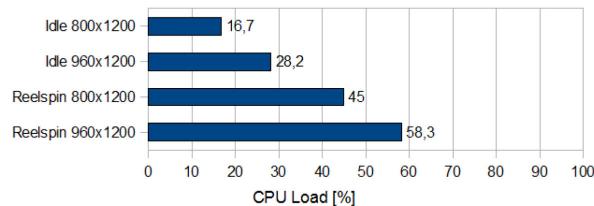

Figure 9.2: EGD texture update performance: CPU load (single core) for different screen resolutions and varying load, at a fixed frame rate of 60 fps.

As can be seen from these results, although the maximum achievable frame rate in idle mode is almost equal for both screen resolutions, CPU load is almost 70 percent higher at the 960x1200 resolution. During a deeper analysis of this somewhat odd behavior, it was possible to identify a call to OpenGL's `glUnmapBuffer()` function as the source of the problem; subsequently decreasing the vertical size of the surface by one row remedied this problem at a resolution of 960x1092 pixels. Looking up the source code of the driver implementation revealed that the driver internally only operates on buffer objects of power-of-two sizes; crossing the 4 MiB border at a resolution of 960x1093 pixels yields an internal buffer rounded up to 8 MiB in size thus producing the observed overhead.

9.1.2 Rendering Performance

For the purpose of measuring actual 3D performance of the hybrid prototype, the bottom half of the game's 960x1200 main screen was overlaid with a configurable number of real-

time rendered coin objects, each consisting of 66 individual vertices and 128 triangles. Repeatedly falling down from the top of the screen, each of these coins is individually rotated around a randomly chosen axis; a sample screen shot is shown in figure 9.3.

Figure 9.3: Screen shot of the "coin flow" performance test, with 250 individually animated coins rendered with a DOT3 bump mapping shader.

In addition to a varying number of individual coin instances, this test was also carried out with respect to three materials differing in complexity: one cycle used a plainly colored material with only lighting enabled, the second cycle additionally featured a single texture applied to each coin, and the third cycle was performed using a DOT3 bump mapping shader program with a color texture and a normal map enabled on two texture channels. Figures 9.4 and 9.5 show a comparison of these measurements.

As expected, the resulting frame rate quickly drops to a crawl at a high number of object instances. Nevertheless, an instance count of 100 still yields an acceptable frame rate of around 60 fps at relatively low CPU utilization; sufficient CPU resources are available for performing other tasks. Considering the overhead imposed by the continuous update of the background surface, these values allow for an undisturbed operation in a practical environment.

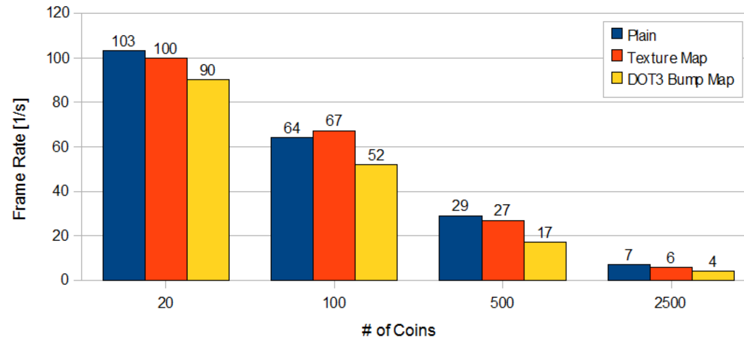

Figure 9.4: Rendering performance: Maximum achievable frame rate for the "coin flow" test case, with varying instance count and three materials of varying complexity.

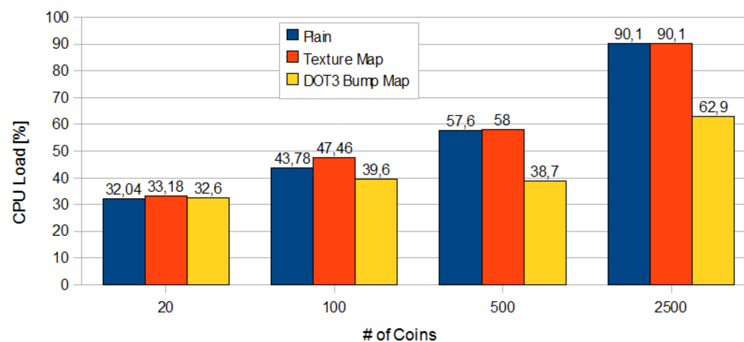

Figure 9.5: Rendering performance: Single core CPU load for the "coin flow" test case, with varying instance count and three materials of varying complexity, at a fixed frame rate of 60 fps.

9.2 Future Improvements

9.2.1 Improved Parallelization

Through the analysis of the interaction between *drawing thread* and *flip thread* depicted in figure 5.4, it becomes apparent that the (somewhat conservative) approach of integrating the actual OpenGL-specific rendering process between the end of a `Draw()` traversal and the actual frame buffer swap leaves open two essential gaps where both CPU and GPU may be idle for some time, hindering optimal parallelization of 2D and 3D rendering. On the one hand, the OpenGL subsystem cannot access the 2D surfaces it depends on before the `Draw()` traversal is finished; on the other hand, the 2D renderer must not modify its surfaces before OpenGL has finished transferring them to video memory.

Solving this problem might be accomplished by introducing additional double-buffer

schemes for both the system memory background surface as well as all renderer objects in use, and by enforcing the double-buffer PBO scheme for all regular textures. Through such a mechanism it would be ensured that both the 2D and 3D rendering modules only access any of their needed resources invisible to the other module. As a consequence, synchronization between the two threads might be implemented more liberally, and idling would only occur with unbalanced workload on both modules, and not out of an innate habit.

From the actual implementation's point of view, applying these changes is not an overly complicated task; introducing a double buffer for the background surface is easily accomplished, and double-buffered renderer objects can be created in a way transparent to the user. However, the increased resource consumption of this approach regarding vertex or pixel buffer objects in video memory might introduce new problems on low-end platforms such as *Synergy* with only 256 MiB of dedicated video memory.

9.2.2 Higher-Level Node Classes

With the newly introduced OpenGL-related MGL scripting nodes forming the basis for hardware-accelerated 3D rendering in the EGD framework, further extensions in the form of higher-level classes can be seen as a logical consequence; the existing *Multimedia Game Base* library with its various application-specific implementations atop the *Multimedia Base* library already provides a good foundation in the field of 2D rendering. Future 3D-based higher-level node classes might exemplarily implement the following ideas or concepts, based on the extensions making up the hybrid prototype:

- Replacement of the 2D ”reelslot” classes to allow for advanced visual effects in classic casino slot machines;
- Real-time rendering of a 3D roulette bowl, possibly zooming in on the bowl or the whole table;
- 3D jackpot meters as demonstrated in chapter 8; and
- A generic particle engine.

Through the open and extensible nature of the framework however, no restrictions are imposed concerning any other design ideas yet to come: The sky's the limit.

Chapter 10

Conclusion

This thesis, motivated by the venture of enhancing Atronic's proprietary 2D-only *Easy Game Development* framework with the capability of hardware-accelerated real-time 3D rendering, presented the course of action during this challenge from initial thoughts to a final prototype. Starting out by reviewing the notion of a *scene graph* and EGD's essential relationship to this concept in chapter 2, the thesis highlighted its resemblances and differences to other scene graph frameworks and toolkits established in the market.

Having obtained an understanding of the project's cornerstones from these initial comparisons as well as the following process of acquiring detailed requirements for the ultimate goal of establishing a real-time capable 3D rendering framework, the project's primary direction began to manifest itself in the form of a hybrid system with the distinctive characteristic of coexistence as well as mutual benefit between 2D and 3D subsystems. Resulting from this process described in detail in chapter 3, the individually collected goals and requirements can be summarized by three essential items:

- Seamless integration into the existing framework on the user level;
- optimal utilization of existing 2D implementations in the framework; and
- parallelization of CPU-based 2D rendering and GPU-based 3D rendering.

In order to guarantee a successful outcome of any efforts invested into this work in the first place, an important role was played by a deeper analysis of the various required target platforms comprising a number of different hardware architectures and operating systems, in terms of overall performance, stability and feature completeness. As a positive result of the observations and preliminary measurements made during this crucial analysis

carried out in chapter 4, the initial project goal with all its drafted requirements showed to be well within reach; the path was cleared for further work in the desired direction.

In the course of discussing overall prototype concepts in chapter 5, three particular approaches for augmenting the existing EGD scene graph hierarchy were presented; subsequent elaboration on the idea of a semi-homogenous graph structure led to further concretization towards a reliable and easy-to-use design. Depicting details of the actual implementation regarding user-visible scripting extensions and their relationship to the underlying internals in chapter 6 provides an additional means to obtain a clear understanding of object interaction and data flow in the developed prototype.

Concluding the work with the results from a variety of practical application examples shown in chapter 8 and corner case performance measurements from chapter 9 to put the resulting prototype through its paces, one can safely consider it not to be just "a flash in the pan". Furthermore, after having merged the prototype code to Atronic's main project trunk, and having successfully piloted it past a thorough system testing phase under the critical eyes of the in-house testing team, observing the final product's overall smooth operation in a genuinely practical environment at *Casinos Austria* certainly opens up a promising perspective.

Bibliography

[1] Tomas Akenine-Moller, Tomas Moller, and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[2] Autodesk Inc. *Autodesk FBX*. `http://area.autodesk.com/fbx/`. Online; last visit: 16-Nov-2009.

[3] Web3D Consortium. *The Virtual Reality Modeling Language*. `http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/`. Online; last visit: 20-Oct-2009.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[5] IBM Corporation and Microsoft Corporation. *Multimedia Programming Interface and Data Specifications 1.0*. `http://www.kk.iij4u.or.jp/~kondo/wave/mpidata.txt`. Online; last visit: 25-Oct-2009.

[6] Microsoft Corporation. *Resource Interchange File Format Services*. `http://msdn.microsoft.com/en-us/library/ms713231(VS.85).aspx`. Online; last visit: 14-Nov-2009.

[7] Christopher Dissauer. *DZY Animation Format*. Unpublished; Atronic internal document, 2004.

[8] Thomas Gößler. *Linked Presentation System Architecture*. Unpublished; Atronic internal document, 2008.

[9] Karsten Juschus, Andreas Richter, and Behnam Tabatabai. *Hibility Module Structure*. Unpublished; Atronic internal document, 2006.

[10] Khronos Group. *OpenGL 2.1 Reference Pages*. `http://www.opengl.org/sdk/docs/man/`. Online; last visit: 10-Oct-2009.

[11] Jerome Edward Lengyel. *Real-Time Hair*. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 243–256, London, UK, 2000. Springer-Verlag.

[12] Andreas Oberdorfer. *blit.dll Interface*. Unpublished; Atronic internal document, 2003.

[13] John Rohlf and James Helman. *IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics*. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, New York, NY, USA, 1994. ACM.

[14] Jeremy Sandmel and Jeff Juliano. *EXT_framebuffer_object*. `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`. Online; last visit: 10-Oct-2009.

[15] Paul S. Strauss. *IRIS Inventor, a 3D graphics toolkit*. *SIGPLAN Not.*, 28(10):192–200, 1993.

[16] Paul S. Strauss and Rikk Carey. *An object-oriented 3D graphics toolkit*. *SIGGRAPH Comput. Graph.*, 26(2):341–349, 1992.

[17] Behnam Tabatabai. *MMI, CCI, MOBs, and Assets*. Unpublished; Atronic internal document, 2004.

[18] Behnam Tabatabai and Christoph Knebelreiter. *An Overview of Resource Management in Hibility Multimedia Software V2.0*. Unpublished; Atronic internal document, 2004.

[19] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.