A Direct3D 11 Implementation of the Unicon 3D Facilities

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Fabian Mathijssen

January 2015

Major Professor: Clinton Jeffery, Ph.D.

Authorization to Submit Thesis

This thesis of Fabian Mathijssen, submitted for the degree of Master of Science with a Major in Computer Science and titled "A Direct3D 11 Implementation of the Unicon 3D Facilities," has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ Date: _____

Clinton Jeffery, Ph.D.

Committee Members: _____ Date: _____

Robert Rinker, Ph.D.

_____ Date: _____

Jim Alves-Foss, Ph.D.

Department Administrator: _____ Date: _____

Gregory Donohoe, Ph.D.

Discipline's College Dean: _____ Date: _____

Larry Stauffer, Ph.D.

Final Approval and Acceptance

Dean of the College of Graduate Studies: _____ Date: _____

Jie Chen, Ph.D.

Abstract

Unicon is a high-level procedural programming language with built-in graphics facilities. We implement a Direct3D 11-based graphics subsystem to be used by Unicon on Windows platforms as an alternative to the existing OpenGL-based implementation. Unicon may be configured to use this subsystem at compile-time through detection of the host platform or user configuration. The existing facilities are several years old and rely on the feature set provided by OpenGL 1.2. As such, they do not make efficient use of the processing power provided by modern graphics hardware. The Direct3D-based graphics improve on the performance of the OpenGL-based graphics through the use of shaders. The performance of the new graphics subsystem is measured through benchmarks that simulate Unicon virtual environments of varying complexities. These results will help ensure Unicon's compatibility with Windows OSs. They have also led to changes in Unicon's platform-independent set of graphics functions that improve the portability of the language.

Acknowledgements

I would like to sincerely thank my thesis adviser, Dr. Clinton Jeffery, for his unrelenting support and patience over the course of my thesis work. His efforts have been instrumental in helping me overcome what in retrospect I consider deficiencies in essential skills.

Additionally, I would like to extend my gratitude to my thesis committee members, Dr. Alves-Foss and Dr. Rinker, for the time and effort vested in reviewing this work.

Finally, I would like to thank my parents and my sister, for without their patience and encouragement I would not have been able to attain a master's degree in an already tumultuous time of my life.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1: Introduction

With the advent of three-dimensional video games, digital stores wherein customers maintain a virtual presence to purchase goods, and various types of scientific visualization software, 3-D computer graphics have become synonymous with interactive virtual environments. However, thus far, only highly skilled individuals and professional software publishers possess the tools and expertise to create such environments and to make them available to a large audience as collaborative online software. Providing tools to build such environments to individuals and groups with backgrounds outside of computer graphics could facilitate their more widespread use. In this thesis, we contribute to this cause by providing easy access to a low-level, specialized graphics application programming interface (API) through a high-level language.

Although programming languages exist that provide high-level bindings to low-level 3-D graphics APIs, these languages are often special-purpose languages and therefore lack the data structures and expression semantics needed for fine-grained 3-D scene manipulation. This is not the case for the programming language Unicon [1], which was built on top of the language Icon [2] and has been in development at the University of Arizona and the University of Idaho [3]. Unicon has helped remedy the lack of a portable, easy-to-use high-level graphics programming language by being usable on a multitude of operating systems and processor architectures.

1.1 Introduction to Unicon

Unicon is a high-level interpreted procedural and object-oriented programming language with an emphasis on string processing. One of the ways it expands on its predecessor, Icon, is in providing accessible graphics facilities though a set of abstract high-level 2-D and 3-D graphics instructions, which are translated into low-level instructions by a graphics subsystem. The current implementation of this subsystem uses OpenGL, but large portions of Unicon's runtime system were kept modular to allow other implementations to be added in the future.

Environments written in Unicon execute within an operating system- and processor architecture-specific runtime environment. In order to attain the highest possible performance for running Unicon 3-D programs, one has to consider whether using an alternative implementation of the graphics subsystem improves performance while potentially enjoying greater developer support on a given host platform. To this end, we investigate the performance of Unicon on Windows machines with a reimplementation of the graphics subsystem that uses Microsoft's Direct3D API to handle Unicon's graphics output. When the Unicon source code is compiled, a target host platform and graphics subsystem are selected either implicitly by the Unicon compiler's configuration, or explicitly by compiler command-line options. The addition proposed in this document will allow Unicon programs that execute on a Windows platform to use a Windows-specific graphics API that is built into the operating system.

1.2 Motivation for the Use of Direct3D

The aim of this contribution to the Unicon project is to provide a more up-to-date, Direct3D 11-based implementation whose performance meets or exceeds that of the current implementation on Windows platforms.

One motivation for using Direct3D 11 instead of OpenGL on Windows is the fact that the OpenGL implementation of Unicon's graphics relies on an outdated version of OpenGL, OpenGL v1.2. It works on all subsequent desktop versions of OpenGL thus far, but it does not use newer and more efficient OpenGL constructs such as index buffers and shaders. Older features like display lists may still work if the programmer supplies a compatibility profile [4]. OpenGL v1.1 and v1.2 code may also not run on any version of OpenGL ES – a subset of the OpenGL API for mobile use – since the OpenGL ES v1.0 standard was written against the OpenGL v1.3 specifications. Porting Unicon's graphics facilities over to the most recent version of OpenGL to ensure portability may also prove problematic because not all operating systems support all versions of OpenGL, and modern operating systems are slow to adopt new OpenGL standards, thereby making the choice to support a specific version of OpenGL arbitrary. Mac OS v10.8 "Mountain Lion", supported at most OpenGL v3.3, 2.5 years after the OpenGL 4.0 specification had been published, and even the most recent version of Mac OS (10.10 "Yosemite") does not support many features added in the past two years on hardware that is readily capable of the features in question. OpenGL feature support on Windows machines depends entirely on which features are supported through the graphics card drivers from hardware manufacturers such as Nvidia, AMD, Matrox and Intel.

Microsoft is currently not in the same predicament. Direct3D is integrated in Windows, and starting with Direct3D v11.0, Microsoft ensures complete uniformity in feature level support across all versions of Windows, but with Windows 8 and 8.1 supporting additional features through incremental updates (Direct3D v11.1 and v11.2). However, applications written to use features added in versions 11.1 and 11.2 retain backward compatibility with Direct3D 11.0, which is fully supported in Windows Vista, 7, 8 and 8.1. In essence, any application written to use any version of Direct3D 11 will run on any Windows OS released in January of 2007 or later, and will be fully supported in the foreseeable future. Microsoft has thereby chosen to break Direct3D's compatibility with unsupported versions of Windows (Windows XP and earlier) in favor of greater uniformity across all supported versions of Windows. Additionally, Microsoft includes software-based rasterizers with full Direct3D 11 support in Windows Vista, 7, 8 and 8.1, which means hardware support in the form of graphics drivers is unnecessary to use Direct3D [5].

Additionally, Microsoft has gone to great lengths to encourage graphics development in C/C++/C# with Direct3D by providing excellent online resources in the form of documentation, programmer forums, and the monitored open-source code distribution platform CodePlex [6]. Microsoft also provides free

state-of-the-art IDEs with advanced shader debugging features, such as Visual Studio For Desktop 2013 Express [7].

Due to Unicon's platform-independent nature, a Windows-only addition to Unicon is questionable on the one hand yet justifiable on the other. Re-implementing the graphics facilities in the latest version of Direct3D will allow Unicon 3-D facilities to run well on Windows platforms for years to come. OpenGL support on Windows has been uncertain in the past, and the prevalence of Windows on personal computers is ample reason to aim to remain compatible with it. Because Direct3D is a closed-source commercial product that has gathered a large amount of developer support more than ten years after its inception, its development cycles have rendered it a more unified experience than OpenGL. Direct3D's performance may or may not exceed that of OpenGL in certain scenarios [8], but a reimplementation of Unicon's 3-D graphics subsystem through an API that is about ten years newer than the one used in the OpenGL implementation is likely to make more efficient use of modern hardware and to yield better performance.

1.3 Why Use Direct3D 11 Now?

From its latest major update (version 11) onward, Direct3D unifies the hardware requirements for graphics processing units (GPUs) that aim to support Direct3D, which makes this version a logical one to re-implement Unicon's graphics facilities in: all future versions of Unicon will run on all GPUs that implement Direct3D 11 because Direct3D 11 does not allow differentiation among GPU instruction sets at the feature set level other than the amount of available graphics memory and types of supported full-screen anti-aliasing support.

The timing of this project avoided a major portability limitation in Direct3D: Microsoft removed parts of the original Direct3D 11.0 specification for Windows 8 and newer platforms. A widely used series of graphics libraries collectively called *D3DX* can be used for development on Windows Vista and 7, but not on Windows 8. This was an issue because one needs to use Windows 8.1 and a recent version of Visual Studio to create several types of Windows-specific applications, such as Windows Store apps. In order not to hinder future development of Unicon's Direct3D-based graphics by using deprecated Direct3D functions, all use of D3DX libraries were removed from the project.

Mathematical libraries such as *XNAMath* and *D3DXMath*, which contain vector and matrix operations for use with Direct3D 8, 9 and 10 were also considered deprecated (but supported) for development on Windows Vista and Windows 7, and became unsupported on Windows 8 and 8.1. All uses of XNAMath and D3DXMath were replaced with calls to the *DirectXMath* library, which lets the CPU use single-instruction-multiple-data (SIMD) instructions to speed up vector and matrix transformations.

Other developer tools, such as the *Effects* framework, which made writing complicated shaders easier, are deprecated but still supported in Windows 8.1. Note that the deprecated status of Direct3D

libraries is announced to varying extents: Microsoft's documentation on Effects does not mention it [9], while an MSDN blog post does [10]. Most libraries mention compatibility information explicitly.

The switches described above allowed the project to use the Windows SDK in Windows 8 without requiring the older DirectX SDK to be installed, too. The Windows SDK subsumes most of the functionality of the DirectX SDK in Windows 8 and 8.1.

Lastly, in the grand scheme of graphics API design, Microsoft and GPU designers have been working for years to make Direct3D a lower-level API, which is to say that the rendering speed one may get out of graphics hardware is mostly limited by how efficiently Direct3D accesses functions implemented directly on said hardware. For the next major release of DirectX, in an effort to bring "console-level" rendering efficiency to Windows platforms, Direct2D and Direct3D are expanded to let programmers configure the rendering pipeline more finely than before [11], at the cost of losing certain high-level functionality. It seems that this would make graphics development more complex for whomever is new to the field, and it would likely make the conversion of Unicon's graphics facilities to a Direct3D-based subsystem a greater challenge.

1.4 Implications of This Study

Efforts to help keep Unicon up-to-date and running optimally are not merely academic exercises, although the academic aspect plays a large role in Unicon development. Unicon is a language upon which other projects are built, such as the Collaborative Virtual Environment (CVE) [12]. CVE allows anyone to model buildings, such as those found on the University of Idaho campus, as three-dimensional virtual environments that may be used by students and educators alike as virtual meeting places. Such virtual environments can be readily found in applications like Second Life, but the fact that CVE and Unicon are open source projects and the implied customizability of CVE set Unicon and CVE apart from the competition. In a nutshell, CVE allows educators to set up virtual classrooms for college students to use as an additional online study resource, and performance improvements in Unicon's graphics subsystem will trickle down into CVE's usability.

Updating Unicon's graphics facilities also opens up new possibilities for future engineering students to improve on the Unicon language by expanding the graphics subsystem further. Adding or improving features of the Direct3D-based graphics implementation may also prove easier for new students looking to get involved with the Unicon project than working on the existing OpenGL-based subsystem.

A high-performance graphics component in a high-level programming language may have unforeseen benefits to unrelated or loosely related projects as well. The Software as a Living City initiative [13] is an example of a software project whose complexity could be reduced by using Unicon's graphics facilities.

1.5 Thesis Objective

The work carried out in this thesis research will result in a high-level C-callable graphics library that is built atop Direct3D 11.0. It is tailored toward being used in conjunction with Unicon and implements in C++ a large subset of Unicon's high-level 3-D graphics instructions. The resulting library will be usable in isolation as a standalone C-callable library that may be used with any C or C++ program, and is linked into Unicon to provide an abstraction of the Direct3D API to Unicon. The amount of work required to run Unicon 3-D programs such as CVE atop this new Direct3D-based library will be limited to adapting the Unicon runtime system to use the new library. This will already be done to the extent that the library source code is linked into and compiles with the Unicon source code. A standalone C program is used to test the library's functionality in order to ensure it is usable as a standalone library. Additionally, the performance and scalability of the new Direct3D-based library are measured with two benchmarks. These measure performance of the library while rendering virtual environments with a wide range of graphical complexities, expressed as the number of primitives or light sources in these environments.

Chapter 2: Background

This chapter places Unicon's 3-D graphics facilities in the context of graphics facilities built into other programming languages. Potential uses for Unicon's 3-D graphics are subsequently mentioned in their own section. Additionally, readers who are unfamiliar with computer graphics are recommended to review Appendix A, which includes descriptions of key computer graphics concepts, an overview of DirectX, and an overview of Direct3D's rendering pipeline.

2.1 Related Work

Unicon is a high-level programming language that excels at string operations, much like its predecessor Icon, which succeeded the language SNOBOL4, a string processing language from the 1960s. SNOBOL-style patterns were added to Unicon in 2005 [14]. However, pattern-based string processing is not the only focus SNOBOL4 and Unicon have in common. ESP3, the Extended SNOBOL Picture Pattern Processor [15], was proposed in 1975 as an addition to SNOBOL4 that provides facilities for constructing, accessing and naming portions of pictures, operations to test relationships between pictures, and patterns that describe classes of line drawings. ESP3's "DRAW" function allowed the programmer to draw several types of lines, much like Unicon's 3-D graphics facilities contain functions for drawing several types of geometric shapes such as spheres, cubes and tori. Although ESP3 provided only 2-D graphics support, Unicon provides functions to transform both 2-D and 3-D graphics.

Specifically, Unicon is a high-level language with an abstract binding to the low-level 3-D graphics API OpenGL and a proposed binding to Direct3D, which is added in this thesis. Other high-level language libraries exist that provide accessible 3-D graphics functions which are implemented with calls to low-level graphics APIs. Some of these map OpenGL or Direct3D functionality on procedural or object-oriented design paradigms, but most provide direct access the API's low-level C/C++ functions. Such language bindings are more common for OpenGL than for Direct3D, although managed bindings to DirectX are available for CLI languages[1]. Examples of CLI languages are IronPython, IronRuby and IronLisp, which are implemented atop a library that runs on the CLI.

JOGL [16], a Java binding for OpenGL, is part of a group of high-performance Java libraries for graphics, audio and multimedia processing called JOGAMP [17]. JOGL provides indirect access to the entire OpenGL library (versions 1.0-4.3 and OpenGL ES v1-3) and combines OpenGL's output with the Swing, AWT and SWT windowing toolkits. JOGL provides the graphics facilities for the Processing programming language, the Gephi graph visualization tool, and many other projects that

[1]The Common Language Infrastructure (CLI) is an umbrella term that refers to a group of language specifications (the *Common Language Specification* or CLS), an interpreted language (the *Common Intermediate Language* or CIL) into which languages are compiled that adhere to these specifications, and the .NET runtime system, which executes programs expressed in the CIL.

can be found at the JOGL homepage. The JOGL specifications state that JOGL provides an exact one-to-one mapping of the C functions in OpenGL's API to Java, with each GL interface packaged separately. The naming specifications for OpenGL functions and constants in JOGL are found in the JOGL project's specifications page [18]. JOGL also contains a Java binding for all core APIs of GLU, the OpenGL Utilities Library, but not the GLU NURBS APIs for drawing NURBS curves and surfaces.

WebGL [19] is a Javascript binding for OpenGL that adheres to the OpenGL ES 2.0 specification but retains backward compatibility with OpenGL ES 1.0. WebGL comes pre-installed with many modern browsers to allow GPU-accelerated GLSL[2] shaders to be executed by control code in Javascript as part of an HTML 5 "canvas" element. Example applications that use WebGL are Google Maps and AutoCAD 360. WebGL functions extend the Document Object Model (DOM), which is to say, the tree of objects that comprise a website is extended with a "gl" object that provides low-level OpenGL functions for binding buffers, creating transformation matrices, compiling shaders et cetera. These instructions do not abstract away from what is being drawn to the screen at all, and work at the same level of abstraction as Unicon's underlying Direct3D-based graphics subsystem.

The Open ToolKit (OpenTK) [20], which supersedes the Tao Framework [21], is a set of libraries that provide low-level .NET language bindings for OpenGL, OpenAL and OpenCL. It is written in C#, but because of its compliance with the Common Language Specification, OpenTK can be used with any .NET language. OpenTK provides a very low-level interface to OpenGL 4.4 and OpenGL ES 3.0, which means one has direct access to most OpenGL functions without abstracting away from their functionality. As a direct result, OpenTK comes with the official OpenGL, OpenAL and OpenCL documentation.

A DirectX binding for the OOP language D has been in development since August of 2004 and is as yet incomplete [22]. It is based on the older June 2010 version of the DirectX SDK, which means support for DirectX 11.1 and 11.2 is missing. It provides a one-to-one mapping of the entire function list contained in the original DirectX header files (d2d1.h, d3d11.h, dxgi.h, et cetera).

The managed DirectX APIs SharpDX [23] and SlimDX [24] are available for all .NET languages. The SharpDX and SlimDX packages are generated directly from the DirectX SDK headers. SharpDX therefore provides direct access to all DirectX functions up to version 11.1, while the most recent version of DirectX supported by SlimDX is 11.0. Example projects that use SharpDX are the Paradox game engine [25] and MonoGame [26]. SlimDX has been used in commercial video games such as *Star Wars: The Force Unleashed* and *Operation Flashpoint: Dragon Rising*.

Haxe [27] is one of the few high-level programming languages with built-in support for cross-platform shaders. Haxe allows the programmer to write vertex and pixel shaders in HxSL, the open source Haxe Shader Language, which can then be translated into shader code for either Direct3D (in the language HLSL) or OpenGL (in the language GLSL). HxSL is a work in progress. It is evident from its list of

[2]More precisely, the same shading language as used in OpenGL 2.0, i.e., v1.20.8.

functions and supported data types [28] that the level of abstraction HxSL uses is no higher than that of HLSL, and its strength therefore lies mostly in being a cross-platform shader language. Haxe has been used in the production of several successful video games and also supports Flash shaders.

The language Chestnut [29] was invented to expedite the process of performing multi-dimensional array operations on a GPU. It makes parallel GPU programming accessible to novices in a way similar to how Unicon makes 2-D and 3-D graphics facilities available to anyone new to computer graphics, and speeds up parallel programming for experts. Chestnut code is more compact than the corresponding CUDA-C code it translates into. Chestnut aims to simplify writing parallelizable code for problems that do not have embarrassingly parallel properties by allowing the programmer to think in sequential terms. Chestnut programs start out as either a GUI-based design in the provided *Chestnut Designer* tool, or a code section directly written in Chestnut. The resulting CUDA-C code can subsequently be optimized by hand. Chestnut is at its core a high-level imperative programming language that intelligently translates sequential "foreach" loops into a series of CUDA threads with their own contexts, offering a mix of functions that translate into compute shaders that execute on the GPU or traditional programs that execute on the CPU. Chestnut also has rudimentary graphics facilities in the form of manipulable screen pixels whose colors can be accessed through a built-in `display` function. Although Chestnut may appeal to inexperienced parallel programmers by turning Chestnut code or GUI designs into executables at the press of a button, Chestnut is highly application-specific because it automates array operations only.

2.2 Applications of Unicon

Unicon's 3-D graphics facilities lend themselves to fields where experimental software is written that requires rapid prototyping. Examples of such fields are commercial and scientific visualization software. One of the earliest research papers that explores the usability of a video game engine for the purpose of simulating combat situations for the U.S. Department of Defense [30] [31], did so along the dimensions of believability, authorability, accuracy and performance. The performance of the engine used in the study – version 2.5 of the Unreal Game Engine (UGE) [32] – was important because many different factors triggered updates in a visualization of a plot of virtualized combat terrain. In 1998, the UGE contained what was one of the two most advanced commercial graphics engines of its day, but the complexity of the virtual environments it was tasked to generate is comparable to what can be generated by Unicon today. Although the authors of the article concluded that accurate collision detection and improved authorability (i.e., creating 3-D models and artwork to improve the simulation's immersiveness and synchronizing scenario and terrain) were the aspects most in need of attention, these should be less of an issue in Unicon because of Unicon's limited built-in user-accessible functionality: while a goal for Unicon is to have many online users interact with virtual 3-D objects simultaneously, the number of ways in which users can modify persistent elements in the environment is limited. A similar multi-user

virtual environment that is in operation today is the Collaborative Virtual Environment (CVE) [12]. It uses Unicon to generate the geometry of the shapes its virtual environments consist of and thereby uses Unicon to provide users an accessible way to create virtual environments for use in CVE.

Unicon's high-level graphics facilities not only lend themselves well to creating virtual environments in pure simulation, but may also be useful in creating collaborative virtual environments that are shared and manipulated based on a user's interactions with his/her surroundings. Such an interweaving of virtual and physical realities was envisioned by Broll et al. in 2000 [33] to give shape to a new way of interacting with other people. Not only do virtual environments exist as imaginary realms in games, but they may also be part of the world around us using what is called a Virtual Round Table (VRT): a virtual environment based on virtual reality technology. The layout of the environment is based on the world around us and shared with other users, who may manipulate the state of the virtual environment through changes in the physical world. Interactions with physical objects may be translated to interactions with 3-D models, which are translated and submitted to others connected to the same virtual reality. Such interactions may be useful in an online learning environment, in urban planning or in the construction of buildings.

Another application area Unicon's 3-D graphics facilities could be used for is that of programming language execution monitors, which visualize the execution of software as part of the debugging process. Jeffery et al. worked on such an application using the Alamo language monitor architecture [34], which generalizes earlier efforts to create a monitor for the Icon language and which has aided in the development of monitors for ANSI C programs. Alamo is an event-driven monitor, which means a monitored program will often transfer control to the monitor and receive control back from the monitor in the form of event requests and reports. In such situations, the monitor receives information about which instructions are taking place and about the state of program memory at that point in time. By instrumenting the runtime system of a given programming language based on events derived from the language's grammar or more complex combinations of preconfigured events, Alamo receives a stream of events that can be evaluated at a semantic level. A high-level graphics programming language could then be used to visualize high-level events such a change in the layout of a vector of values.

To generalize the Alamo project, program and algorithm visualization (PAV) systems use graphics code to display the contents of data structures visually for educational purposes. As Urquiza-Fuentes and Velázquez-Iturbide explained in a survey of algorithm visualization software in 2009 [35], the interactiveness of a PAV system is particularly important for use in a classroom setting, where the student would ideally have access to an easily configurable graphics engine with attached code interpretation logic that visualizes data structures and program code. In Price et al.'s taxonomy of program visualization software [36], such applications are examples of algorithm visualizations and program visualizations. Unicon may not only enable instructors to build applications that visualize the contents of data structures, but it may also aid students in college courses on algorithm design in determining how efficient

a given algorithm is, the key being that program code may be read in by a visualization program and that it may look for flags in the code to determine what to visualize. Thereby, the notion that program flags can be used to instrument a renderer on which code sections to visualize extends beyond the idea that this has to happen at compile-time, as was the case in the Alamo project. For course instructors, Unicon would be substantially easier to learn than the more popular low-level graphics APIs such as Direct3D and OpenGL, and students will be able to see the layout of data structures more clearly than before.

Although Alamo represents a running application using 2-D graphics, such as to display tree structures as two-dimensional trees, the idea of visualizing software ecosystems as "living cities" [13] reaches much further: using three-dimensional visualization through an easy-to-use graphics programming language, we open up the possibility of modeling object-oriented software executions after a city. Class instances may be modeled as buildings and the information passed between them as cars on strips of highway that connect the buildings. The age of a class may be visualized as visible wear and tear on the buildings, and bugs may be visualized in a way that draws the programmer's attention to a code section in need of revision. What Unicon may help achieve is to allow programmers to write platform-independent visualization tools while foregoing the need to learn low-level and platform-specific graphics APIs.

Chapter 3: Research Question

The motivation behind this research project is twofold. On the one hand, although the OpenGL-based graphics subsystem is capable of rendering the environments thrown at it thus far, we are interested to know if it can be complemented with a Direct3D-based alternative without sacrificing VM performance. With the benefits of an alternative low-level graphics port for Unicon in mind (as described in the introduction), the research question is formulated as follows:

Does implementing a basic Direct3D 11-based rasterizer for Unicon yield viable graphics performance of the Windows distribution of Unicon?

In other words, in accordance with what is commonly stated in engineering research papers, we state that the outcome of this thesis is a viable product. The fact that the Direct3D graphics port is at its core a video game graphics engine, the question of what constitutes a viable product yields the following performance measures:

1. Number of frames per second (FPS).
2. Variance between frame draw times (frame time variance).
3. Performance scalability with increasing scene complexity: does the number of rendered frames per second decrease linearly with the number of shapes drawn to the screen?
4. Performance scalability in scenes with an increasing number of light sources.
5. Optional: Video RAM (VRAM) use in scenes of increasing complexities.

These measures are common among popular graphics engine benchmark suites such as PassMark Software's *PerformanceTest* [37]. Not all of these measures are equally relevant. FPS and frame time variance determine how smoothly rendered a virtual environment appears to the user, and also measures how responsive the virtual environment is to user input. These metrics are to be used to establish the performance in a number of base cases, such as rendering a blank screen and rendering readily available virtual environments in CVE. More importantly, we measure how frame rates and frame times hold up to a synthetic benchmark that renders scenes with increasing complexity. The Direct3D port may have no trouble rendering existing environments in CVE, but the FPS measure may hit a ceiling on slightly more complex environments (making the rendered image appear to stutter), or frame time variance may increase substantially (making performance seem inconsistent, which means graphical output delivered at high numbers of frames per second is interspersed with slowly rendered frames).

The number of used light sources lies between zero and eight. Varying the number of light sources and the number of objects the light sources interact with should affect performance of the Direct3D-based graphics port negligibly due to the low maximum number of light we use concurrently. This is

because lighting calculations in the Direct3D port are performed in parallel on the GPU, whereas they are performed on the CPU in the OpenGL-based graphics facilities.

VRAM use is less relevant and will not be evaluated, because the Direct3D graphics port will only run on Direct3D 11-capable graphics cards. Few Direct3D 11 cards exist with less than 512 megabytes of VRAM, which is guaranteed to be more than enough to store the geometry of even very complex scenes. We do not measure VRAM usage in the Direct3D-based facilities because accurate measurements of available VRAM are hard (if not impossible) to obtain: modern GPUs use various optimization techniques to increase, for instance, shader execution speed at the cost of space. VRAM is also shared with all Direct3D applications, including the window manager, and modern graphics chips use VRAM aggressively: more data is stored in VRAM if more VRAM is available, which means VRAM use depends on the amount of available VRAM, the manufacturer of the graphics chip and the graphics drivers used.

We expect the Direct3D port to yield better overall performance than the existing OpenGL-based graphics facilities. However, even if it does not, a rendering speed that translates to smooth visuals (meaning, frames rendered at 24 FPS or higher) is deemed viable. We do note that in practical use, a frame rate of 24 FPS, which is a standard in the movie industry, only appears smooth when the frame time variance is negligible.

The research question is answered in Chapter 6.

Chapter 4: UniconD3D Internals

This chapter explains how the new Direct3D-based graphics facilities work. It is written from the perspective of a standalone application and is supported with UML 2.0 diagrams where needed. These UML diagrams are included in Appendix B. The standalone Direct3D-based renderer is henceforth referred to as "UniconD3D". A discussion of how UniconD3D is integrated with Unicon is deferred to the next chapter. If the reader is not familiar with computer graphics or the Direct3D API, it is recommended to review Appendix A before reading on. Please note that UniconD3D offers three-dimensional graphics facilities only, it offers no functionality for drawing two-dimensional shapes.

4.1 3-D Model Composition

Complex 3-D models usually consist of groups of smaller, less complex models. At the lowest level, these models consist of primitive geometric shapes ("primitives") such as spheres, cones and tori. The implementation of models in UniconD3D is no different: a model is recursively defined as a collection of models and primitives, such that only the primitives contain geometry data and other information such as textures and materials. A shape's geometry is defined as a set of *vertices* and a set of *indices*. Vertices are the points in 3-D space that form a shape when properly connected. Indices are used to specify which vertices are connected. Maintaining separate lists of vertices and indices allows the programmer to abstract away from the actual coordinates of the vertices, and consider only a shape's topology in general terms by referring exclusively to its indices. This is explained further in Section 4.3. Indices also allow us to make repeated use of a single vertex by sharing it when specifying adjacent triangles (and by extension, adjacent quads) that share this vertex, however UniconD3D does not use this property. Materials, which are used for lighting calculations, are explained in Section 4.6. Models and primitives also contain IDs, names and transformation matrices so they can be moved, scaled and rotated. The described model hierarchy allows UniconD3D to propagate a transformation at the top of a model tree down to the model's constituent primitives. This makes it easy for the programmer to transform a model as a whole, rather than having to transform each contained primitive manually.

Model transformations are performed by transforming a model's transformation matrix; the model's geometry remains unchanged between frame renders. Instead, each time a frame is rendered, a model's transformation matrix is used to transform the model's vertices from local space to homogeneous clip space in a vertex shader. This design is not forced upon the programmer, but it is considerably cheaper to change only a model's transformation matrix and not its vertices, because matrix transformations in the vertex shader are performed by the GPU in a highly parallel fashion. Please refer to Appendix A for a brief explanation of transformation spaces and shaders.

Not all models are rendered at all times. UniconD3D groups models based on the virtual environment they belong in, where such an environment is defined as a plot of virtual land that is occupied by models.

This is the equivalent of a "game level" in video games. Only one such environment is rendered at any time. Each environment is described in a separate text file using a human-readable format. In these files, henceforth called *area files*, model trees are described as blocks of text containing model descriptions, primitive descriptions, light sources and environment maps. In accordance with the previous paragraph, each model description must contain at least one primitive description, either directly or indirectly through its descendant models. We include a brief example of an area file as Listing 4.1. Displaying the contents of one environment now becomes a matter of reading in the corresponding area file, calling a function that helps prepare the renderer render only the models in this area, and calling a function that renders the environment to the screen. Furthermore, the renderer is able to prevent prevent specific models from being rendered, so they can be hidden from the user at will.

Listing 4.1: An example area file that describes a single cube

```
Model {
    name:                Example_Model
    translation:         0.0 0.0 0.0
    scaling:             1.0 1.0 1.0
    rotation:            0.0 0.0 0.0

    Primitive {
        type:            cube
        name:            Example_Cube
        facelength:      1.0
        radius:          2.0
        translation:     0.0 0.0 0.0
        scaling:         1.0 1.0 1.0
        rotation:        0.0 0.0 0.0
        nrOfSlices:      10
        nrOfRings:       10
        material: {
            ambient:     0.2 0.2 0.2
            diffuse:     0.4 0.4 0.4 0.1
            specular:    0.2 0.2 0.2 0.1
        };
        textures {
            logo.jpg     0
        };
    }
}
```

The environment described in Listing 4.1 is stored in a file with the ".area" extension. A custom file format for area files, with its own structure, was chosen to facilitate its rapid implementation. A list of all such files is maintained in a separate file in the same directory as the area files, so that UniconD3D can load the contents of all area files after Direct3D has been initialized. The initialization process for Direct3D is described briefly in Appendix A.

Of special note is the concept of *culling*: the ability to configure the Rasterization Stage of Direct3D's rendering pipeline to show or hide triangles whose vertices appear in counterclockwise order as seen from the user's perspective. By default, the "back" side of a triangle, whose vertices appear in counterclockwise order and should be invisible to the user, is hidden, which means that the triangle's back side is culled. UniconD3D always disables back side culling for specific primitives. Disabling culling of front or back sides of primitive faces lets one implement a range of effects, such as transparency, and allows UniconD3D to display primitives like polygons and filled polygons regardless of which angle they are viewed from.

4.2 Retrieving Model Geometry

Classes exist in the UniconD3D code to match the layout of area files. The classes *Primitive*, *Model* and *Area* store their eponymous data structures, while an extra class called *ModelFactory* was added to help generate vertices and indices for the primitives. The class that orchestrates the interaction between model geometry and the Direct3D device and device context to render frames is the *Graphics* class. The Graphics class contains a list of Area objects, each Area object contains a list of Model objects, and each Model object contains lists of Model objects and Primitive objects.

Before Direct3D is initialized in the Graphics class, the list of Area objects is filled by reading in the contents of each area file and generating an Area object from each file. Each Area object is filled in by reading its own area file. While doing so, the Area class has access to the ModelFactory class to generate the lists of vertices and indices for each Primitive the Area contains. The ModelFactory class therefore consists entirely of static functions. The process of generating geometry information in the form of vertices and indices in the ModelFactory class is explained in the next section.

At the end of the process, the Graphics class proceeds to initialize Direct3D, which means retrieving the vertices and indices from all Primitives in all Models in all Areas and placing them in vertex and index buffers. When the vertex and index buffers are created, their contents are assigned to and subsequently copied into a vertex buffer and index buffer on the graphics adapter. In UniconD3D, the contents of the vertex and index buffers are immutable, which means these buffers would have to be recreated whenever a model or primitive is deleted or added at runtime. In order to hide a given model from sight in a running Unicon environment, UniconD3D may decide not to render it while keeping its geometry in the vertex and index buffers. This is how UniconD3D is able to render one environment instead of being forced to render them all at once: before an area is rendered, the number of indices the area contains and the offset in the index buffer for the area are computed and stored globally in the Graphics class, and only these indices – and by extension, the vertices they refer to – are rendered.

The fact that models in Unicon are primarily generated as primitive shapes influenced the design of the Primitive, Model and Area classes greatly. Ten types of shapes can be generated and combined

to create more complex shapes; algorithms to subtract one shape from another or to add one shape to another and omit duplicate or unnecessary vertices accordingly was omitted from UniconD3D because such functionality does not exist in Unicon. The programmer is also allowed to enter lists of vertices manually to create shapes that cannot (easily) be built by combining primitives, such as organic shapes. Lists of vertices for such shapes can be generated by using modeling software such as Maya or 3D Studio Max, exporting models to human-readable file formats such as OBJ and X3D, and extracting vertex lists from them. The rationale behind this design choice in Unicon was trying to strike a balance between offering a way to create shapes in a primitive-centric way that is accessible to anyone, while also allowing artists with access to high-fidelity 3-D models to import their models in Unicon with some extra work involved.

In conclusion, UniconD3D's approach to drawing area geometry allows for a clear division between models, which are complex, and primitives, which are simple. This ensures many operations may only be applied to primitives (e.g., texture operations) and simplifies the logic of gathering all vertices and indices contained in an Area.

4.3 Generating Geometric Primitives

This section deals with the problem of generating vertices and indices for the different geometric primitives Unicon supports. Additionally, if textures are applied to a primitive to give it a more realistic appearance, one must move, scale and rotate each texture so that it "wraps around" the primitive correctly. This problem is known as *UV mapping*, because placing a texture on a primitive face involves selecting a portion of the image with n pairs of (u, v) coordinates, and mapping these coordinates onto the n vertices that comprise the primitive face.

When the exact number of vertices and their positions are known ahead of time, as is the case with cubes and quads, generating their geometry is trivial. For most shapes however, the number of vertices depends on user-specified parameters, such as the degree of tessellation in a sphere. Because Direct3D will render a model face only if its indices are defined in a clockwise order as seen from the camera's perspective, the vertices the indices refer to must appear in clockwise order when they comprise a triangle that should be visible to the camera. Algorithms for generating a list of vertices, a list of appropriately ordered indices and a list of UV coordinates are given below for each primitive type. Note that UniconD3D only generates non-geodesic shapes: their faces are always constructed from convex quadrilaterals ("quads"), whereas a geodesic object is one whose shape is approximated by a minimal number of triangles instead of quads. To illustrate the difference, examples of a regular sphere and a geodesic sphere are shown in Figure 4.1.

Texture mapping is largely done automatically, the only information required from the user is information on which texture is applied to which primitive face, if any. Which primitive faces the user

can choose from is explained per type of primitive in the subsections below. The approach taken to texture mapping is in line with how textures are mapped to primitive faces in Unicon Technical Report 9c [38], with the exception of filled polygons.

Direct3D 11 has no built-in facilities for generating primitive shapes, as opposed to OpenGL and earlier versions of Direct3D. Third-party libraries exist that do this, but we chose to rely on these as little as possible. The algorithms for generating the different types of primitives described below are original work.

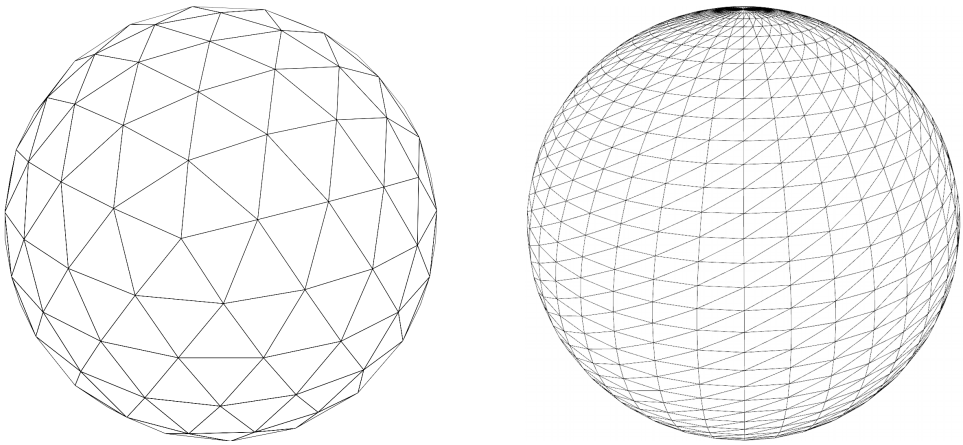


Figure 4.1: A geodesic sphere (left) and a regular sphere (right). The geodesic sphere was generated with 3DS Max 2015, the non-geodesic sphere was generated with UniconD3D and contains 50 horizontal rings and 50 vertical slices.

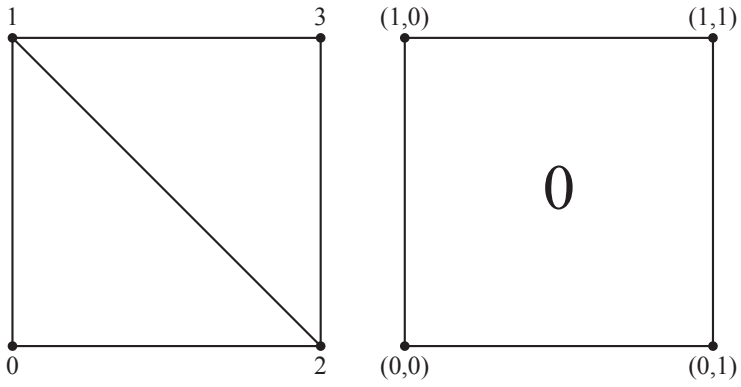
4.3.1 Convex Quadrilaterals

A convex quadrilateral, "quad" or "plane" is any shape that consists of two triangles that share two vertices, such that edge between two shared vertices would not intersect the edge between the non-shared vertices, if the latter edge existed. An example convex quadrilateral is shown in Figure 4.2a. The quads generated by UniconD3D are strictly convex, and triangular quads may be created by placing two vertices in the same triangle at the same coordinates. A quad is encoded with four vertices and six indices, of which four indices are unique. The vertices are trivially generated: a quad with legs of length l that is centered at the origin of a 3-D graph will have vertices V at the following coordinates:

$$V = \left\{ \left(-\frac{l}{2}, 0, -\frac{l}{2}\right), \left(-\frac{l}{2}, 0, \frac{l}{2}\right), \left(\frac{l}{2}, 0, -\frac{l}{2}\right), \left(\frac{l}{2}, 0, \frac{l}{2}\right) \right\}$$

A quad is formed from these vertices with the following set of indices I:

$$I = \{0, 1, 2, 2, 1, 3\}$$



(a) Quad indices (b) Quad UV coordinates

Figure 4.2: Placement of indices and texture coordinates for quads

This means the quad consists of two triangles described by the indexed vertices $\{0, 1, 2\}$ and $\{2, 1, 3\}$.

A quad has only one face, which is filled up entirely by a single texture, if one is assigned to the quad. Since UV coordinates range from $(0, 0)$ to $(1, 1)$ unless the texture is repeated or only a portion of the texture is shown, the texture coordinates T for the four vertices in V are:

$$T = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$$

The texture coordinates are illustrated in Figure 4.2b. Primitive faces are henceforth numbered $0...n$ to make them easier to identify in subsequent explanations of UV mapping algorithms.

4.3.2 Cubes

Because the number and positions of the vertices in a cube do not change based on user-set parameters, the positions of the vertices of a cube can be hardcoded. A cube consists of six planes with legs of length l and is centered at the origin of a 3-D graph. Its vertices V are trivially generated and are therefore omitted here.

The index sequence for the quad described in the previous subsection can be repeated for each face of the cube to generate all of the cube's indices. We do so by creating six such sets of indices and adding a different offset to the indices for each face. In doing so, we assume that the corresponding vertices will be contiguous in the vertex buffer, which is true in UniconD3D's implementation. The offset to the indices of each face are increments of four – the number of vertices per face – to obtain the set of indices I that describe a cube in Figure 4.3.

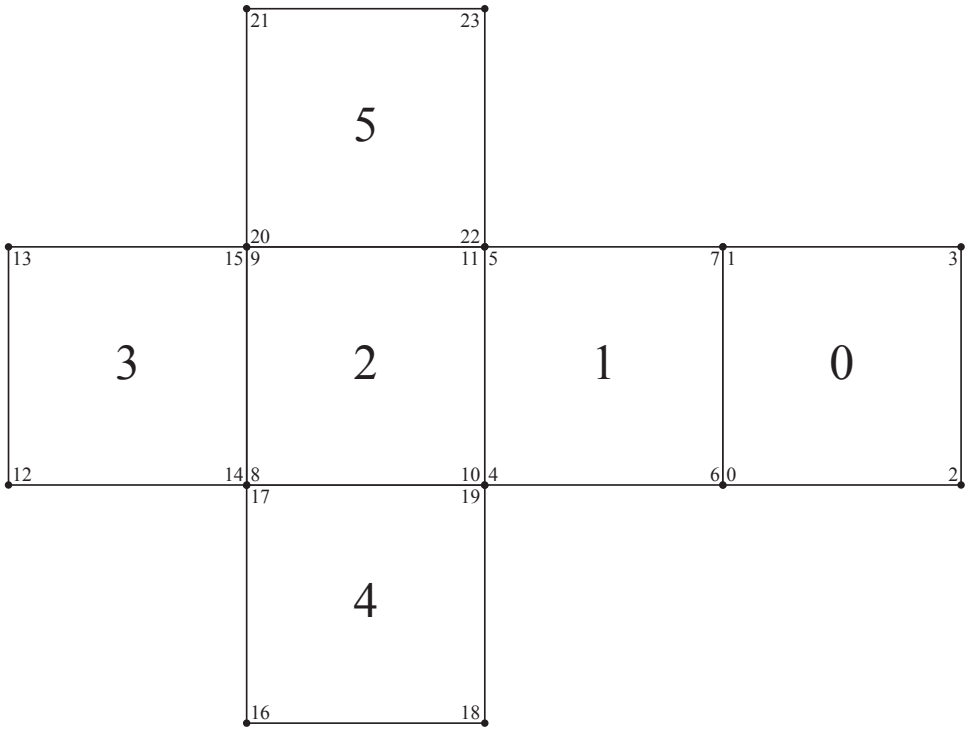


Figure 4.3: An exploded view of an indexed cube with numbered faces

This principle extends to models of arbitrary complexity: for every four vertices that comprise one face of the model, the corresponding indices that index these vertices are:

$$I = \{i, i+1, i+2, i+2, i+1, i+3\}$$

Here, i is the number of the face that is being indexed.

Texture coordinates for cubes are trivially generated: the same single texture is repeated once on each face, which yields the same UV coordinates as those for a quad, but repeated once for each cube face. The face numbers are included in Figure 4.3 for the sake of clarity.

4.3.3 Cylinders

The vertices, indices and texture coordinates of a cylinder are generated in steps: first the mid section is created, then the top cap and finally the bottom cap.

The vertices of the mid section consist of two horizontal "rings" of $n + 1$ vertices each, which are later connected with indices to form convex quadrilateral facets. Because of the way the cylinder is oriented when it is first created – that is, with both caps being horizontal – the Y coordinates of the vertices of the mid sections are $\frac{h}{2}$ for the top ring and $-\frac{h}{2}$ for the bottom ring, where h is the height of the cylinder. The top cap may have a different radius than the bottom cap to allow the user to create cone-shaped cylinders.

The X coordinate x_i of each vertex i in the top ring is calculated as:

$$x_i = \sin\left(\frac{i \cdot 2\pi}{n}\right) \cdot r_t$$

Here, n is the number of vertical slices the mid section consists of (i.e., how "smooth" the mid section is) and r_t is the radius of the top cap. The X coordinates of the bottom cap are generated identically, with r_t replaced by r_b, the radius of the bottom cap.

Similarly, the Z coordinate z_i of each vertex i in the top ring is calculated as:

$$z_i = \cos\left(\frac{i \cdot 2\pi}{n}\right) \cdot r_t$$

UniconD3D generates vertices for the top and bottom cap in alternating order. Figure 4.4 depicts how the quadrilateral facets should look after the cylinder's indices connect its vertices, regardless of the values for r_t and r_b.

The process of generating indices for facet i from the left end of the "strip" of convex quadrilaterals in Figure 4.4 follows the same approach as generating indices for a single convex quadrilateral, yielding the following index list for each facet i:

$$\{i, i+1, i+2, i+2, i+1, i+3\}$$

However, because the above formulas for the X and Z coordinates describe a circle in a counter-clockwise direction, and this left end of the quad strip is in fact the right end, the indices are instead specified in reverse order:

$$\{i+3, i+1, i+2, i+2, i+1, i\}$$

Note that we seem to draw two more vertices than necessary: the vertices on either end of the "strip" of quads will overlap. This is done because although their positions are the same, and although both ends could be connected with indices without requiring two additional vertices, the texture coordinates for the vertices on either end are different. Two extra vertices are needed because UniconD3D stores UV coordinates in the same data structure that describes a vertex.

The vertices that comprise the top cap are placed at the same coordinates as the vertices in the top ring of the middle section. The reason they exist is for the same reason two extra vertices were placed at one end of the mid section's quad strip: although their positions are identical to those of the vertices already in place at the cylinder's top, their normals are perpendicular to the normals of the vertices that comprise the mid section. Normals are explained briefly in Section 4.6. The top cap's indices connect the newly added vertices to one additional vertex in the middle of the top cap, i.e., at coordinates $\left(0, \frac{h}{2}, 0\right)$. The vertices of the bottom cap are created in the same way, at coordinates $\left(0, -\frac{h}{2}, 0\right)$.

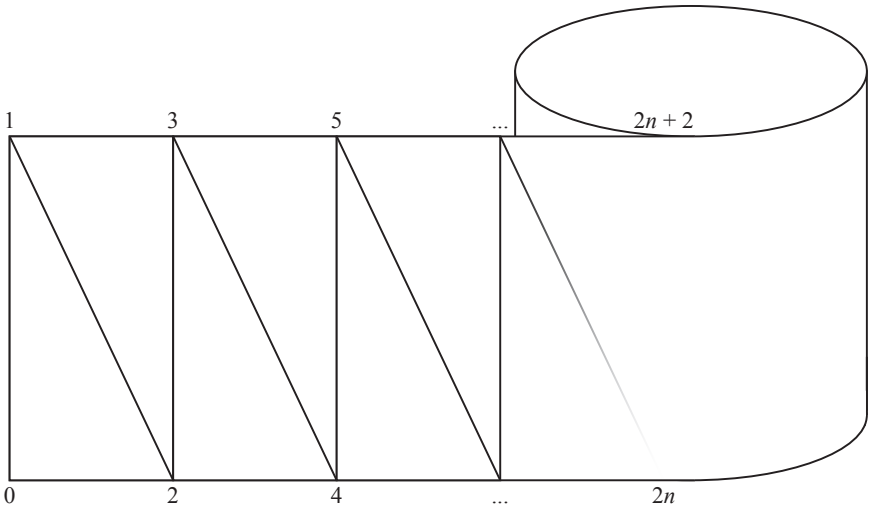


Figure 4.4: A cylinder with an exploded mid section consisting of indexed quads.

Indices connect the top cap's vertices one triangle at a time. Triangles are created by repeatedly connecting the most recently added vertex to the single most recently added vertex and the vertex in the middle of the top cap. This means that for each vertical slice i the cylinder consists of, the following indices are added:

$$\{(2n+2)+i+1, (2n+2)+i+2, 3n+3\}$$

Here, n is the number of vertical slices the cylinder consists of. An offset of $2n+2$ was added to each index because the mid section consisted of $2n+2$ vertices. The index of the vertex in the middle of the top cap is $3n+3$ because the top cap itself has $n+1$ vertices. The indices of the bottom cap are generated like those of the top cap, but in reverse order because their respective vertices are meant to be visible only when the bottom of the cylinder is visible to the camera. To the indices of the bottom cap we also add an additional offset of $n+1$. The indices of the top cap are illustrated in Figure 4.5;

Cylinders have three texturizable surfaces: the mid section, the top cap and the bottom cap. Texture coordinates for a cylinder's vertices are calculated separately for each of these surfaces. The vertices in the mid section have V coordinates 0 and 1 for the vertices in the bottom ring and top ring, respectively. The U coordinate for vertical pairs of two vertices along the mid section depend on how far along the mid section the vertices lie. Because a cylinder is split into n vertical slices, each U coordinate u_i in slice i should be $\frac{i}{n}$, but because the vertices are generated to describe rings in counterclockwise order, u_i is defined as $1 - \frac{i}{n}$.

The U coordinate u_i of each vertex i in the top cap excluding the center vertex is defined as:

$$u_i = \frac{1}{2} \cdot \left(\sin\left(\frac{i \cdot 2\pi}{n} \right) + 1 \right)$$

The V coordinate v_i of each vertex i in the top cap excluding the center vertex is defined as:

$$v_i = \frac{1}{2} \cdot \left(\cos\left(\frac{i \cdot 2\pi}{n} \right) + 1 \right)$$

Finally, the UV coordinates of the center vertices in the top and bottom caps is trivially $(0.5, 0.5)$.

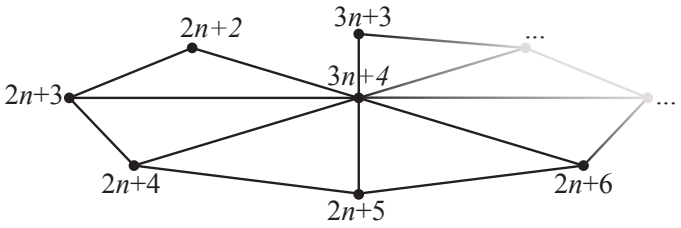


Figure 4.5: The indexed vertices of the top cap of a cylinder with eight vertical slices.

4.3.4 Spheres

The vertices of a sphere are generated in a series of horizontal rings of varying radii. The number of horizontal, parallel rings and vertical, nonparallel slices is user-specified. If the user specifies the sphere to have m horizontal rings and n vertical slices, $m + 1$ rings and $n + 1$ slices are generated. The extra ring and slice exist because although the first and last ring, as well as the first and last slice, will overlap with one another, the vertices in them will have different texture coordinates. Assume a sphere that has m horizontal rings, n vertical slices, and a radius r. The Y coordinate y_i of each horizontal ring i such that $0 \leq i \leq m$ is defined by the following sine wave:

$$y_i = r \cdot \sin\left(\frac{i \cdot \pi}{m} - \frac{\pi}{2}\right)$$

Then, on one such horizontal ring i, each vertex j has X coordinate x_j such that $0 \leq j \leq n$, which is defined as the following sine wave:

$$x_j = r \cdot \sin\left(\frac{j \cdot 2\pi}{m}\right) \cdot \cos\left(\frac{i \cdot \pi}{n} - \frac{\pi}{2}\right)$$

Finally, the Z coordinates z_k of vertex k such that $0 \leq k \leq n$ on horizontal ring i is defined as:

$$z_k = r \cdot \cos\left(\frac{z \cdot 2\pi}{m}\right) \cdot \cos\left(\frac{i \cdot \pi}{n} - \frac{\pi}{2}\right)$$

The indices I for the four vertices placed at the intersections of horizontal ring i and $i + 1$ and vertical slices j and $j + 1$ are generated as follows in sets of six:

$$\begin{aligned} I = \quad & \{(i \cdot (n+1) + j)\}, \\ & \{((i+1) \cdot (n+1) + j)\}, \\ & \{(i \cdot (n+1) + j + 1)\}, \\ & \{(i \cdot (n+1) + j + 1)\}, \\ & \{((i+1) \cdot (n+1) + j)\}, \\ & \{((i+1) \cdot (n+1) + j + 2)\} \end{aligned}$$

This pattern is identical to that for quad indices, but accounts for offsets of the current ring and slice for which the indices are generated. The indexed quads can in fact be combined to form a convex quadrilateral surface subdivided in $m+1$ by $n+1$ sub-surfaces, as depicted in Figure 4.6.

As with the cylinder, the sphere's vertices are generated in counterclockwise order, which means the pattern for indices listed above should be reversed to create the following pattern:

$$\begin{aligned} I = \quad & \{((i+1)\cdot(n+1)+j+2)\}, \\ & \{((i+1)\cdot(n+1)+j)\}, \\ & \{(i\cdot(n+1)+j+1)\}, \\ & \{(i\cdot(n+1)+j+1)\}, \\ & \{((i+1)\cdot(n+1)+j)\}, \\ & \{(i\cdot(n+1)+j)\} \end{aligned}$$

A sphere has only one texturizable surface, which spans the entire sphere. Texture coordinates for a sphere with m rings and n slices are trivially generated: the U coordinates are determined by which slice is drawn and the V coordinate is determined by which ring a vertex lies on. Intuitively, the U coordinate u_j of vertex j on any horizontal ring is $\frac{j}{n}$, while the V coordinate v_i of all vertices on ring i is defined as $\frac{i}{m}$. However, since vertices for rings and slices are generated in counterclockwise order, u_j is defined as $1-\frac{j}{n}$ and v_i is defined as $1-\frac{i}{m}$.

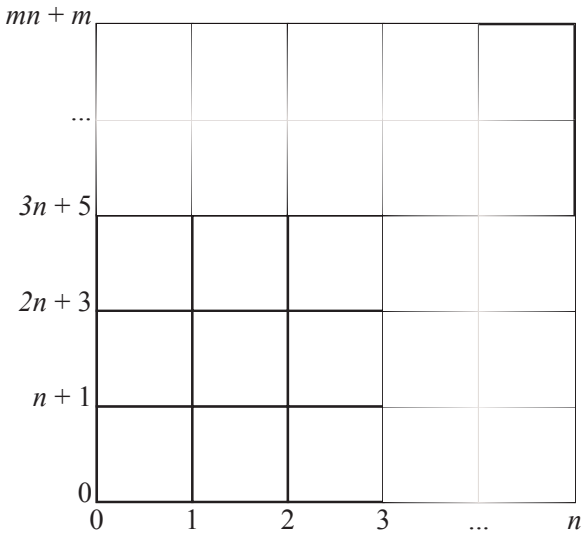


Figure 4.6: The indexed vertices of a sphere with n horizontal slices and m vertical slices.

4.3.5 Tori

As with a sphere, the vertices of a torus are generated as a series of horizontal rings with varying radii and varying heights. Assuming a torus that contains m horizontal rings and n vertical slices, as well as an inner radius r_1 and an outer radius r_2, the rings are generated in steps of $\frac{1}{m}$, which means the Y coordinate y_i of each of the $m+1$ rings i is defined as:

$$y_i = \cos\left(\frac{i \cdot 2\pi}{m}\right) \cdot (r_2 - r_1)$$

The above formula describes a vertical circle upon which the radii of horizontal rings is also based. The radius varies between $\left(r_1 + \frac{r_2 - r_1}{2}\right) \pm \left(\frac{r_2 - r_1}{2}\right)$, and the amount of variation is the product of a sine wave, which results in the following formula for the radius r_i of ring i:

$$\left(r_1 + \frac{r_2 - r_1}{2}\right) + \left(\left(\frac{r_2 - r_1}{2}\right) \cdot \sin\left(\frac{i \cdot 2\pi}{m}\right)\right)$$

Then, the X coordinate x_j for each of the $n+1$ vertices j on a given ring i is defined as:

$$x_j = r_i \cdot \sin\left(\frac{j \cdot 2\pi}{n}\right)$$

Finally, the Z coordinate z_j for each of the $n+1$ vertices j on a given ring i is defined as:

$$z_j = r_i \cdot \cos\left(\frac{j \cdot 2\pi}{n}\right)$$

The indices and texture coordinates for a torus are generated exactly like those for a sphere. As with the sphere, the surface of a torus is rectangular when "unwrapped", and triangles can be constructed in an identical fashion using the property that the vertices at the intersections of each pair of adjacent rings and each pair of adjacent slices delineate a convex quadrilateral. Because a torus has only one texturizable surface, exactly half of the assigned texture will be visible from the Euclidean center of the torus.

4.3.6 Disks

A disks is a flat object that exist in 3-D space. The user specifies its shape by entering an inner radius r_1 and an outer radius r_2, which means the "thickness" of the disk equals $r_2 - r_1$. The user also provides angles a_1 and a_2 to specify which portion of the disk is drawn: if $a_1 \neq a_2$, an incomplete disk is drawn with a gap of $\min(|\ a_1 - a_2\ |, |\ a_2 - a_1\ |)$ degrees. A disk starts at angle a_1 and ends at angle a_2 such that $0 \leq a_1 \leq 360$ and $0 \leq a_2 \leq 360$. a_2 may be greater than a_1. Finally, the user also specifies the amount of tessellation of the disk by entering a number of slices n.

The vertices are generated in two horizontal rings of $n+1$ vertices each. The Y coordinate for all vertices is 0.

The X coordinate x_i for vertex i on the inner ring is defined as:

$$x_i = r_1 \cdot \sin\left(a_1 + \frac{i \cdot a_3}{n}\right)$$

Here, $a_3 =\mid a_1 - a_2 \mid$. The corresponding X coordinate x_j on the outer ring is defined identically, but with r_1 replaced with r_2.

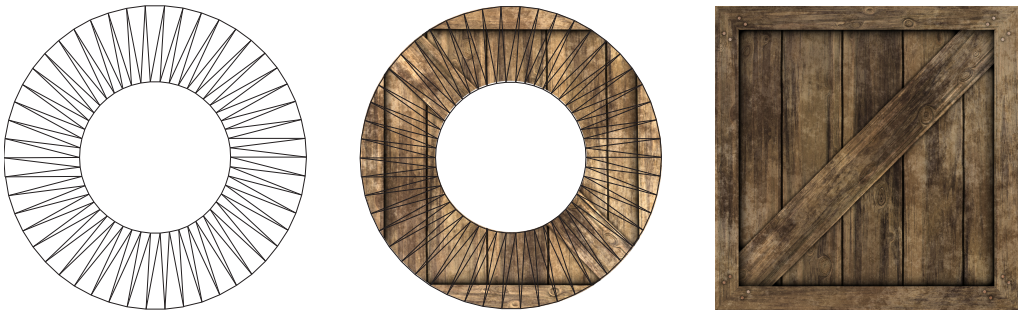
The Z coordinate z_i for vertex i on the inner ring is defined as:

$$z_i = r_1 \cdot \cos\left(a_1 + \frac{i \cdot a_3}{n}\right)$$

The Z coordinate z_j for vertex j on the outer ring is defined identically, with r_1 replaced with r_2.

The indices for a disk are generated exactly like those for the mid section of a cylinder, since they both describe the same topology. A disk can be seen as the mid section of a cylinder with a larger bottom radius than the top radius, and the top cap at the same Y coordinate as the bottom cap.

Disks have a single texturizable face. The texture coordinates for a disk are generated such that the disk would appear to be positioned at the center of the image while spanning the full width and height of the image. An example is shown in Figure 4.7.



(a) Indices of an example disk.

(b) Indices of the example disk superimposed on a textured disk.

(c) The used example texture.

Figure 4.7: An example textured and untextured disk, which contains 50 slices.

The U coordinate u_i for vertex i on the inner disk is defined as:

$$u_i = \frac{x_i \cdot r_1 + \frac{r_2}{2}}{2 \cdot r_2 + 0.25}$$

Similarly, the U coordinate u_i for vertex i on the outer disk is defined as:

$$u_j = \frac{x_j \cdot r_2 + \frac{r_2}{2}}{2 \cdot r_2 + 0.25}$$

The V coordinate v_i for vertex i on the inner disk is defined as:

$$v_i = \frac{z_i \cdot r_1 + \frac{r_2}{2}}{2 \cdot r_2 + 0.25}$$

The V coordinate v_i for vertex i on the outer disk is defined as:

$$v_j = \frac{z_j \cdot r_2 + \frac{r_2}{2}}{2 \cdot r_2 + 0.25}$$

Backface culling is turned off when disks are rendered, so that disks are visible from both sides.

4.3.7 Lines

Although lines lack depth, they exist in 3-D space. No vertices need be generated for lines, because lines are entered by the user as a set of vertex coordinates.

The indices for lines are trivially generated by connecting each vertex i to one subsequent vertex $i+1$ for each vertex i such that $0 \leq i < n-1$, where n is the number of user-entered vertices.

Because lines in Direct3D 11 are one pixel wide[1], lines can hold no textures and no UV coordinates need be generated. One can however specify a color for each vertex, which is used by the vertex shader to automatically assign a color to each pixel on a line piece between two such colored vertices. The line segment between two colored vertices v_1 and v_2 is given a color gradient with colors ranging from color(v_1) to color(v_2).

Backface culling is turned off when lines are rendered.

4.3.8 Points

As with lines, points have no visible depth to them and are entered by the user as groups of vertices. Since they are one by one pixel in size when rendered, no textures can be applied to them, but one can assign a color to each vertex that holds a point coordinate, which is used by the pixel shader to color the pixel for that point.

The indices for points are not used to connect points and are only needed for the sake of consistency. They are trivially generated by assigning index i to each vertex i such that $0 \leq i < n$, where n is the number of user-entered vertices.

Backface culling is turned off when points are rendered.

4.3.9 Polygons

Polygons are unfilled, untextured, planes that are bounded by multiple angles. They are closely related to filled polygons, described in the next subsection, but polygons exist as line sections whereas filled polygons exist as filled, possibly textured, surfaces. All primitives in this section except lines and points are technically polygons, but we don't consider them polygons in UniconD3D because we ascribe special attributes to filled and unfilled polygons.

The geometry of polygons and filled polygons can be created with two methods, one for convex polygons and one for concave polygons. Convex polygons are shapes with angles of at most 180 degrees, while concave polygons have angles greater than 180 degrees. Convex polygons are easily deconstructed into triangles $\{i+1, i+2, 0\}$ such that $0 \leq i < n-2$, where n is the number of vertices in the polygon. This means the triangles are laid out in a *triangle fan* topology: after the first three vertices

[1] Assuming no sub-pixel anti-aliasing is used, in which case lines are approximately one pixel wide.

are connected with indices to form a triangle, each subsequent vertex is connected with indices to the most recently added vertex and the first vertex.

Up until version 10.1, Direct3D supported such triangle fan layouts directly, thereby foregoing the need for an additional step to generate indices, but as of v11.0 the Input Assembly stage of the rendering pipeline can no longer be configured to use a triangle fan topology. UniconD3D therefore indexes the polygon as a set of line segments, connecting each vertex to the last and connecting the last vertex to the first. The indices are therefore generated as follows: each vertex i is connected to vertex $(i+1)\%n$ where n is the number of vertices the polygon consists of.

Polygons cannot be textured, but the vertices a polygon consists of can be given colors. The line segment between two colored vertices v_1 and v_2 is given a color gradient with colors ranging from color(v_1) to color(v_2).

Backface culling is turned off when polygons are rendered.

4.3.10 Filled Polygons

Filled polygons are similar to the polygons described in the previous subsection, but instead of indexing a given set of vertices as though they describe a series of connected line segments, indices are generated to create triangles that simulate a triangle fan. As alluded to in the previous subsection, this means for each triangle i that the polygon can be decomposed into, the following indices I are generated to create it:

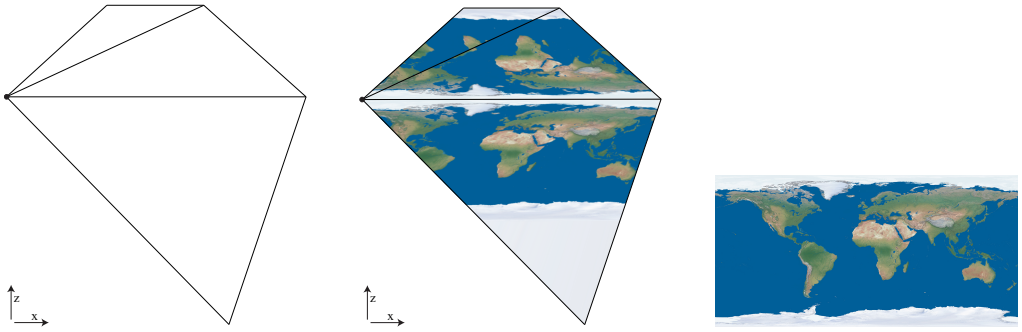
$$I = \{i+1, i+2, 0\}$$

This method only works for convex polygons. Concave polygons cannot be drawn in UniconD3D, and they are not supported by Unicon's high-level, graphics API-independent, 3-D graphics instructions. Algorithms exist to decompose any concave polygon into a minimal set of convex polygons [39], but implementing one is beyond the scope of this project.

The method used to generate texture coordinates in Unicon's existing OpenGL 1.2-based 3-D graphics facilities is minimally described in the original OpenGL specifications, Unicon Technical Reports refer exclusively to the official OpenGL specifications when explaining how texture coordinates are chosen. Specifically, the method used by Unicon's OpenGL-based facilities generates UV coordinates automatically using one of the three listed OpenGL functions, which is not explained further in the OpenGL 1.2 specifications. Another method is mentioned, which places a texture on all sides of the filled polygon by projecting different parts of the texture from within the polygon. By disabling backface culling, the same texture will appear "wrapped around" the entirety of the polygon. However, unless one uses (a) a combination of at most six textures to span the entire 360 degrees of vision along all axes to cover all sides of the polygon with a texture, or (b) a composite image file format such as *dds*, which is unsupported by Unicon, the texture will look unnaturally stretched and the end result will be unpredictable. Specifically, faces of the polygon that stand at a large angle (approaching 180 degrees), as seen from

the polygon's Euclidean center, will invariably receive a small portion of the projected texture and look washed out.

In the absence of a preferred solution, we chose a rather arbitrary algorithm that produces consistent and easy-to-use results. The first vertex in the filled polygon is chosen to have UV coordinates $(0, 0)$, and every vertex that follows is assigned texture coordinates that correspond linearly with where the vertex should be placed on the texture if the first vertex were in the lower left corner of the texture image. The sections of the polygon that are located "below" the first vertex are given negative texture coordinates, which means the original texture is flipped vertically and stretched to fit the section of the polygon below the first vertex. An example of such texture placement is shown in Figure 4.8, the polygon in this figure contains vertices $\{(0, 0, 0), (1, 0, 1), (2, 0, 1), (3, 0, 0), (2, 1, -2)\}$.



(a) Indices for example pentagonal filled polygon. The dot marks vertex position $(0, 0, 0)$.

(b) Texture-mapped version of the polygon with the indices superimposed.

(c) The used example texture.

Figure 4.8: A textured and untextured example filled polygon.

Because any primitive may be rotated at will, there are no clear definitions for the directions "above" and "below" in determining texture coordinates. We therefore state that each vertex has a distance relative to the first vertex in the combined Y and Z directions and the combined X and Z directions. The vertex with the greatest Euclidean distance in the YZ direction delineates the "top" of the polygon and is given V coordinate 1.0. The vertex with the greatest Euclidean distance in the XZ direction indicates the "right" end of the polygon and is given U coordinate 1.0. All other vertices are given UV coordinates that fall between these bounds.

Backface culling is turned off when filled polygons are rendered.

4.4 Rendering Text

Until Microsoft published version 11.0 of the DirectX API, Direct3D contained high-level functions to access a built-in text renderer that could use any font included with Windows. This text renderer, called DirectWrite, was able to access Direct2D and Direct3D canvases directly. However, Microsoft removed support for the DirectWrite-based font renderer in DirectX 11 in order to give game developers greater control over how text is rendered, which created a situation where DirectWrite and Direct2D could no longer write on a Direct3D canvas directly. Solutions proposed by Microsoft and the community at the MSDN forums were to write your own Direct3D 11-compatible text renderer, or to build additional logic into the graphics engine to share a Direct3D 11 canvas between Direct3D and Direct2D by locking it, re-casting it to a Direct3D 10 canvas, drawing to it, casting it back to a Direct3D 11 canvas and unlocking it. The former approach is nontrivial and time-intensive for anyone not familiar with font rendering. The latter approach was doable, but presented additional challenges and incurred a performance hit. Sharing a canvas in such a manner also involved using Direct3D 10-specific data types, which are not supported by the Windows SDK that is present in Windows 8 and 8.1. Concretely, sharing the Direct3D 11 canvas with Direct3D 10 and D2D would make it impossible to compile the graphics code on future operating systems unless the DirectX SDK from June 2010 is installed in addition to the Windows SDK. The availability of the DirectX SDK and its compatibility with future versions of Visual Studio and Windows is questionable.

A third solution presented itself in the form of a third-party library called *FW1FontWrapper*, which is offered on CodePlex, an open source software distribution site for add-ons to Microsoft products [40]. This library is used in UniconD3D because it works directly with Direct3D 11 render targets and provides an easy way to render text. A copyright notice regarding distribution of the FW1FontWrapper library is included in Appendix C.

4.4.1 Rendering 2-D Text

The FW1FontWrapper library directly supports writing text to 2-D pixel coordinates. These coordinates specify the upper left corner of the text's first character relative to the render target the text is drawn to; UniconD3D uses the back buffer as the render target. The programmer must specify the font, font size and font color of the text, and may set any number of flags to affect text wrapping, clipping, indentation and alignment options. 2-D text is drawn over all other window contents and is particularly useful for displaying rendering statistics and debugging information. 2-D text is available to the user as a special type of geometric primitive, and is added to area files in a similar manner.

4.4.2 Rendering 3-D Text

Because the FW1FontWrapper library does not support placing text at 3-D coordinates in an environment, a different approach was taken to create the illusion text is placed at 3-D coordinates. The chosen 3-D coordinates of a given string of text, which are given in world space, are transformed to projection space to obtain the on-screen 2-D coordinates the text should be drawn at. If these 2-D coordinates are outside the edges of the window, that is, outside the user's field of view, the text is not rendered at all. If the 2-D coordinate are inside the output window, the 2-D coordinates are "picked" to determine how far away from the camera an object is located in the same direction as the text. If the distance between the camera and the nearest object in this direction is smaller than the Euclidean distance between the camera and the text's 3-D coordinates, the text should be hidden from sight because another object is located in front of it, and the text is not rendered. If this is not the case, the text is rendered.

This implementation choice means a picking operation is performed for each 3-D text which is in the environment for each rendered frame. This is unfortunate, because the picking operation is expensive: in order to see if a 2-D pixel coordinate in projection space is occupied by a 3-D model in the underlying view space, a vector is draw from the camera in the direction of a point in view space. This vector may also be drawn in world space. For each primitive P in the environment, this vector is transformed to P's local space and used to calculate if the vector intersects with any of the triangles that comprise P. Normally the picking vector is transformed to local space before the intersection is calculated, but because Direct3D's built-in intersection function[2] fails on intersection calculations in local space, the primitives' vertices and the picking vector are transformed and intersected in world space instead. This is more expensive than transforming the picking vector to local space and leaving the primitives' vertices unchanged. (Please recall that transformations of large number of vertices is only feasible when done in parallel on a GPU.) This approach to deciding whether 3-D text strings are drawn means a text string is not rendered if any of its first characters are hidden behind another object.

3-D text strings are entered by the user as geometric primitives in area files, much like 2-D text strings. However, the user may also enter a minimum and maximum size of the 3-D text as additional parameters in the primitive description, which lets the text scale with the distance between the text string and the camera.

[2]This is TriangleTests::Intersects(), see documentation here:
`http://msdn.microsoft.com/en-us/library/windows/desktop/microsoft.directx_sdk.triangletests.intersects(v=vs.85).aspx`

4.5 Textures

Textures are images that are used to fill the space between vertices in order to make objects look solid and increase the realism of a scene. This is done by opening an image file on the hard drive or in memory, loading its contents into a buffer, creating a *shader resource view* from the buffer, and transferring the shader resource view to VRAM, where it is made available to the pixel shader. The pixel shader then samples the resource view to determine which color a surface has at any given screen coordinate. By repeating this process for each pixel in the output window, the pixel shader will sample different textures for the different models that appear on the screen.

DirectX features limited built-in support for creating shader resource views from image files. Common files formats such as JPEG, GIF and PNG are supported, but wrappers that contain multiple images in one file such as DDS are not. In order to support more advanced file formats, and because UniconD3D uses DDS files for environment maps, we opted to use a third-party library called the DirectX Tool Kit (DirectXTK/DXTK) [41] to create shader resource views from all files. Environment map creation is explained below.

Textures for a primitive are included in the primitive's description in the area file the primitive is in. In rendering a given primitive to the screen, index by index, we are faced with the choice of when to decide to use a given texture for subsequent indices, until another texture is specified. One also needs to decide how to render faces of primitives for which no texture is specified.

For the first problem, the Primitive object that holds the primitive's user-set parameters also holds a list of indices that tell the renderer when the corresponding textures in a list of texture names are to be used. Before the indices of a primitive are rendered, the number of indices to be rendered with a given texture from the primitive's list of texture names is calculated and rendered after the pixel shader is configured to use this texture as its current resource to sample.

As for the second problem, this approach implies that primitive indices for which no texture is set shall not be rendered. For instance, when the first texture of a primitive must be used from the primitive's seventh index onward, the first six indices will not be rendered. Additionally, this approach implies that once a texture is defined to be used from a certain index onward, the same texture is applied repeatedly for subsequent indices (i.e., model faces) until all indices are drawn.

Although commercial video games seldom use untextured surfaces because these break the suspension of disbelief, code was added to UniconD3D's handling of textures to ensure no primitive face is rendered invisible. In the standalone version of the renderer, color information was added to each vertex to be used when texture information is not available. In the integrated version of the renderer, Unicon may choose to draw primitives in a "textured mode" or an "untextured mode", and either a texture or surface color would be chosen depending on which mode the renderer was set to prior to drawing the indices of a primitive. This is explained further in Chapter 5.

4.5.1 Environment Maps

Each area may contain one special textured primitive called an *environment map*, which is a collection of images that can be assigned as textures to a geometric shape that encloses all of a scene's geometry. UniconD3D assigns one environment map to a cube primitive in each area, and renders this cube such that the camera is located at its center. The cube moves along with the camera, which gives the impression that the dimensions of the cube are infinitely large. So long as the textures that cover each cube face connect seamlessly to one another, this illusion is never broken. Unlike other primitives, when the environment map is drawn, the depth buffer is not modified, which means the environment map cannot overlap with another shape in the scene. Therefore, the dimensions of the environment map are irrelevant. Unicon does not use environment maps, which means UniconD3D is instructed to create an empty environment map by default, which does not show up. The functionality needed to draw environment maps is left in UniconD3D, but a high-level function for creating environment maps that can later be called by Unicon is omitted. When environment maps are not used, the window space not occupied by models holds the color with which the window was last erased.

4.5.2 Rendering to Textures

UniconD3D provides support for textures that can be modified at runtime. Unicon allows the programmer to render 2-D primitives to an auxiliary window, to store the contents of this window as an in-memory texture, and to update the texture's appearance in a rendered scene on-the-fly in the OpenGL-based graphics facilities. UniconD3D offers to reload textures stored on the disk on-the-fly whenever they are changed. This functionality can later be used by Unicon to set the contents of a 2-D or 3-D output window as a texture, and overwrite an existing image file if one with the chosen name already exists. UniconD3D will then pick up on the fact an image file's write time has changed, and update the corresponding shader resource.

UniconD3D uses built-in Windows kernel functions to access an image file's write time, i.e., the last time the image file was modified. The resolution of write times depends on the file system that is used, and may range from several nanoseconds to several seconds. A few write time resolutions are included in Table 4.1. Because the proposed Direct3D-based renderer operates only on Windows platforms, it is assumed the user will use the NTFS file system, which means the write time resolution allows a texture to be updated by the renderer with each frame redraw operation at a refresh rate of 60 frames per second, provided the processor is fast enough to do this.

UniconD3D goes through its list of shader resources at every frame draw and verifies the corresponding file's write time against the file's last known write time. If the file has been modified since the last two frames were rendered, a new shader resource is created from it that overwrites the existing shader resource with the same name. How quickly one wants to re-load image files into textures is expressed

as a number of frames, UniconD3D provides a function that changes this refresh speed. This refresh speed may need to be tuned depending on the number of textures used in the currently rendered area, because the time complexity of the update operation increases linearly with the number of textures.

Table 4.1: Write access resolutions for different file systems

File system	Write time resolution
FAT, VFAT	2 s.
NTFS	100 ns.
UNIX/Linux file systems	1 s.

4.6 Lighting

UniconD3D uses a local illumination model as defined by Möller et al. [42]. This means that reflected light from one 3-D model in a scene does not affect the amount of reflected light from nearby models. For example, if a model A stands between a light source and another model B, B is lit as though A never existed. The lighting system is not global because research on reducing the computational expense of realtime global illumination models has yet to yield a model that is sufficiently computationally inexpensive for use in real-time 3-D graphics engines. UniconD3D also uses a direct lighting model rather than a deferred lighting model, which means the number of lights in any given scene has to be small for reasons of computational complexity. Deferred shading techniques make better use of the different stages of the rendering pipeline to perform different parts of shading calculations simultaneously, much akin to how multithreading on a CPU may be used to spread a calculation out over multiple clock cycles to allow the CPU to perform multiple calculations in parallel. In accordance with the lighting model in the OpenGL-based Unicon graphics, UniconD3D allows up to eight light sources in one scene.

4.6.1 Different Light Types

When a light ray strikes a model at an angle, the model's reflectiveness is used to calculate the intensity of reflected light of each color in the direction of the camera. The amount of reflected red, green and blue light is calculated separately to give red, green and blue reflections. Such reflectiveness is encoded in what are called *materials*. This means UniconD3D has to provide an abstraction of different types of light sources and different types of reflective properties to reflect light from each light source type. Each type of light source produces a different visual effect and has a different computational complexity. The

types of light below are combined to create abstractions of different types of light sources we encounter in our everyday lives, such as flashlights and lightbulbs.

1. Ambient light: the light that brightens the entire scene with one color. The objects' materials define how much of the ambient light is absorbed. Ambient light represents a combination of numerous weak and indirect light sources (such as the moon) that light objects weakly, even when no light source appears to be present.

2. Diffuse light: the light that is reflected off an object into the camera. The camera always sees this diffuse light. Diffuse light represents scattered light that bounces off a matte (uneven) surface. Until deferred lighting models were introduced, the number of light sources on screen with diffuse light was usually limited to three in commercial games. More is possible but prohibitively expensive.

3. Specular light: the light that is reflected by a very smooth surface. This type of light is reflected at a specific angle that generates a cone of reflectance through which the reflection may be seen. The camera will not see the cone of reflectance unless the camera intersects it.

4. Emissive light: the glow around a light source. Emissive lights are not used in UniconD3D.

The different types of light sources that can be created from the above light types are:

- Ambient light: a light color defined as three floating-point values – one for red, green and blue – that affect the light reflected by all materials in the environment. Ambient light is not reflected in a specific angle and is therefore visible to the camera from all viewing angles.

- Parallel ("directional") light: light that strikes all objects from a certain direction, stored in a direction vector. It behaves like the sun in that directional light illuminates all objects at the same angle. Like ambient light, parallel light is not reflected in a specific angle and is therefore visible to the camera from all viewing angles.

- Point light: a light source that radiates spherically in all directions. Point lights use diffuse and specular light and therefore attenuate. Point lights also have a range parameter that specifies at which distance the spot light can no longer be seen (regardless of whether attenuation would have rendered the light invisible). Please see the explanation of light attenuation farther into this section. Like ambient light, point light is not reflected in a specific angle and is therefore visible to the camera from all viewing angles.

- Spot light: a light source that casts light rays in a cone-shaped area (called the "cone of reflectance") with an angle φ. The intensity of light in the cone decreases from the center toward the edge. The intensity of the light at angle φ in the cone is calculated by $\max(\cos\varphi, 0) \cdot s$ where

the "smoothness factor" s is used to change how smooth the transition is from the center to the edge of the cone. Indirectly, s also changes the size of the cone. Spot lights attenuate like point lights, but calculating the light color adds an extra factor to multiply with. See the section below. Unlike the other light source types, spot light is only visible to the camera if it is located in the spot light's cone of reflectance.

4.6.2 Light Attenuation

The following discussion of the attenuation of different light source types was adapted from a discussion by F. Luna [43] and is common to many video games from before deferred shading became popular, i.e., around 2005.

Directional lights do not attenuate. Point lights and spot lights use a combination of diffuse light and specular light and therefore attenuate as a function of the distance between the light source and the lit object. Theoretically, diffuse and specular lights are attenuated using the inverse square law but more commonly a few attenuation factors are defined that divide the specular and diffuse lights. The attenuation of point light color PL of a point light is defined according to the following formula, as given by Luna:

$$PL = A + \frac{k_d D + k_s S}{a_0 + a_1 d + a_2 d^2}$$

Here, A is the ambient light factor, k_d is the diffuse light intensity, D is the diffuse light color, k_s is the specular light intensity, S is the specular light color. The coefficients a_0, a_1 and a_2 are attenuation factors that combine a constant, linear and quadratic amount of attenuation to weaken the diffuse and specular light. Note that each light intensity factor and each light color (ambient, diffuse, specular) exists as a vector of three floating point values: one for red, green and blue, so the formula for PL above is repeated thrice: once to calculate each color channel.

The attenuation of spot lights SL behaves according to the following formula, also given by Luna:

$$SL = k_{spot} \left(A + \frac{k_d D + k_s S}{a_0 + a_1 d + a_2 d^2} \right)$$

Spot lights behave almost like point lights, except their intensity depends on the additional factor k_{spot}, which changes depending on whether the camera is in the cone of reflectance of the spot light. The falloff of the light's strength is a function of the reflectance cone's angle φ, k_{spot} is defined as:

$$k_{spot} = \max(\cos \varphi, 0)^s$$

Here, exponent S (a spot light color R, G or B) can be changed to manipulate the size of the cone of reflectance.

4.6.3 Implementation

Lighting calculations are one of the two things UniconD3D does entirely in the pixel shader, the other being texture sampling. HLSL shaders and program code in C++ work together to compute the above light color L by following the steps below:

- Structures for each of the different light types are implemented twice: once in the program code as C structs, once in the HLSL file as HLSL structs. Each corresponding pair of structs has its members defined in the same order, including certain additional fields that pad members whose size is not 32 bits. This is done because HLSL constant buffer member variables are processed in groups of four bytes.

- A pixel shader constant buffer is defined in the HLSL file that contains an array of eight point light structures. Another field in this constant buffer specifies how many of these point light structures are actually being used.

- Another pixel shader constant buffer contains the material properties for the primitive shape that is currently being drawn.

- After both these pixel shader constant buffers have been updated, the pixel shader is run, which only performs lighting calculations. The pixel shader starts by sampling the texture of the primitive that is currently being drawn, and multiplies the sampled color from this texture for a given screen pixel by the multiplied results of up to eight point light color calculations. The result is a color with RGB values.

- To this color we add an alpha component that represents the pixel's transparency. Section 4.7 explains how UniconD3D handles transparency. Eventually, the pixel shader returns this RGBA value.

Again, please note that this implementation of point light calculations in the pixel shader was largely adapted from sample code by Luna, it was merely adapted to process larger numbers of point lights, spot lights, combinations of materials and textures, and our choice of transparency calculations.

The shader constant buffers, especially those used by the pixel shader, are not updated equally frequently: the constant buffer with material properties is updated once for each primitive in the scene, while the constant buffer with point light structures is updated only when a new area is rendered for the first time. A separate constant buffer that holds the camera's position is updated for each rendered frame. Using multiple constant buffers depending on the access frequency of their fields is a common practice in shader programming[3].

[3]See this page for tips on choosing constant buffer layouts:

http://msdn.microsoft.com/en-us/library/windows/desktop/ee416643%28v=vs.85%29.aspx

4.7 Transparency

UniconD3D provides two methods to render (portions of) primitives as transparent surfaces, both of which are implemented largely in shaders.

On the one hand, one could render a transparent model twice: first the sides that are invisible to the camera are rendered by turning backface culling off, and then the sides that are visible to the camera by turning backface culling back on. If, in the pixel shader, the alpha component of each pixel in this model is now set to a value less than 1.0 (but greater than 0.0), the Direct3D device context can be set to combine the output from the pixel shader with pre-existing color information in the back buffer through texture blending. For any given pixel on the screen, the colors of at most eight underlying textures are "blended" together by considering the colors of the pixels' RGBA values and performing an operation on these colors, which uses the alpha values of all pixels to determine how much each pixel color contributes to the final color that is shown to the user. Examples of such blending operations are adding to the final color (which is the most common operation) or subtracting from it. The disadvantage to this technique is that one has to specify which portion of a model has to be rendered transparently, which is most easily done by specifying a transparency percentage for each rendered model. This requires the programmer to update the pixel shader to use the correct alpha value at the right time by updating this value in a constant buffer accessible to the pixel shader. The pixel shader would then use this alpha value for all rendered pixels until it is given a new alpha value to use for subsequent pixels. An implementation of this method is explained in Section 4.7.1.

On the other hand, transparency can be applied more finely by combining texture blending from the above approach with the use of transparent textures. In such textures, alpha channel information is added to each pixel in the original image file from which the texture is created, so that the programmer does not need to manually provide alpha values through a shader constant buffer.

4.7.1 Rendering Transparent Models Twice

A transparent primitive needs to be rendered in portions: a primitive consists of faces that are visible to the camera and faces that are invisible to the camera. Unless the rasterizer stage has been configured otherwise, only those faces which consist of triangles who have their vertices specified clockwise through their indices are rendered. This means that the faces of a model that face away from the camera will always have their vertices defined counterclockwise. By extension, the faces that face away from the camera may be revealed by removing the faces that face the camera.

Direct3D allows one to specify different Rasterizer Stage configurations ("rasterizer states") to do this. The *culling mode*, which is defined in each rasterizer state, and which persists until it is specified otherwise, determines which vertices are rendered to the screen: those facing the camera or those facing away, or both or none. In order to recognize per Primitive object which primitives are transparent,

transparency was added as a user-specified parameter for primitive descriptions in area files. When a primitive is recognized as transparent by the graphics code, it will be rendered twice: once counter-clockwise, then clockwise, in that order. This means that when the transparent object is rendered, the Rasterizer stage is first configured with a rasterizer state that enables counterclockwise culling, then the object is rendered for the first time, subsequently the Rasterizer Stage is configured with a rasterizer state that enables clockwise culling, and finally the object is rendered a second time. This process repeats itself for all transparent objects.

In order to be able to set transparency with a finer granularity than on a per-primitive basis, transparency information was added to UniconD3D on a per-index basis. Each primitive contains a list of indices and corresponding transparency factors, so that primitives are rendered in portions that each have a different transparency factor. However, because textures operate on the same principle – i.e., primitives were already rendered in portions that each used one texture – primitives are now rendered in smaller partitions, where a change in texture or a change in transparency can both trigger an update of the pixel shader's per-model constant buffer.

Now the only remaining issue is determining the order in which multiple transparent objects are rendered that lie behind one another as seen by the camera. In other words, an infinitely long vector from the camera's position to any point on the screen may intersect with multiple transparent objects. If these are not rendered in descending order of distance to the camera, some transparent objects may incorrectly hide other transparent objects from sight because the blending operation is performed in the wrong order. The following steps were taken to resolve this problem:

1. The set of primitives in the current area to be rendered is divided into two sets: one set of opaque primitives and one set of primitives with any amount of transparency. The set of opaque primitives is rendered first with the default rasterizer state, which hides primitive faces facing away from the camera.

2. The set of primitives with transparency is sorted in descending order based on the primitives' distances to the camera. This happens whenever a frame is rendered because the camera's position is updated with each rendered frame, thereby changing the distances between the camera and the transparent primitives. Ordering the transparent primitives in this manner is not expected to detract from UniconD3D's performance much, since there are likely not very many transparent primitives to begin with. The most efficient sorting algorithms, heap sort, merge sort and quick-sort all have average time complexities of $O(n \log(n))$, but only heap sort and quick-sort occupy a constant amount of space. Although of these two only quick-sort is stable (i.e., it preserves the order of keys with equal values), since this doesn't matter, we looked instead at the expected and worst-case running times. Quick-sort has an unfavorable worst-case running time of $O(n^2)$ but it

is usually faster than heap sort. A quick survey on preferred sorting algorithms in this case favors quick-sort over heap-sort because it tends to be faster than heap sort on average, so quick-sort was chosen to sort the transparent primitives.

3. Because dividing the list of all of an area's primitives into lists of opaque and transparent primitives happens in linear time and these lists never change between frame renders of the same area, these lists are instead created and cached when an area is rendered for the first time, or when the renderer prepares to render a new area for the first time.

The number of transparent primitives could become large enough that sorting them with a single-threaded implementation of quick-sort becomes prohibitively expensive, since quick-sort still executes in logarithmic time on average. A simple benchmark was run to see the computational impact on frame times of sorting transparent primitives. This was done with areas that contained increasing numbers of primitives, the results are included as Figure 4.9. The increase in frame times in environments with sorted primitives compared to environments with unsorted primitives is negligible when the number of rendered primitives stays under 25,000. Because the complexity of a typical scene in CVE contains the equivalent of around 12,500 cubes, we conclude that single-threaded quick-sort incurs too small a performance hit to justify further steps. Each value in Figure 4.9 was obtained by recording the average frame draw times of 30 frames in a scene without primitive sorting and the average for 30 frames with primitive sorting. The number of cubes was increased in each subsequent scene by 1000 primitives, in a range of 0-25,000 primitives. The benchmark code that produced these results is available on `http://unicon.org`.

An interesting strategy to minimize the impact of ordering transparent primitives could be to categorize all transparent primitives and place each primitive in one of a small number of lists. Each list would then represent a range of distances between the objects in it and the camera. The problem then becomes determining the granularity of categorizing different ranges, i.e., determining the number of lists. A good strategy could be to do with these ranges what others have done with texture sizes: a high level of detail in a texture matters less as the texture is farther removed from the camera. So there would be many lists of distances close to the camera, and only a few for distances farther away from the camera. Alternatively, one could rewrite UniconD3D's quick-sort algorithm to use multiple threads.

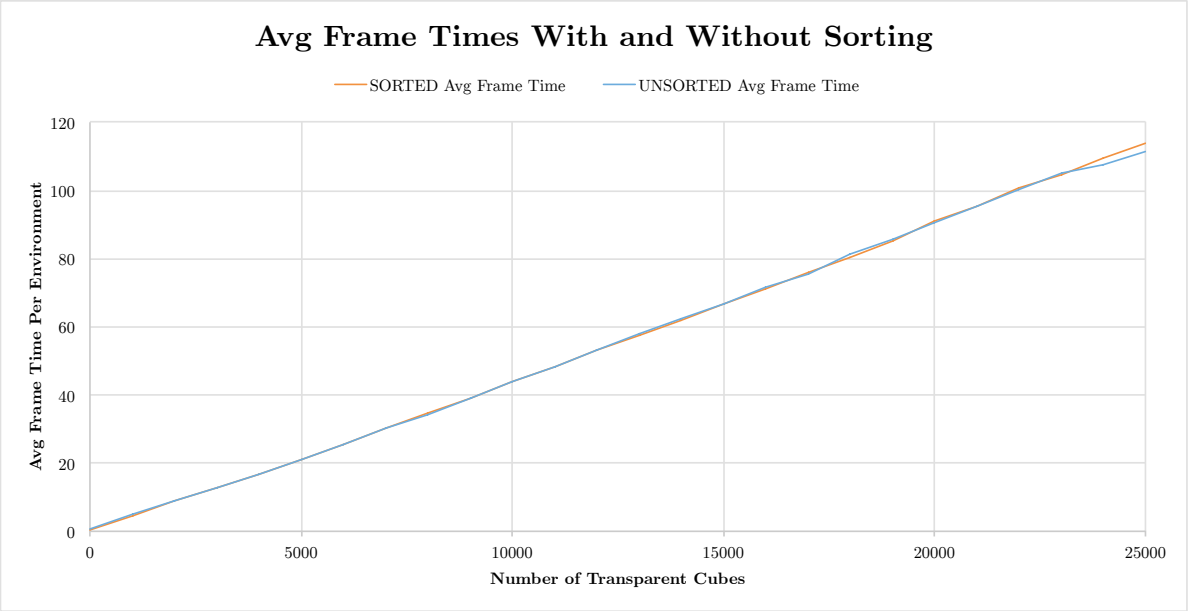


Figure 4.9: Average frame times in environments with up to 25,000 sorted and unsorted transparent primitives.

4.7.2 Using Transparent Textures

The second approach to implementing transparency is more fine-grained than the aforementioned approach and involves creating textures from images files that contain alpha channel information. Specific file formats such as GIF and PNG have the option of storing four channels (RGBA) per pixel instead of three (RGB) but they are not very flexible when it comes to adding an alpha channel to existing images later. With the June 2010 version of the DirectX SDK, Microsoft distributed a free set of tools with which an alpha layer can be added manually to images in a range of advanced image file formats. Examples of these are DXT1 through DXT5. Such formats allow the user to combine several image files into one composite file with the .dds extension and add alpha information to them in the form of a grayscale image, where darker gradients create a high degree of transparency and brighter grays indicate more opaqueness. Textures stored as DDS files that were created from DXT5 images, for instance, can be read in the pixel shader and used to set the pixel shader output's alpha component automatically.

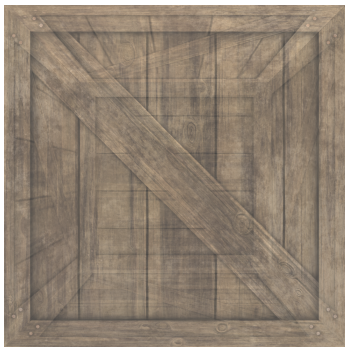
4.7.3 Comparison

We note that on on the one hand, per-pixel transparency through alpha values encoded in the image file yields more options to use transparency creatively. Certain shapes, such as wrought iron fences, are more easily created by applying a single texture with contained transparency information to a plane,

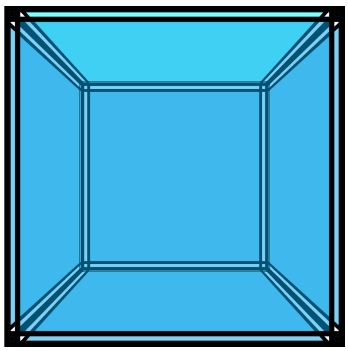
than by approximating the shape with many smaller fully opaque primitives. However, creating textures that include this transparency information correctly requires more work from the user than supplying transparency factors to regular primitives. It should also be noted that complex shapes that combine opaque and transparent elements can be created easily by combining opaque primitives with primitives with transparent faces, which foregoes the need for the user to familiarize themselves with DDS creation tools or specialized Photoshop plugins.

Additionally, images with contained transparency information generally require a high resolution to hide the fact that they are raster images, lest they look "blocky" to the user. This increases the strain on the amount of available video memory. Conversely, drawing models that consist of combinations of smaller fully opaque primitives and partly transparent primitives incurs a larger number of calls to the Direct3D device context to switch rasterizer states and transport blocks of memory from RAM to VRAM. However, experience has shown that the time spent on such transfers is negligible for scenes with a low complexity.

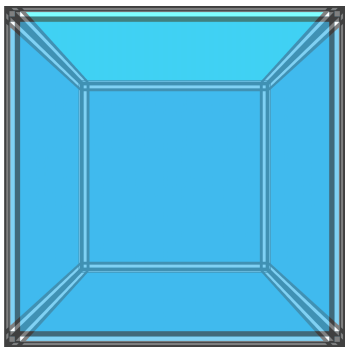
From a design standpoint, the per-face transparency factors are more in line with the primitive-centric world view Unicon promotes, where the first concern is accessibility. Regardless, both options are made available to Unicon. In the final release of UniconD3D, which is integrated in Unicon, both types of transparency are mixed: for each rendered pixel, the pixel shader assigns to the alpha value the maximum value of a user-specified transparency factor and an alpha value contained in the image. This lets one to combine both types of transparency. Figure 4.10 gives an impression of what the separate and combined transparency methods look like.



(a) A cube with a transparency factor of 0.5.



(b) A cube that uses per-pixel transparency to look like it contains openings.



(c) A cube with combined per-pixel transparency and a per-model transparency factor.

Figure 4.10: A transparent cube as seen from a one-point perspective, demonstrating the three supported types of transparency in UniconD3D.

4.8 The Camera

The implementation of UniconD3D's user-controllable camera was adapted directly from an implementation by Luna [43], although in practice, most implementations of such a camera are similar. UniconD3D's camera is defined as several vectors to denote its position and orientation, as well as a small number of functions to move the camera with respect to the camera's current heading. As in OpenGL, a camera has a position in 3-D space, an "up" vector to indicate its rotation over the X axis, and a "target" vector to denote its orientation over the Y axis.

UniconD3D's Camera class also maintains the current view and projection transformation matrices, which are sent to the vertex shader for each rendered frame. The parameters that are necessary to create a projection matrix are the smallest and largest viewable distances, to determine at which distances objects are rendered and when they become invisible, as well as a field-of-view that determines the camera's angle of view when the contents of a scene are transformed from view space into projection space. UniconD3D is set up to offer an angle of view of 45 degrees by default, which is a common value in commercial video games that prevents the graphics output from looking distorted. The (flat) plane in UniconD3D's projection matrix that delimits the far end of the camera's field-of-view is technically distorted, because this plane is perceived curved by human eyesight because our eye lenses are spherical. UniconD3D also does not apply vignetting, i.e., an exposure difference between the center of the output window and the window's edges. Vignetting effects could be trivially added to UniconD3D as a pixel shader in the future, to direct the user's attention to the center of the output window.

Chapter 5: UniconD3D Integration

After UniconD3D was determined to be feature-complete as a standalone rasterizer, the next development phase revolved around adapting UniconD3D to serve Unicon as a library. We will henceforth call this library UniconD3DLib. Our aim for UniconD3DLib is to have it contain all functionality needed to run CVE, which means that a subset of all Unicon-level graphics functions must have Direct3D-based implementations in UniconD3DLib. Any Unicon-level 3-D graphics functionality that is unused by CVE is omitted in UniconD3DLib in the interest of saving time.

UniconD3D underwent significant changes to bridge the gap between UniconD3D's data-centric world view and Unicon's primitive-centric world view. In all design choices during the integration process, our main consideration was to get CVE to work correctly, because CVE is one of the most graphically intensive applications that run on Unicon.

The integration process consisted of three phases: first, a list of Unicon-level graphics functions was prepared for which UniconD3DLib must contain equivalent functions. Because UniconD3D was written in C++ and Unicon's runtime library is written in C, UniconD3D was adapted to serve Unicon in the form of a static library with a wrapper API to make C++ functions available to C. In doing so, an example C program was developed that uses these graphics functions in much the same way Unicon does. The C program therefore acted as an early warning system to help spot compatibility issues. It was refined and expanded over time, and prompted the most significant changes to UniconD3D in the integration process. Section 5.4 discusses the portion of the Unicon-level 3D graphics facilities that are irrelevant to the goal of getting CVE running on Direct3D.

Subsequently, after UniconD3DLib provided all required functionality in the form of C-callable functions, a stub interface in the source code of Unicon's runtime library was expanded to use these functions. Because the list of Unicon-level graphics features mentioned previously was implemented precisely, and because they had been previously implemented in OpenGL, dropping the pieces of UniconD3D into place in the Unicon source code did not yield significant changes to the UniconD3D code, other than the order in which Direct3D is initialized, primitives are drawn, and the output window is refreshed. This transition is explained further in Section 5.2.

Finally, the performance of the resulting integrated subsystem was tested with two benchmark applications written in C. While the choice for benchmark scenarios is alluded to in Chapter 3, Chapter 6 motivates this choice in greater detail and discusses the benchmarks' implementations.

The complete list of Unicon-level functions for which C-callable implementations must exist in UniconD3DLib is included as Appendix D.

5.1 Window and Camera Operations

UniconD3D contains several window and camera operations that were made available to calling C code in Unicon when UniconD3D was made available as a library. For each output window in a running Unicon program, Unicon maintains a *window context* which contains variables that help configure UniconD3D. Examples of these are the window's title, the current color with which new primitives are drawn, a current texture that is applied to new primitives, and the window's resolution.

The process of creating a window class, opening a window with a given title that extends this window class, and showing the window on-screen was removed from UniconD3D and placed in the calling C code. This means that creating a window and showing it are now two distinct steps, which the UniconD3D library makes available through two separate functions. Additionally, while UniconD3D sets the dimensions of the window according to a C++ header file with configuration information in the UniconD3D source code, the window's dimensions are now set by the calling C code when the window is created, and the resolution stored in the window context is used.

While UniconD3D automatically closes the output window when graphics resources such as the Direct3D device and device context are destroyed, as a library, UniconD3DLib allows the programmer to close the window while frames are still being rendered without being displayed. The window can then be re-opened if needed.

The viewport, that is, the canvas to which new frames are rendered, was configured by UniconD3D to have the same resolution as the output window. As a library, UniconD3D now allows the calling C code to set the viewport resolution independently of the window resolution, which lets the Unicon programmer render a high-resolution or low-resolution image to an output window of any resolution, which should help tailor Unicon's performance to the system it runs on. The viewport may be changed as often as the programmers desires.

In UniconD3D, the position of the camera and the direction it faces were directly tied to user actions, such as keystrokes or mouse movements. The resulting library on the other hand defers camera control to a single function that sets the camera's position and heading directly. Changing the camera's position in small increments to create the illusion of movement has become the responsibility of the running Unicon program. However, whenever a frame is rendered, the camera's current position and heading are automatically used to update the camera's view and projection matrices, and to send these to the vertex shader in order to properly transform the scene's geometry during the current frame render.

In UniconD3D, the contents of the output window were automatically erased to an RGB color that existed in a configuration header file. This happened whenever a new frame was rendered without requiring user involvement. As a library, UniconD3D no longer does this without requiring user involvement, but instead offers two functions to instantly erase the window: one that takes red, green

and blue values to erase the window to a given RGB color, and one that erases the window to the current color as set in the running Unicon program's window context.

Although Unicon supports opening and using multiple windows to render 3-D graphics output to, UniconD3D does not, and operations such as clearing or closing a window pertain to a single UniconD3D output window. By extension, Unicon supports transferring a window's 3-D context to another window, while UniconD3D does not. This was deemed acceptable because CVE does not use multiple 3-D output windows, but others may choose to implement this feature in the future.

5.2 Creating and Drawing Primitives

A large disconnect exists between how Unicon composes a frame of rendered 3-D primitives, and how this is done in UniconD3D. For each 3-D output window, Unicon maintains a list of primitives to be rendered. This list can be changed at runtime, and whenever a frame is rendered, the list is traversed and each primitive is rendered in order. UniconD3D on the other hand maintains a more "data-centric" world view, which means UniconD3D reads the contents of a scene from an area file, and is subsequently free to change the contents of the vertex buffers, index buffers, render targets, transparency levels and more until Direct3D is fully initialized. At that point, the contents of all buffers are immutable, and only certain additional information for each primitive, such as whether it is visible or how it is transformed, may be changed between frame renders.

Central to the integration process is the understanding that area files are still used internally by UniconD3D to build and render virtual environments. Although UniconD3D could read in as many area files as it is instructed to, create a different scene for each area, and render different scenes at will, as a library, UniconD3D uses only a single area file. This area file is changed whenever UniconD3D is instructed to draw additional shapes: when a new primitive is added after Direct3D has been initialized (and possibly several frames have been rendered), the Area object is removed and replaced with a new one, the area file is re-read, and the vertex and index buffers are re-created and transferred to VRAM. The buffers are re-created quickly enough that the resulting performance loss is negligible.

We chose to keep using the existing human-readable text file format from UniconD3D in the library because these let the programmer change or copy the layout of a scene if they so choose, they make UniconD3D easier to debug if the integration process encounters any problems, and they make UniconD3DLib easier to expand by other Unicon contributors. Functions for manipulating area files are easy to write, and an area file is only accessed once before Direct3D is initialized, and once for each time it is changed afterward. After Direct3D is initialized, the contents of the area file also exist in memory and the area file itself becomes irrelevant until the calling C code changes it again.

This division between processing an area file and initializing Direct3D, which used to be one and the same in UniconD3D, was enabled by separating these two processes and requiring the calling C

code to initialize Direct3D with two separate function calls: one to read and process the area file, which fills the vertex and index buffers, and one to initialize the rest of Direct3D. The former function can be executed as many times as needed, incurring a negligible performance hit as mentioned above, the latter pauses the program's execution for several seconds on all test systems. If Direct3D resources are initialized without reading in an area file first, an empty area file is generated and used, which yields an empty output window.

Primitives are initially added before the vertex and index buffers are filled for the first time. Functions in the Area class, such as `DrawCube` or `AddLight`, write model descriptions, primitive descriptions, light sources and environment maps to the area file and use the fact that, while the Graphics class is not yet fully initialized, it can keep track of the current color, current texture, current transparency level, current transformation matrices and other information needed to describe primitives and light sources fully. Each subsequent occurrence of a function that draws a shape activates a flag that lets UniconD3DLib know that the contents of the area file have to be re-processed before the next frame is drawn.

In UniconD3D, certain operations, such as one that creates shader resources from all image files that are mentioned in the area file, were separated so they could be called at any time after Direct3D had been initialized. The resulting order in which the above functions are executed, is summarized in Figure 5.1.

Unicon lets the programmer decide how the vertices in a polygon or filled polygon are interpreted, but does not provide this choice for primitives whose vertices are generated automatically. UniconD3D on the other hand offers several such "mesh modes" to choose from, regardless of which type of primitive is drawn next. If the mesh mode is left/set to the default mesh mode, subsequent primitives are drawn as intended. For polygons and filled polygons, this means in the OpenGL implementation that such shapes are drawn as "triangle fans", wherein each vertex is connected to the first vertex and the most recently drawn vertex. In UniconD3D however, as described in Section 4.3, this means a polygon's vertices are interpreted as a list of lines, and a filled polygon's vertices are interpreted as a list of triangles. Likewise, front face and back face culling are normally enabled or disabled automatically by UniconD3D depending on the type of primitive that is drawn, but the UniconD3D library offers a function to turn either option on or off.

Because Unicon does not support model hierarchies, as explained in Chapter 4, the UniconD3D library places each primitive in a model with a default name ("default") and the default transformation matrix. Each light source is also enclosed in its own minimal model description.

Unicon allows the programmer to specify the thickness of lines in polygons, lines, line segments and points, but UniconD3D does not allow this. Recent versions of OpenGL and Direct3D disallow specifying the thickness of rendered lines and points. One could cheat by writing a geometry shader to automatically expand specific pairs of vertices to form a quad between them, of which the thickness

depends on the quad's distance to the camera. This way, a line would show up as having the desired thickness. However, because CVE does not use lines much, we decided to omit this feature.

Environment maps are currently unsupported in Unicon, but a function for adding one has made it into the UniconD3D library. By default, the UniconD3D library adds an empty environment map at the top of each area file, which may be filled in programmatically when environment maps are used by Unicon in the future.

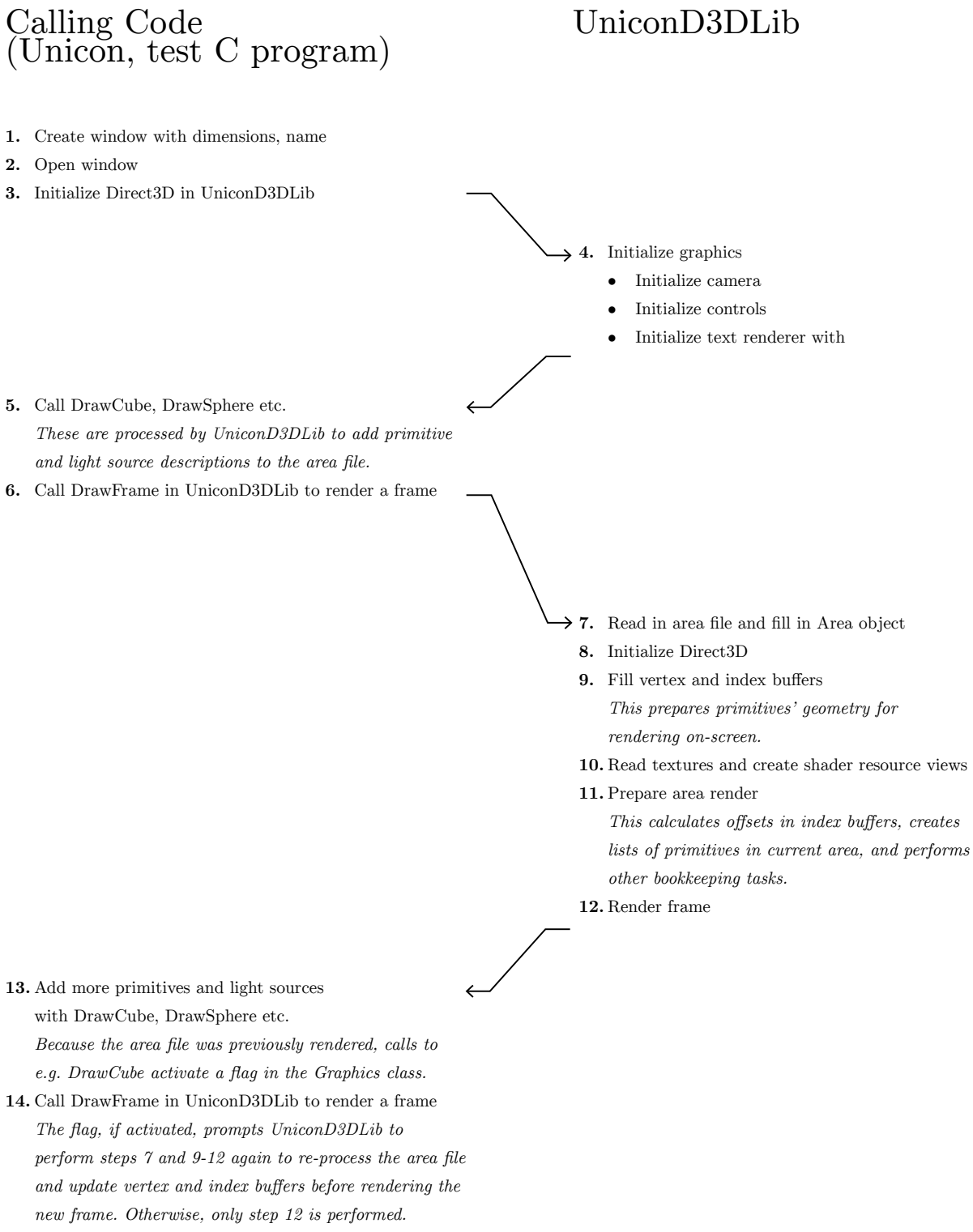


Figure 5.1: Abstracted overview of the rendering process in UniconD3DLib.

5.3 Transformation Matrices

Unicon uses OpenGL's stack-based implementation of transformation matrices to select the proper transformation for subsequently drawn primitives or for the camera. To this end, Unicon maintains two matrix stacks: one for primitives and one for the camera, and these can be switched between using Unicon's `MatrixMode` function. Whenever a primitive is rendered, the top of the model transformation matrix stack is used to transform the model's vertices, and whenever the camera is updated, the top of the camera transformation matrix is used to transform the camera's position and heading.

UniconD3D on the other hand maintains three transformation vectors per primitive, creates transformation matrices with these vectors on the fly for each frame draw operation, and sends the transformation matrices to a vertex shader which transforms a given model's vertices in parallel. This takes some of the load of rendering models off the processor and delegates it to the graphics processor.

In order to convert Unicon's approach to assigning transformations to primitives into UniconD3D's approach, the Graphics class was altered to hold stacks of transformation vectors. Stacks of vectors are used instead of stacks of matrices, because the area file lists primitive transformations in terms of a translation, scaling and rotation vector, and because these vectors can not be retrieved with complete certainty from a transformation matrix that contains a translation, a scaling and a rotation. Whenever a transformation operation is applied at the Unicon level, a corresponding function is called in the UniconD3D library, i.e., `translate`, `scale` or `rotate`. Each of these functions updates all three stacks of transformation vectors, and Unicon-level matrix operations like `pop` or `identitymatrix` affect all three stacks of vectors as well. This approach removes the problem of determining how Unicon-level transformation matrices are composed (row-major, column-major, left-handed, right-handed).

In UniconD3D, each primitive contains its own transformation matrix, which is transformed by the view and projection matrices in the camera. Correspondingly, the functions that add new primitives in the UniconD3D library allowed the calling C code to specify these transformation vectors when new primitives were created. This feature was removed from the final UniconD3D library because at the Unicon level, only a translation vector is given for each primitive.

5.4 Textures and Materials

Unicon uses the current color in the graphics context to color subsequently drawn primitives. Textures are similarly applied. UniconD3D did not originally include color information in primitive descriptions in area files, but later added a color to each primitive which was passed into the vertex shader as an argument, only to be output by the vertex shader as an argument into into the pixel shader, where it would be picked up and used instead of the texture normally associated with the primitive, if there was any texture. However, during its conversion into a C-callable library, UniconD3D finally settled on

including color information in the vertices of each primitive to reduce the amount of communication through vertex shader constant buffers. As a result, each primitive in the UniconD3D library has a color that may or may not be ignored in favor of a texture.

Although lines, unfilled polygons and points could be textured in the OpenGL version of Unicon's graphics facilities, more recent versions of OpenGL and Direct3D enforce a thickness of 1 pixel for each rendered line and point, thereby reducing the usefulness of textures on lines and points. Therefore, adding color is the only way to change the appearance of lines and points in the UniconD3D library. Any texture applied to a polygon, line or point is ignored by UniconD3D.

As explained in Section 4.5, UniconD3D allows the programmer to specify an arbitrary number of textures for each primitive, so long as a list of indices was also provided. The indices would mark where each respective texture would start to be used by the pixel shader. Because Unicon allows the programmer to specify only one texture per primitive, which is repeated on each primitive face, the UniconD3D library uses only one texture per primitive, i.e., the current texture in the window context. The necessary code to use an arbitrary number of textures, is still included with the UniconD3D source code, but has been disabled for the sake of uniformity[1].

In UniconD3D, textured primitives whose textures cannot be found were not rendered at all. This design decision made sense because a missing texture indicates a programmer error, and it is desirable to hide such errors when possible. This was changed in the library: textured primitives with missing textures show up as having either the current color, white or black. Which of these options is used can be set by the programmer at any time, but the default option is to use the current color.

Unicon features a function that takes the contents of a 2-D window or 3-D window, creates an in-memory texture from them, and applies this texture to a primitive. UniconD3D on the other hand contains a function that re-converts an existing texture into a shader resource view and applies it instantly to the primitives that use this texture. In order to combine both functions to let a primitive in the UniconD3D library use the contents of a window as a texture, the UniconD3D library contains a function that creates a shader resource view of said window contents and saves it to the disk with the appropriate file name so a target primitive uses it as a texture.

Because the function that updates shader resource views based on changes in the on-disk image file yields no significant frame rate drops, the UniconD3D library provides no functions for turning this functionality on or off. However, the library does provide a function that sets the refresh speed of such update operations.

[1]This code also helped the programmer abstract away from the primitive's indices by allowing them to instead specify a list of primitive face numbers. These face numbers are explained in Section 4.3.

5.5 Omitted Graphics Functionality

Two kinds of features were omitted in the UniconD3D library. Firstly, UniconD3D does not support drawing to multiple output windows while the Unicon-level graphics instruction set does. Where needed, Unicon-level graphics function signatures that specify a window to apply an operation to were changed in UniconD3D to omit a selected window altogether.

Secondly, a function that transfers the textures from one window context to another window context was omitted in UniconD3D, because only one window context will be used with a running UniconD3D process. Another function, which selects a given window as the "current" window on which subsequent graphics output is drawn, was also left out of UniconD3D.

All functions that draw 2-D shapes directly onto an existing texture were left unimplemented in UniconD3D, because UniconD3D in its current state does not support drawing directly onto a texture. These functions are also not used by CVE. Similarly, the function that sets the width of subsequent lines and points also does not exist in UniconD3D.

5.6 UniconD3D Integration Into Unicon

After the example C program was expanded to include all functions CVE is likely to use, the contents of the C program were used to compare the arguments required by Unicon-level functions with the arguments to the C-callable functions in UniconD3DLib. These largely corresponded directly, with the exception that UniconD3DLib does not support operations on multiple output windows, as explained in the previous section.

UniconD3DLib was subsequently compiled and linked into Unicon's runtime system. For each discovered call to the OpenGL-based facilities by the Unicon runtime system, a similar call is made to the Direct3D-based library. In many cases, the argument lists provided by Unicon-level function calls to the graphics subsystem did not match those expected by their equivalent functions in UniconD3DLib. The C-callable interface in UniconD3DLib was adapted to provide the expected interface to Unicon where needed. This process revealed one missing functionality in UniconD3DLib: Unicon must be able to turn any of the eight light sources in an environment on or off at will, while UniconD3DLib only provided functions for adding lights, without letting the user change a specific light source afterward. This issue was trivially resolved. Additionally, the functions that create a window and open it, which were previously delegated by UniconD3DLib to the example C program, were made part of UniconD3DLib's interface to Unicon. Otherwise, no missing functionality in UniconD3DLib was discovered during the integration process.

Chapter 6: Evaluation

This chapter motivates and describes the benchmarks used to validate the performance of UniconD3D, as well as the benchmark results. These benchmarks are run on the Direct3D-based graphics library only.

6.1 Existing Unicon Benchmarks

At the time of writing, no benchmarks exist that test Unicon's built-in 3-D graphics facilities. Graphics benchmarks for various platforms exist in the form of commercially available packages, such as FutureMark's *3DMARK* [44], but none exist that test 3D graphics performance in Unicon programs. We therefore develop benchmarks to test UniconD3D's graphics performance against a range of situations that represent the complexities of environments one encounters in CVE. Although these benchmarks test the performance of UniconD3D in isolation, the overhead of any calls from Unicon to UniconD3D is not incorporated in them.

The existence of the Icon benchmark suite [45] prompted the development of the Unicon benchmark suite, which adds to the Icon benchmark suite a number of benchmarks to test integer and floating-point arithmetic, programmer-defined generators, suspension and backtracking, tables, sets and threads [46]. These areas were tested by executing algorithms developed for the Computer Language Benchmarks Game [47], which does not test a language's built-in graphics facilities.

6.2 Benchmark Setup

In each of these benchmarks, UniconD3D is instructed to render a series of environments with increasing numbers of primitives. The range of the number of primitives we expect UniconD3D to render satisfactorily lies in between what one could find in commercial video games between 2002 and 2007. We expect UniconD3D to render scenes with complexities akin to those in state-of-the-art video games from 2002 easily, while scenes in games from 2007 onward should prove too complex. *Halo: Combat Evolved* constituted the state of the art in video game graphics when it was published in 2002 for Microsoft's newly released Xbox, and the rendered portion of a virtual environment in Halo CE (called a "portal") contained between 10,000 and 15,000 triangles, which is to say, between 833 and 1250 cubes, since one cube contains 12 triangles. In 2007, *Crysis* fulfilled a similar role, and would draw between 8,000 and 10,000 triangles in menus, and around 1,000,000 triangles per frame in most in-game scenes at the "high" graphics settings.

CVE does not normally use nearly as much geometry as Crysis in any of its environments, but we maintain a ceiling value for the scene complexities of 1,000,000 triangles, or 85,000 cubes. We choose to

render sets of cubes instead of another type of primitive, or a collection of different types of primitives, because there is an inherent performance penalty connected to rendering many simple primitives rather than just a few very complex primitives, such as spheres or tori.

The results of the benchmarks described below, as well as the benchmarks themselves and the specifications of the system they were run on, can be downloaded from `http://unicon.org/unicond3d`.

Benchmark 1: Frame Rates and Frame Time Variances

We display an empty environment, which is initially expanded with 100 additional cubes after every 60 rendered frames. After the first 150 such environments, in which 15,000 cubes are drawn in total, 500 cubes are added after every 60 rendered frames. After 50 such scenes, in which the environment's complexity increases by 25,000 cubes in total, cubes are added in increments of 5000 until the environment contains 85,000 cubes. We chose to reduce the granularity of the increases in environment complexity for three reasons. One the one hand, frame times become unacceptably large in environments with more than 11,500 cubes, resulting in frame rates below 24 frames per second, and on the other hand, the complexity of a typical scene in CVE is not expected to approach this number. Thirdly, the lower granularity yields significant time savings, since environments with over 50,000 cubes take over two minutes to load.

The frame time is recorded for each rendered frame and written to a comma-separated value (CSV) file, which contains 51,000 frame times. The time it took to render a typical frame in scene n, which contains $100n$ rendered primitives, is computed as the average over its 60 runs. The resulting average, expressed in milliseconds, is translated to a frame rate expressed in a number of frames per second. We choose a suboptimal scenario, wherein half the primitives are transparent, all primitives are textured and the locations of all primitives are randomized. Included in the frame time of the first frame is the time required to process the area file, which is relatively expensive, whereas the subsequent 59 frames that depict the same scene are relatively cheap to render. This approaches the use case where 100 primitives are added to a scene each second, which does not happen in CVE. The scenes are rendered at a resolution of 1800 by 1000 pixels.

The same test is used to find the highest and lowest amounts of frame time variance per environment. We calculate the difference between the frame time of each frame and the mean frame time of the environment, and express the resulting ranges in milliseconds.

Benchmark implementation

In order to measure the elapsed time between frame draws, either of the following two Windows kernel functions can be used:

- `void WINAPI GetSystemTimeAsFileTime(LPFILETIME lpSystemTimeAsFileTime)`

- `BOOL WINAPI QueryPerformanceCounter(LARGE_INTEGER *lpPerformanceCount)`

The second option was used. Its resolution of the former function depends on the system clock resolution, which can be verified with the *ClockRes* tool from Microsoft SysInternals [48]. The system used for the benchmark evaluation has a timer interval of 1.000 ms. The second option however measures system time in CPU clock ticks for a limited time (up to 47.1 days), which is more accurate. Other Windows-specific timing functions, such as `DWORD WINAPI GetTickCount(void)`, have a resolution in the range of 10 milliseconds to 16 milliseconds.

The code for this benchmark is included in Appendix E.

Benchmark 2: Lighting Performance

To benchmark the performance of UniconD3D's lighting system, we take 16 scenarios with complexities ranging from zero to 15,000 primitives. The number of primitives in scenario n is $\frac{15000n}{15}$. We display each such scenario eight times with an ascending number of light sources. We set the range of the light sources (i.e., how far the light will reach before lighting calculations are stopped in the pixel shader in UniconD3D) to a near-infinite value so lighting calculations are performed for all primitives in a scene. We expect the resulting eight series of average frame time measurements to be very close to each other, because the lighting calculations are performed in parallel in a pixel shader and the time they require should be negligible.

6.2.1 Hardware Setup

The system used to run the benchmarks uses the components listed below. All components are set to factory specifications.

- CPU: Intel Core i7 960 (4 cores, 8 threads 8MB L2 cache, base frequency of 3.2 GHz and turbo boost frequency of up to 3.46 Ghz)
- RAM: 18 GB of DDR3 at 1333 MHz, with 7-7-7-20 timings
- HDD: Corsair Force GT 240 GB, connected through LSI MegaRAID 9260-8i
- GPU: Asus GTX670-DC2T-2GD5 with 1344 CUDA Cores and 2GB of GDDR5 VRAM [49], in conjunction with 64-bit Nvidia GeForce drivers version 344.75 for Windows 8.1 [50].
- OS: Windows 8.1 Professional, includes all updates for Microsoft Windows up to KB3012199.

6.3 Results

6.3.1 Frame Rates and Frame Time Variance

The results from benchmark 1 are summarized in two figures. Figure 6.1 illustrates the average frame times per environment in milliseconds, as well as the minimum and maximum frame time variances per environment. Because of the small differences between the minimum frame time differences and the average frame rates, the graph's Y axis uses a logarithmic scale.

When the frame times are measured, the time required to read and process the area file are added to the first frame time of each environment. As the environment's complexity increases, the time required to process the area file increases, and so does the time required to sort all transparent primitives in the scene. The fact that all transparent primitives must be sorted before they are drawn, is computationally expensive: the time complexity of the selected sorting algorithm, quick-sort, is $O(\log(n))$ at worst. The results of benchmark 1 make clear that too much time is spent on processing changes to the area file when it already contains over 8300 primitives: that is where the highest frame time first exceeds 16 milliseconds.

The fact that the minimum frame time variances (the bottom graph) so closely approximate the average frame times confirms empirical findings that once an area is processed, subsequent frames are rendered quickly until the area needs to be re-processed.

An alternative organization of the vertex and index buffers, which is explained in Chapter 9, should remove the high maximum frame time variances, because in that organization, vertex and index buffers need never be destroyed and rebuilt. The difference in performance such a reorganization may yield is illustrated in Figure 6.2. Although the minimum acceptable frame rate (24 frames per second) is achieved only in scenes with at most 900 primitives, if the first frame time from each environment is ignored, the minimum frame rate can be sustained in environments with up to 11,300 primitives.

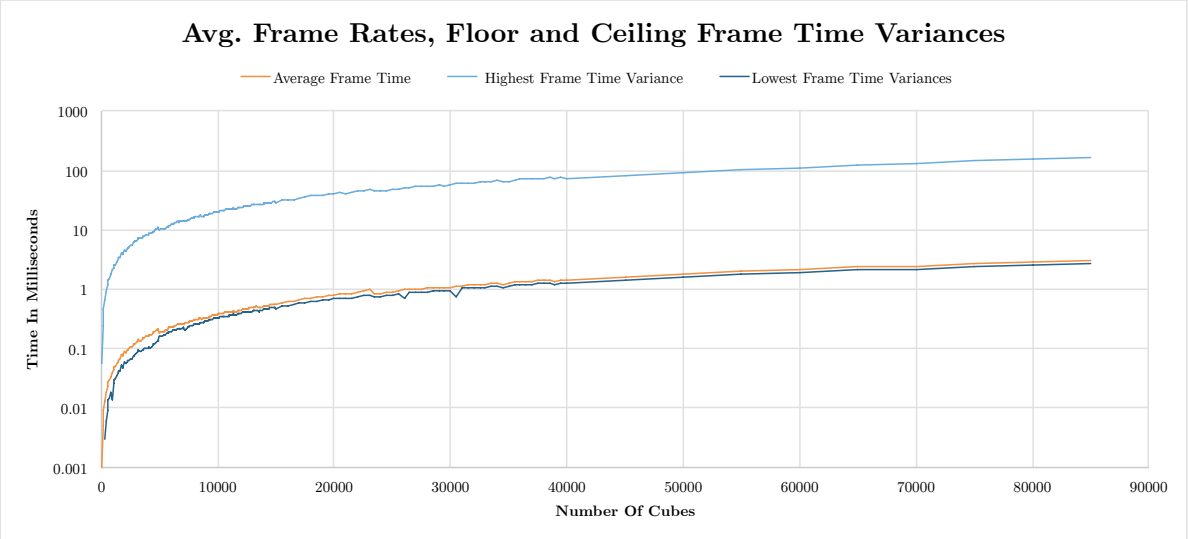


Figure 6.1: Average frame time and the range of minimum and maximum frame time variances. The Y axis uses a logarithmic scale with base 10.

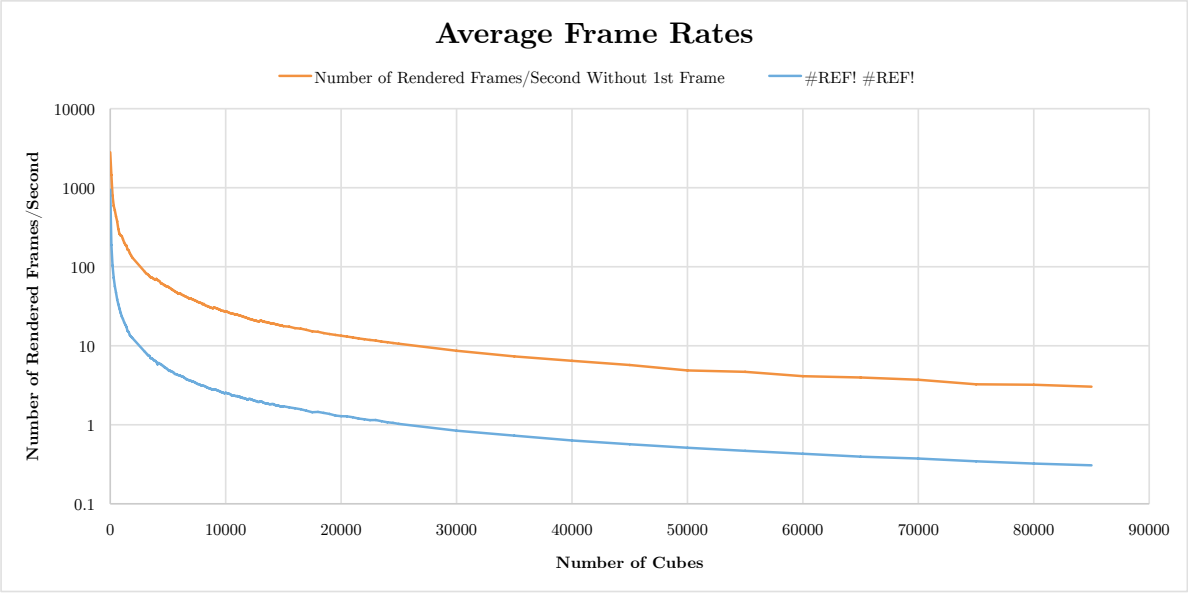


Figure 6.2: Average frame rates with the first frame time included in and precluded from each environment's average frame times. The Y axis uses a logarithmic scale with base 10.

6.3.2 Lighting Performance

The average frame times in each of the 16 environments, with 0-8 light sources, are shown in Figure 6.3. As with the frame rate evaluation in Figure 6.2, the frame time of the first frame in each rendered environment was not included in the average. The differences in average frame times for all environments are virtually the same, any differences between them lie in the order of a few milliseconds. Such variances may be attributed to other (unrelated) processes running on the evaluation system. By extension, because the average frame times in environments with eight light sources differ negligibly from those in environments with zero light sources, adding more light sources would not add significantly to the frame times in environments that contain more primitives.

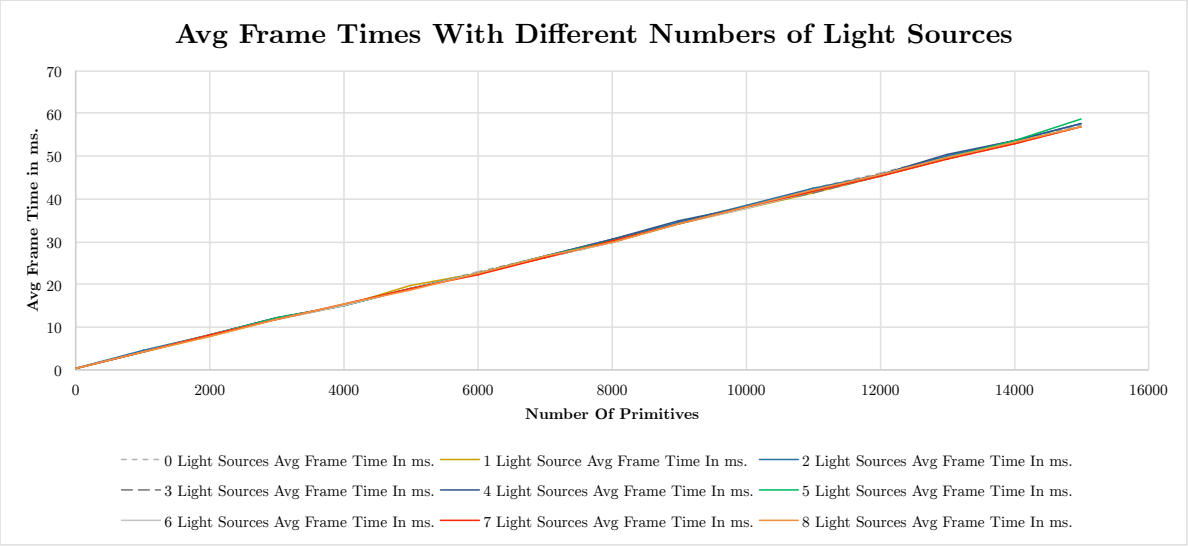


Figure 6.3: Average frame times for environments with 0-8 light sources 0-15,000 primitives. The first frame time in each environment is precluded from the environment's average frame time.

Chapter 7: Conclusions

This chapter summarizes the findings and conclusions of the thesis work, and gives a list of recommendations regarding the design of Unicon's high-level graphics instruction set, as well as improvements to the Direct3D-based and OpenGL-based graphics facilities.

7.1 Thesis Objective

We have created a high-level graphics library called UniconD3D that uses Direct3D 11.0 to perform basic 3-D graphics operations, specifically, those required by the Collaborative Virtual Environment. It has been made C-callable and it is tailored to the needs of the Unicon programming language. It has been linked into and compiled with the Unicon runtime system and will serve as a complementary set of graphics facilities alongside Unicon's existing OpenGL-based graphics subsystem. The amount of work required to integrate UniconD3D fully into Unicon's runtime system and to run CVE with UniconD3D to provide 3-D graphics is limited to debugging the portions of Unicon's runtime system that call the Direct3D-based facilities.

UniconD3D is an effort to future-proof the Unicon programming language on Windows platforms, and provides a new perspective on the design of Unicon's high-level 3-D graphics instruction set. This may lead to the removal of OpenGL-specific functions from the Unicon-level graphics instruction set, and the addition of new Unicon-level graphics instructions.

We have also quantified the performance of UniconD3D by rendering a range of virtual environments with complexities that match or exceed those in the Collaborative Virtual Environment, and found that UniconD3D delivers acceptable frame rates and frame time variances in environments with up to 138,000 triangles, or 11,500 cubes. Depending on the number of primitives in the environment, the first frame in an environment takes between several milliseconds and almost 3 minutes to render, while subsequent frames are rendered in a fraction of this time. Additionally, because the lighting system is implemented largely in pixel shaders, lighting calculations are performed in a highly parallel fashion and yield no significant performance differences between scenes with one light source and those with eight light sources.

7.2 Recommendations

This section proposes future changes and extensions to the new Direct3D-based and the existing OpenGL-based 3-D graphics facilities in Unicon.

7.2.1 Suggested Additions to the OpenGL-Based Facilities

Image Formats

Unicon's OpenGL-based graphics could benefit from accepting images in a wider variety of formats, including composite image types such as *dds*. This would let one create environment maps for skyboxes or planar reflections, which require an image to span the entire field of view.

Environment Maps

Environment maps are unsupported in the Unicon language, because corresponding high-level Unicon instructions are missing, and in Unicon's OpenGL-based graphics facilities because OpenGL 1.2 does not natively support composite image file formats.

Finer Granularity for Texture Mapping

Geometric primitives like cubes of cylinders could be made to look more realistic when they contain multiple textures. A cube for instance can hold six textures, one for each face, and a cylinder could hold three textures, one that wraps around the mid section, one for the top cap and one for the bottom cap. Although UniconD3D supports placing textures at arbitrary triangles or at arbitrary texture faces, Unicon does not.

One way for the user to specify multiple textures for a single primitive is to have him or her enter a list of image file names and a list of face numbers, so that each image file is converted into a texture that is applied to its corresponding face number. Calculating which vertices a texture is applied to is then deferred to the graphics subsystem.

A computationally cheaper way to specify multiple textures per primitive is to provide a way to combine multiple textures in a single file. Unicon would need a set of rules for specifying "templates" for such combined image files, so that the Unicon programmer knows where the texture for each primitive face is placed in the image file.

Transparency

Unicon lacks the commands to set per-pixel or per-face transparency of a primitive shape depending on the alpha channel of the shape's texture. Instead, it relies on high-level commands to set the transparency of a shape on a per-primitive basis. In order to accept per-pixel transparency, Unicon's OpenGL-based graphics facilities must calculate the alpha value of pixels in a pixel shader and accept textures that store more than 24 bits per pixel

As a way to sort the primitives in a scene more quickly, one could divide them into sets, based on how far away from the camera they are. First, the most recent sorting of primitives would have to be cached

in RAM. Subsequently, in the simplest case, the primitives could be divided into two groups, where only the primitives in the latter half of the cached sorted list (i.e., the primitives closest to the camera) are updated each frame, and the primitives in the first half are updated every other/third/fourth/etc. frame. This means that in the rare case many transparent primitives are located close to the camera, some may occasionally appear to be rendered in the wrong order. To largely prevent this from happening, this technique could only be applied to scenes with more than n primitives for some threshold value n.

7.2.2 Additions to the Graphics Facilities at Large

Lighting

The number of lights in UniconD3D is near-infinite because lighting calculations are performed exclusively in pixel shaders. Even so, the number of lights is limited because each additional light source incurs a performance penalty, since lighting calculations are performed sequentially for all light sources. Deferred shading techniques, including those that deal with deferred lighting, have offered a less resource-intensive solution to render environments with hundreds of light sources. We therefore recommend that whenever light sources are added with Unicon's high-level graphics instructions, they are calculated in shaders to increase the maximum number of possible light sources.

Tessellation

Organic geometric primitives should not be rendered with great detail when they appear small on-screen. Tessellation stages were added to the DirectX and OpenGL rendering pipelines to let programmers set the level of detail of such shapes, but Unicon offers no directives for when tessellation should be used, and Unicon's OpenGL-based graphics facilities currently do not offer support for tessellation.

Texture Filtering

Textures appear blurry when viewed at an angle, this could be ameliorated when they are filtered with any modern texture filtering technique, such as trilinear, bicubic or anisotropic filtering.

Collision Detection

Collision detection could be deferred to the 3-D graphics subsystem, which can add a bounding box to each primitive and perform collision detection calculations on these bounding boxes. Collision detection may prevent the camera from passing through a primitive unless a specific Unicon directive is passed to the primitive when it is created, or may prevent two primitives from occupying the same space.

Unicon might specify whether the camera or a primitive may pass through another primitive with directives to make the space occupied by a primitive traversable, inaccessible, or accessible while performing some other action. For instance, the steps of a staircase may consist of primitives that render

the space they occupy inaccessible, but if the camera attempts to move "into" these primitives, the camera may be moved up or down rather than being brought to a halt.

In many modern video games, low-precision real-time collision detection is part of a *physics engine*, a subsystem that works closely with a game's graphics engine to calculate rigid body dynamics, soft body dynamics and fluid dynamics. GPU support through specialized physics-related instruction sets has gained popularity since the company Ageia was purchased by Nvidia in 2008 and their work in hardware physics acceleration was incorporated in Nvidia GPUs [51].

Shadows

Unicon and its OpenGL-based and Direct3D-based graphics subsystems have no support for real-time shadows. Adding these would increase the realism of a scene significantly.

Animations

Animations are currently handled at the Unicon level, which means each primitive is transformed sequentially, one after the other, to make it appear as though several primitives are animated at once. This principle could be enhanced by adding simple animations to the graphics subsystem as, for instance, an animation speed and a sequence of 3-D coordinates that a primitive must move through. This way, the additional primitive transformations for animations are handled without needing attention from the Unicon programmer after the animation has been preconfigured once.

Normal Mapping

Normal maps may be added to a primitive to add an extra layer of depth to the primitive's textures. A normal map functions as a low-resolution texture that hold for instance 8 or 16 bits per pixel, but this information is used to in addition to the per-vertex normals to affect how lighting strikes an object, albeit at a much finer granularity. Normal maps are therefore commonly used to emphasize more clearly what material a surface is made of (a normal map that is mono-valued across all pixels will make the object's surface it is applied to look perfectly smooth).

Drawing on Textures

Instead of maintaining a separate window to draw to and to use this window as the texture for a primitive face, it could be useful to support drawing directly on a primitive's surface in future iterations of Unicon. This is not currently supported in UniconD3D.

7.2.3 An Alternative Re-Implementation of UniconD3D

What follows is a proposed technique to let UniconD3D make more efficient use of the vertex and index buffers at the expense of increased CPU-GPU communication. It is not implemented in the submitted version of UniconD3D.

UniconD3D currently stores many copies of the same vertex and index layouts in the vertex and index buffers. There is a way to store only one version of each primitive in these buffers. The layout of the first few hundred vertices and indices would be fixed, which means the code for rendering a frame could be reduced in size, for two reasons. Firstly, the vertices and indices of all areas need no longer be gathered prior to constructing the vertex buffer. Secondly, the index locations of a primitive that is about to be rendered are known at compile-time, which means UniconD3D no longer needs to compute them for every frame.

While reading in an area file and constructing Primitive objects from the human-readable descriptions, the only information that needs to be stored per primitive are parameters that are not included with the vertex and index lists. Saying that the next shape to be rendered is a sphere would be enough to render the geometry of a sphere. In exchange for this convenience, information that is currently encoded in the vertices, such as a primitive's color and transparency, will need to be exchanged between the CPU and shader constant buffers, which could potentially slow UniconD3D down. Specifically, if a future version of UniconD3D allows the programmer to specify a per-vertex color and transparency factor, rendering a single sphere could require hundreds of constant buffer updates.

A problem then arises when the user draws shapes with geometry that is not precomputed: what if one needs to draw a sphere with a specific number of rings and slices? In this case, the tessellation stages could be used to expand the primitive with a default number of rings and slices into a more complex primitive with the desired number of rings and slices.

The graphics class would maintain a list of all primitives, their start and end indices in the index buffer, and their names. Completely custom shapes that a tessellation cannot generate, like polygons and filled polygons, can be added to this list, and their vertices can be added to the vertex buffer. The same may happen with lines because it is significantly easier to specify the locations of vertices in a line than it is to specify a line at the origin, and to have to translate, scale and rotate it to get it at the right location. A "point" primitive could exist as a separate vertex in the vertex buffer, or it may even be stored as the very first vertex in the vertex buffer, as part of the vertices of a plane. By extension, the vertices in a plane may just as well be four of the vertices in a cube, to reduce the strain on the vertex buffer even further. In case the list of different primitives becomes very large, it could even be stored in a more efficient data structure (e.g., a binary search tree to reduce the time complexity of searching for the right start index from $O(n)$ to $O(log(n))$. If using tessellation is not an option, this property could also be exploited by assigning a number to each primitive in a scene that is the outcome

of a hashing function. For instance, a cube could have number 2, say, while a sphere with n rings and m slices could have number $1000 + m + \frac{1}{n}$. This makes it easier to look up the start and end indices for a primitive in the list, and prevents the programmer from adding multiple copies of a given primitive to the vertex and index buffers.

Although the speedup gained by not having to precompute the locations of all primitives in VRAM is welcome, and although this technique may work especially well with Unicon because Unicon-level graphics instructions don't allow for per-vertex changes in the rendering process, it is worth nothing that the vertex buffer is usually large enough to hold all shapes, regardless of how many copies of primitives are stored in it. Modern GPUs have several gigabytes of VRAM at their disposal and each vertex takes up 48 bytes in UniconD3D. The greatest benefit to this new system would be the severely reduced amount of bookkeeping in the graphics class to keep track of which indices will be rendered next.

A much more welcome effect of using such a technique is that using it removes the problem of having to re-initialize the vertex and index buffers when a new shape is read in after a frame is rendered, because all vertices and indices are already set, unless a new polygon, filled polygon or line is added. This issue also exists for spheres, cylinders, disks and tori if tessellation is not used.

Bibliography

[1] C. Jeffery, S. Mohamed, J. A. Garaibeh, R. Pereda, and R. Parlett, *Programming with Unicon*. Unicon Project, 2nd ed., 2013. `http://unicon.org`.

[2] R. E. Griswold and M. T. Griswold, *The ICON Programming Language*. Annabooks, 3rd ed., 1996.

[3] C. Jeffery, "Unicon.org – The Unicon Programming Language Homepage," July 2014. `http://unicon.org/`.

[4] OpenGL.org, "Core And Compatibility in Contexts," October 2012. `http://www.opengl.org/wiki/Core_And_Compatibility_in_Contexts`.

[5] Microsoft Corporation, "Windows Advanced Rasterization Platform (WARP) Guide (Windows)," December 2014. `http://msdn.microsoft.com/en-us/library/windows/desktop/gg615082(v=vs.85).aspx`.

[6] Microsoft Corporation, "CodePlex – Open Source Project Hosting," December 2014. `http://www.codeplex.com`.

[7] Microsoft Corporation, "Visual Studio – Download Overview," accessed in December 2014. `http://www.visualstudio.com/downloads/download-visual-studio-vs`.

[8] The Valve Linux Team, "Faster zombies!," August 2012. `http://blogs.valvesoftware.com/linux/faster-zombies/`.

[9] Microsoft Corporation, "Effects (Direct3D 11)," accessed in December 2014. `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476136(v=vs.85).aspx`.

[10] C. Walbourn, "Effects for Direct3D 11 Update," October 2012. `http://blogs.msdn.com/b/chuckw/archive/2012/10/24/effects-for-direct3d-11-update.aspx`.

[11] M. Sandy, "DirectX 12," March 2014. `http://blogs.msdn.com/b/directx/archive/2014/03/20/directx-12.aspx`.

[12] C. Jeffery, "CVE – A Collaborative Virtual Environment," October 2013. `http://cve.sourceforge.net/`.

[13] C. Jeffery, "Visualizing software ecosystems as living cities," in *HCI International 2013 – Posters Extended Abstracts* (C. Stephanidis, ed.), vol. 373 of *Communications in Computer and Information Science*, pp. 640–644, Springer Berlin Heidelberg, 2013.

[14] S. Gaikaiwari, *Adapting Snobol-style Patterns to the Unicon Language*. Las Cruces, NM, USA: New Mexico State University, 2005. http://www.cs.nmsu.edu/~sgaikaiw/Thesis.pdf.

[15] L. G. Shapiro, "ESP3: A High-level Graphics Language," *SIGGRAPH Comput. Graph.*, vol. 9, pp. 70–77, Apr. 1975.

[16] JogAmp.org, "JOGL – Java Binding for the OpenGL API," accessed in December 2014. http://jogamp.org/jogl/www/.

[17] JogAmp.org, "JOGAMP – Graphics, Audio, Media and Processing Libraries," accessed in December 2014. http://jogamp.org.

[18] JogAmp.org, "JOGL Specification Overview," July 2013. http://jogamp.org/deployment/jogamp-next/javadoc/jogl/javadoc/overview-summary.html#overview_description.

[19] Khronos Group, "OpenGL ES 2.0 for the Web," 2014. https://www.khronos.org/webgl/.

[20] The OpenTK community, "OpenTK – Home of the Open Toolkit Library," accessed in December 2014. http://www.opentk.com.

[21] The TAO community, "The Tao Framework," April 2013. http://sourceforge.net/projects/taoframework/.

[22] The Sankaty Group, "Open Source Development for the D Programming Language," November 2011. http://www.dsource.org/projects/bindings/wiki/DirectX.

[23] A. Mutel, "SharpDX – Managed DirectX," 2012. http://sharpdx.org.

[24] SlimDX Group, "SlimDX Homepage," 2011. http://slimdx.org.

[25] Silicon Studio Corp., "Paradox – Game Engine," accessed in December 2014. http://paradox3d.net.

[26] MonoGame Team, "MonoGame – Write Once, Play Everywhere," accessed in December 2014. http://www.monogame.net.

[27] Haxe Foundation, "Haxe – The Cross-platform Toolkit," September 2014. http://http://haxe.org.

[28] Haxe Foundation, "HxSL – Haxe Shader Language," December 2013. http://old.haxe.org/manual/hxsl.

[29] A. Stromme, R. Carlson, and T. Newhall, "Chestnut: A GPU Programming Language for Non-experts," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, (New York, NY, USA), pp. 156–167, ACM, 2012.

[30] R. McAlinden and W. Clevenger, "Using commercial game technology for the visualization and control of constructive simulations," in *Proceedings of the 2007 spring simulation multiconference – Volume 3*, SpringSim '07, (San Diego, CA, USA), pp. 225–232, Society for Computer Simulation International, 2007.

[31] U.S. Army PEO STRI, "OneSAF Public Site," April 2011. `http://www.peostri.army.mil/PRODUCTS/ONESAF/`.

[32] Epic Games, Inc., "Game Development Tools and Game Engine for Game Developers – Unreal Engine 4," accessed in December 2014. `http://www.unrealengine.com`.

[33] W. Broll, E. Meier, and T. Schardt, "The virtual round table – a collaborative augmented multi-user environment," in *Proceedings of the Third International Conference on Collaborative Virtual Environments*, CVE '00, (New York, NY, USA), pp. 39–45, ACM, 2000.

[34] C. Jeffery, W. Zhou, K. Templer, and M. Brazell, "A Lightweight Architecture for Program Execution Monitoring," *ACM SIGPLAN Notices*, vol. 33, pp. 67–74, July 1998.

[35] J. Urquiza-Fuentes and J. A. Velázquez-Iturbide, "A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems," *The ACM Transactions on Computing Education*, vol. 9, pp. 9:1–9:21, June 2009.

[36] B. Price, R. Baecker, and I. Small, "An Introduction to Software Visualization," *Software Visualization*, pp. 3–27, 1998.

[37] Passmark Software, "3D Benchmark – Video card speed test," accessed in December 2014. `http://www.passmark.com/products/pt_adv3d.htm`.

[38] N. Martinez, C. Jeffery, and J. A. Gharaibeh, "Unicon 3D Graphics User's Guide and Reference Manual," February 2014. `http://unicon.org/utr/utr9c.pdf`.

[39] B. Chazelle and D. P. Dobkin, "Optimal Convex Decompositions," *Computational Geometry*, pp. 63–133, 1985. `http://www.cs.princeton.edu/~chazelle/pubs/OptimalConvexDecomp.pdf`.

[40] E. Rufelt, "FW1FontWrapper – Home," October 2011. `http://fw1.codeplex.com`.

[41] Microsoft Corporation, "DirectX Tool Kit – Home," November 2014. http://directxtk.codeplex.com.

[42] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[43] F. Luna, *Introduction to 3D Game Programming with DirectX 11*. USA: Mercury Learning & Information, 2012. http://www.d3dcoder.net/d3d11.htm.

[44] FutureMark Corporation, "3DMark Cross-Platform Benchmark for Windows, Android and iOS," accessed in December 2014. http://www.geforce.com/drivers/results/79891.

[45] R. Griswold, "Icon Benchmarks," *The Icon Newsletter*, pp. 6–8, September 1989.

[46] S. Newton and C. Jeffery, "A Unicon Benchmark Suite," June 2014. http://unicon.org/utr/utr16.pdf.

[47] B. Fulgham and I. Gouy, "The Computer Language Benchmarks Game," accessed in December 2014. http://benchmarksgame.alioth.debian.org.

[48] M. Russinovich, "ClockRes v2.0," June 2009. http://technet.microsoft.com/en-us/sysinternals/bb897568.aspx.

[49] ASUSTeK Computer Inc., "GTX670-DC2T-2GD5 – Specifications," accessed in December 2014. http://www.asus.com/Graphics_Cards/GTX670DC2T2GD5/specifications/.

[50] NVIDIA Corporation, "GeForce – Drivers," November 2014. http://www.geforce.com/drivers/results/79891.

[51] NVIDIA Corporation, "GeForce – PhysX," accessed in December 2014. http://www.geforce.com/hardware/technology/physx.

Appendices

Appendix A: Concepts in Direct3D

This appendix is provided to bring the reader up to speed on the Direct3D rendering pipeline, shaders and transformation spaces. The reader is advised to review this chapter before proceeding to Chapter 4.

A.1 Introduction to DirectX

DirectX was first published in 1994 as a family of technologies that were designed to offer an affordable alternative to what were the de facto graphics facilities in 1994 (most notably OpenGL and the facilities in DOS). DirectX still exists today as a family of APIs for processing user input and generating audio and 2-D and 3-D image content that use the multimedia-oriented features on CPUs and GPUs to speed up their execution. Direct3D is a part of DirectX and defines a limited set of graphics instructions geared toward speeding up the development of multimedia applications for Windows operating systems. When it was introduced, DirectX required less extensive hardware support from graphics chipset manufacturers than was the case for OpenGL, which has traditionally targeted the professional graphics market with a complete graphics API that requires workstation-class hardware to run. Direct3D has since grown into the primary graphics instruction set used in video games for Windows computers, with nearly all mainstream[1] commercial video games relying on hardware support for DirectX version 9 or later to run.

Low-level graphics APIs such as Direct2D, Direct3D and OpenGL exist primarily to leverage GPUs' and general-purpose CPUs' ability to transform large pools of homogeneous data very quickly through parallelization. Consumer CPUs such as those by Intel and AMD support specialized SIMD instruction sets that use large on-processor SIMD registers that are well-suited to contain parts of a data stream, such as texture information in video games or parts of frames (still images) in a movie. GPUs are designed exclusively with this process in mind, and tend to contain many low-power processor cores with limited instruction sets and large pools of shared memory. As technology advances, this shared memory is placed in increasingly large quantities on the video card PCB, but it is accessible to the CPU through specific instructions in the chosen graphics API, although CPU access to video RAM incurs a significant performance penalty.

[1] The distinction between mainstream ("AAA") games and independent games is an arbitrary one. Generally speaking, mainstream game development is financially supported by a publisher while "indie" game development is not.

A.2 The Rendering Pipeline

A graphics rendering pipeline consists of a number of stages that each handle a part of the task of rendering 3-D geometric shapes to the screen. It is largely implemented in hardware, although certain stages are user-programmable.

The Direct3D 10 pipeline, which is wholly contained in the Direct3D 11 pipeline, has 8 input streams leading to video RAM. Each has 16 elements and each element has 1-4 data items in some canonical format (e.g., *float32*). The following pipeline stages operate on this stream of input data:

1. Vertex data is read from these streams by the *Input-Assembler stage*. Vertices can be indexed, which means additional buffers called index buffers may be used to calculate vertex offsets and to cache vertices by index. More importantly, the index buffers reduce the number of vertices to be stored in vertex buffers, since different indices may point to the same vertex (i.e., contain the same offset in a vertex buffer) because a point in 3-D space may be shared by multiple faces of a 3-D model. Blocks of vertices may be instanced (i.e., replicated) by the input assembler.

2. The *Vertex Shader stage* then accesses memory buffers to retrieve up to 128 textures and up to 16 constant buffers. Constant buffers are CPU-accessible registers in video memory that may be written to at any time. They usually contain transformation matrices that transform a model's vertices in a vertex buffer from one transformation space to another, usually from local space to projection space. Indices help the vertex shader choose the correct vertices for such transformations. Transformation spaces are explained farther into this chapter.

3. The *Geometry Shader stage* subsequently generates additional vertices based on input vertices. A good example is a sphere model, which should be perfectly round and cannot be accurately described with a finite set of vertices. To rectify this issue, the geometry shader stage may create additional vertices until the imperfections in the sphere's geometry are no longer discernible by the user. The geometry shader stage is not used in UniconD3D.

4. The *Stream-Output stage* streams (sequentially outputs) subsets of the incoming vertex information from the geometry shader into an input buffer, ideally (but unrealistically) at the same speed at which the input assembler stage reads its own streams.

5. The *Set-up and Rasterization stage* is a fixed pipeline stage that handles clipping, culling, viewport transformations and other operations that affect the output regardless of the involved geometry. The rasterization stage is commonly used to affect *how* a model is rendered, e.g., as a wireframe or inside-out.

6. The *Pixel Shader stage* samples textures in any of a multitude of ways to provide texture sampling, such as anti-aliasing. Because the pixel shader, like any shader, is a function written in the C-like language HLSL, it is used in UniconD3D to call functions that perform lighting calculations.

7. The Pixel Shader has access to an additional shader called the *Compute Shader*. It facilitates high-performance parallel calculations especially suited to compute more complicated lighting and physics effects. UniconD3D does not use the Compute shader.

8. The *Output-Merger stage* finally blends textures and stencils the textures, which means textures may be combined and the output may be cropped so as to avoid rendering to specific parts of the screen.

To the above rendering pipeline stages, Direct3D 11 adds three additional stages that were meant to reduce the amount of I/O between RAM and VRAM. These stages are left unused by UniconD3D and we therefore only explain them briefly.[2]

The *Hull-Shader stage*, *Tessellator stage* and *Domain-Shader stage* sit between the Vertex Shader and Geometry Shader stages, in that order. The Hull-Shader stage accepts vertex input from the geometry shader stage and adds information in the form of additional vertices (commonly referred to as *control points*) to groups of vertices called Bezier Patches. These control points may contain large numbers of vertices in the Bezier Patches they span. The Tessellator stage determines to which extent it should apply additional transformations on the control points, and subsequently the Hull-Shader stage determines to which extent the vertices in the control points' Bezier Patch should be transformed to cover the area outlined by the control points.

In modern video games, the above tessellation stages are commonly used to set the level of detail of a model depending on how close the model is to the camera. This can be achieved by setting the number of control points per Bezier Patch depending on how close the Bezier Patch is to the camera.

Computed data structures are stored in on-GPU RAM. These data structures are 1-D, 2-D or 3-D images as well as vertex and index buffers. Data structures may be passed between rendering passes to speed up execution. Single-pass rendering may be made more efficient by applying arrayed resources such as texture maps and render targets and having the shader compute the resources' array indices between passes. The geometry shader helps sort or replicate resources as array elements, and subsets of resource arrays may be stored as "views" to select subsets of arrayed resources. The geometry shader may also keep certain array resources untyped until the moment they are used in a rendering pipeline stage, as a limited form of type casting. Although resources cannot be both input and output, they may be bound to almost any of the pipeline stages to make resource reuse in iterative computation easier.

[2] A concise overview of the tessellation stages can be found here:

`http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx`

Starting with version 9, Direct3D offers a range of different data types that shaders operate on. All types are 32-bit, but smaller variables are usually packed to fit into one 32-bit variable.

Direct3D 10 and 11 emphasize feature set uniformity across all models of DirectX-compatible graphics cards by allowing variation only in a card's ability to sample 32-bit floating point textures, and support for different levels of multisample anti-aliasing. There has also been a shift toward removal of all user-programmable constructs from the core Direct3D API, such as lighting and fog. Another shift was to make the pipeline as general (read: programmable) as possible, which lead to programmable vertex, geometry and pixel shader stages in Direct3D 10, and the aforementioned programmable tessellation stages in Direct3D 11. Although Microsoft considered making the input assembler user-programmable to allow the programmer to specify their own vertex layouts, this was decided against because the same results could be achieved through the vertex shader. The geometry shader consists of multiple processing units, but because the geometry shader preserves the order of operations, the output of any processing units working in parallel must be buffered. This means a compromise had to be struck between parallelism and hardware cost. As a result, the increasingly intensive use of the geometry shader stage in video games that used Direct3D 10 eventually led to the inclusion of the tessellation shader stages in Direct3D 11.

A.3 Shaders

Until a programmable rendering pipeline was introduced in 2001, with the advent of vertex and pixel shaders, no stages of the pipeline could be altered at runtime. In this context, "programmable" means changeable through the use of custom shaders. The focus of graphics hardware design has been on applications in digital entertainment, where graphics hardware manufacturers and graphics API designers attempt to achieve two perpendicular goals: offering agile rendering control while providing the capability to process large amounts of data. Dealing with the surrounding issues has always required a high level of cooperation between graphics API designers and graphics hardware manufacturers such as AMD, Nvidia and Matrox. The graphics architecture limitations that DirectX 10 was designed to ameliorate are:

- The high overhead incurred with state changes of any kind, for instance, loading in any number of new resources for a geometric shape to be drawn.
- The large amount of variation in processing capabilities of graphics cards: the feature set supported in hardware by the graphics card was typically a subset of the features described in DirectX.
- The slow synchronization between CPU and GPU.
- Data type limitations: shaders could not use all data types, such as integers, and data type requirements for shaders are weakly specified.

- The limited resources for shaders, which has led to research on splitting complicated shaders into relatively simple shaders that run in multiple passes.

In versions of Direct3D prior to Direct3D 10, the vertex shader and pixel shader were the only two programmable rendering pipeline stages. These work in tandem and process model geometry and pixel effects, respectively. There are more pixel shaders than vertex shaders because there are more pixels than geometry to process. Direct3D's and OpenGL's shaders are stored as bytecode that is interpreted in a runtime environment on the GPU. They abstract away the differences in implementations of the pipeline in different vendors' graphics cards. Although shaders could be programmed without an intermediary language to give the programmer more direct access to the graphics card's hardware capabilities, the High-Level Shader Language (HLSL) was introduced by Microsoft as a C-like language to increase programmer productivity. It can be compiled just-in-time (JIT)[3] while OpenGL's shaders are completely precompiled. The CPU directs both the rendering pipeline and shading programs, and provides shaders with constant values and textures in on-chip registers.

A.4 Initializing Direct3D

All graphics engines built around Direct3D follow roughly the same initialization steps to make hardware resources available to the programmer. Although an overview of the Direct3D initialization process is included here to help clarify the layout of UniconD3D in Chapter 4 and the integration process in Chapter 5, it is kept concise lest the author repeat existing literature on the Direct3D API.

A.4.1 Initializing the Device and Device Context

D3D provides an abstraction of the concept of a graphics adapter by providing a software construct of its own, a *device*. This device contains an abstraction of the graphics adapter itself (useful for choosing a graphics adapter in case multiple are available to D3D), which type of graphics driver to use (i.e., whether or not to emulate D3D statements in software, and if so, which software rasterizer to use), which DirectX feature levels are supported by the graphics card and which device context is used. These aforementioned feature levels describe which versions of DirectX are supported by the graphics card; UniconD3D uses these to ascertain if it can be used on a given computer.

The aforementioned *device context* is a construct used to configure the rendering pipeline, also called the *swap chain*. Aside from creating vertex and index buffers, the device context also contains functions to create textures from images and to draw (parts of) the geometry of a scene to the screen. The resulting output – pixel colors in a two-dimensional window – undergoes two operations before it is drawn to an off-screen buffer called the *back buffer*:

[3] With the introduction of DirectX 11.1, the programmer may opt to precompile HLSL shaders instead.

- Each pixel has associated with it a depth, defined as the distance from the camera to the pixel's position in 3-D space. This depth is compared to the depth value that belongs to the pixel element at the corresponding 2-D coordinates in the back buffer. If the new pixel's depth is smaller than that of the existing pixel in the back buffer, the new pixel replaces the existing one in the back buffer. This process is called a *depth test*. The depth information per pixel in the back buffer is stored in a separate buffer called the *depth buffer*. The depth test can be changed or disabled with the device context. This is useful when you want an object A to be drawn as if it were located in front of another object B from the camera's perspective, even though A is located behind B.

- After all pixels are drawn to the back buffer, the pointers to the front and back buffers are switched so that the former back buffer is now the front buffer, and the front buffer's contents are drawn to the screen.[4]

As alluded to above, the device context initialization includes initializing a back buffer, depth buffer and a *viewport*. The viewport is the area on the screen that is used to render output to.

A.4.2 Initializing the Pipeline

After the device, device context and miscellaneous buffers are initialized, all (pixel, vertex, geometry, etc.) shaders are compiled and stored in *blobs*: buffers that contain compiled bytecode. Since shaders consist of an arbitrary number of C functions, the runtime environment is given pointers to the shaders' starting addresses in their respective blobs. These pointers are created by the device, but the device context determines which shaders are being used. This gives the programmer the freedom to implement, say, multiple lighting models and choose a lighting model depending on the amount of available VRAM.

A.5 Transformation Spaces

As explained in Chapter 4, the geometry of 3-D models in Unicon consists of groups of points in 3-D space called vertices. These vertices are usually defined relative to the vertices in the same model only, and not relative to the vertices of another model. The X, Y and Z coordinates of a model's vertices can be transformed with transformation matrices that represent an amount of translation (movement), rotation and scaling along the three axes. Such transformation matrices can be used to position models relative to one another, rotate models and scale models. To make discussion of transformation matrices more uniform, different *transformation spaces* are defined:

- Local space (or model space): the transformation space that places all vertices of a 3-D model relative to each other. Models created by modeling programs such as Maya or 3D Studio Max can be exported in local space.

[4]This assumes there is only one back buffer; some engines use multiple back buffers. There is always one front buffer.

- World space: the transformation space where models are positioned relative to each other. The world space coordinates of a model are obtained by transforming a model's vertices with the model's translation, scaling and rotation matrices.

- View space: the transformation space that transforms vertices from world space so they are shown from the camera's perspective. The view space transformation matrix is defined as a matrix that contains the camera's position, a vector of the camera's right direction and a vector with the camera's up direction.

- Projection space (commonly called *homogeneous clip space*): the transformation space that translates the models' vertices from view space into two-dimensional pixel coordinates that are rendered to the output window. Just like the viewable area of a real-life camera, the projection area looks like a pyramid that is placed on its side: it enlarges objects close to the camera and makes far away objects smaller. It is defined as a combination of a near plane, a far plane, the field of view (FOV) – i.e., the angle of the "top" of the pyramid, which determines the width of the projection plane – and the aspect ratio. The process of reversely transforming 2-D screen coordinates into view space, world space or local space coordinates in order to find out which model occupies a given screen coordinate is called *picking*.

Appendix B: UniconD3D Overview

This appendix contains UML 2.0 diagrams to clarify the discussion of the UniconD3D implementation in Chapter 4. Figure B.1 illustrates the relationships between all classes that generate an process geometry: the `Primitive`, `Model`, `Shape`, `Area` and `ModelFactory` classes.

Additionally, Figure B.2 illustrates the relationships between the other classes: `Graphics`, `Controls`, `Config`, `Timer` and `Camera`. Most notably, the Graphics class receives commands from the code in UniconD3D's "main" file, which contains the WinMain function. This main file serves as a proxy through which Unicon-level commands are accepted by UniconD3D, and is provided as a C-callable wrapper interface to calling code such as the Unicon runtime system.

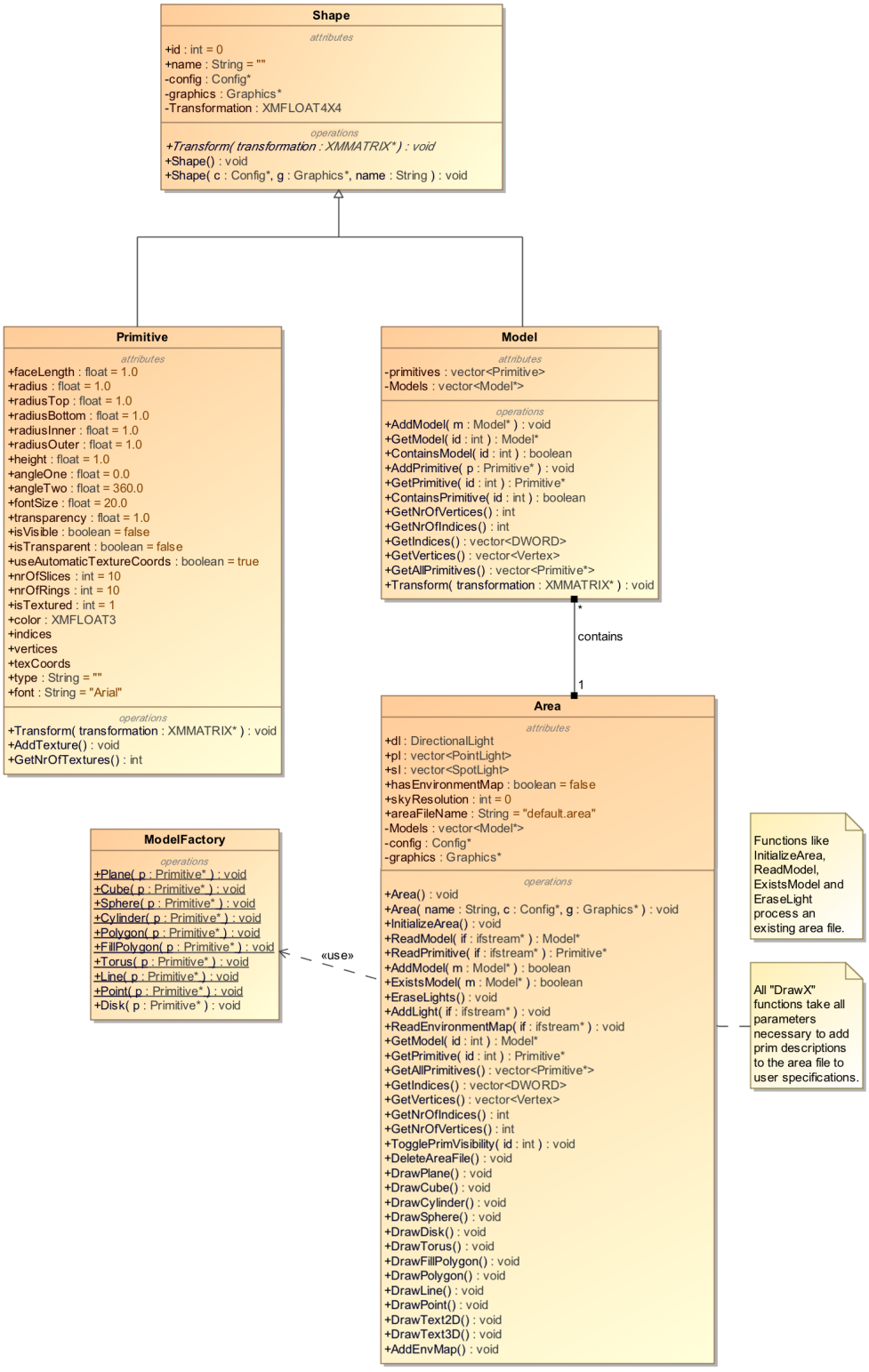


Figure B.1: Class diagram illustrating relations among models and primitives.

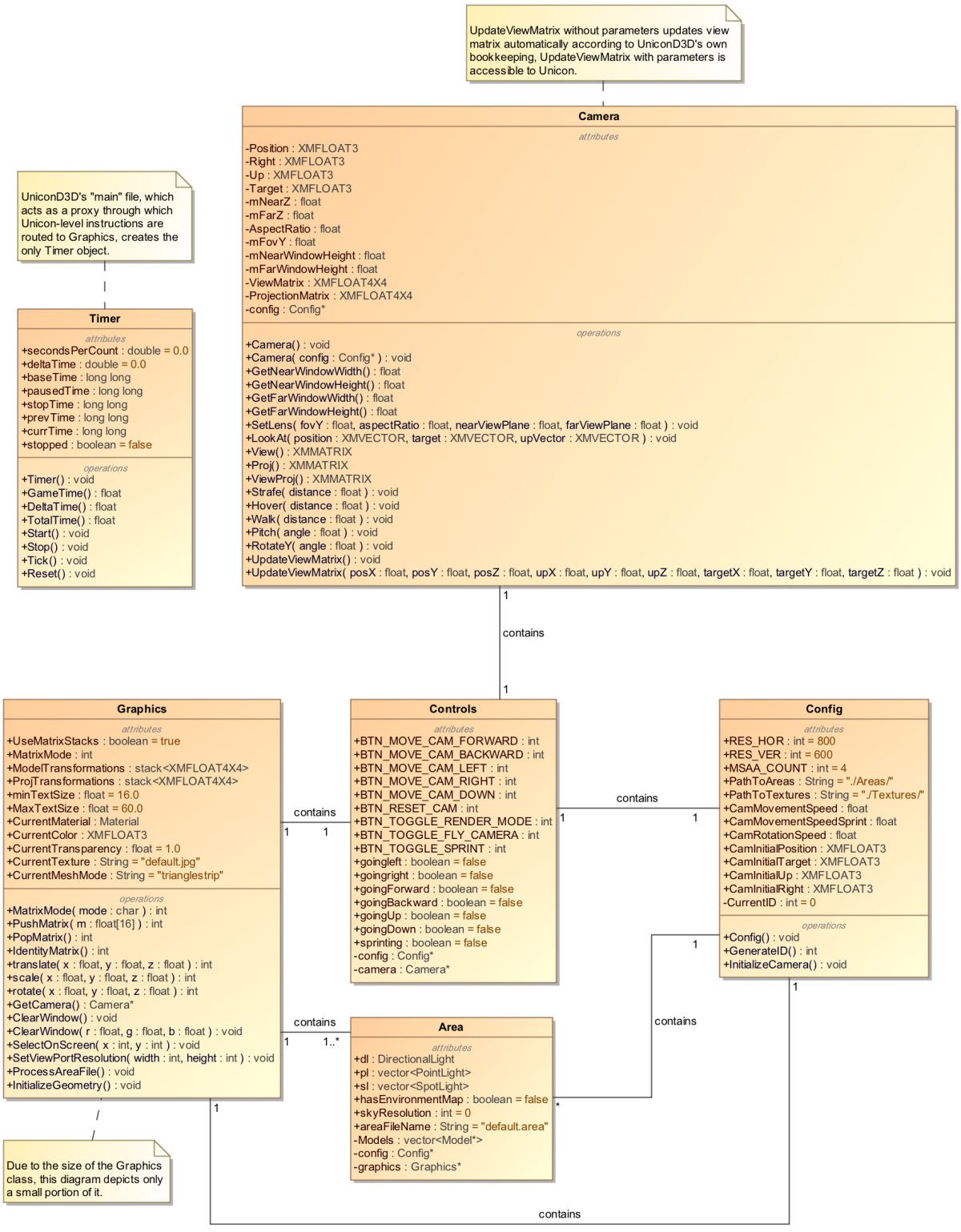


Figure B.2: Class diagram illustrating relations among camera and control classes.

Appendix C: Copyright Notices

DirectX ToolKit

The source code in this project uses the DirectX ToolKit (DXTK) as an external library. It can be found here: `https://directxtk.codeplex.com`. The DirectX ToolKit is governed by the Microsoft Public License, which is described here: `https://directxtk.codeplex.com/license`. It allows others to publish and distribute any part of the DirectX ToolKit with the understanding that it is provided as-is.

FW1FontWrapper

The source code in this project uses the FW1FontWrapper as an external library. It can be found here: `http://fw1.codeplex.com`. The FW1FontWrapper is governed by the MIT license, which is described here: `http://fw1.codeplex.com/license`. It allows others to publish and distribute any part of the FW1FontWrapper with the understanding that it is provided as-is.

Appendix D: UniconD3D Function Conversion Table

This appendix contains a function list derived from the interface rd3d.ri, the interface in Unicon's runtime system that connects UniconD3DLib to Unicon's high-level graphics instruction set.

rd3d.ri

A list of interfaces that are called by the Unicon runtime system to translate Unicon-level 3D graphics commands through UniconD3DLib. Functions that are not a priority for the completion of this thesis, are printed in terracotta text.

Initialization and window functions

Unicon function name	UnionD3D function name	Explanation	Implementation comments
void init_3dcanvas(wbp w)	void init_3dcanvas()	Initialize the 3D subsystem for rendering, including allocating any resources necessary to use 3D on w's canvas.	
int release_3d_resources(wbp w)	int release_3d_resources()	Free/release 3D system resources.	
int create3dcontext(wbp w)		Allocate resources needed for a 3D graphics context.	
int destroycontext(wbp w)		Free 3D resources associated with w's context.	
void makecurrent(wbp w)	-	Make window w's 3D display the one currently used by 3D subsystem calls. At least on OpenGL this is needed, since OpenGL calls do not include a parameter for what window they operate on.	Apparently used by many other function in ropengl.ri as a helper function to set the current window context.
int copytextures(wcp wc1, wcp wc2))		Copy the texture resources allocated in wc2 into newly created context wc1.	
int redraw3D(wbp w)	int redraw3d()	Redraws a 3D window by traversing its display list.	
void initializeviewport(int w, int h)	void initializeviewport(int w, int h)	Initialize the viewport to a given width and height.	The viewport is D3D's internal back buffer resolution, not the width and height of the output window!
void swapbuffers()	void swapbuffers()	Make rendered output visible on-screen.	this function was made C-callable, and RenderFrame(double) was changed so it doesn't call swapchain->Present(0, 0) automatically anymore.
void erasetocolor(float r, float g, float b)	void erasetocolor(float r, float g, float b)	Erase/set the 3D window to RGB color	
void erasetocolor()	void erasetocolor()	Erase/Set the 3D window to current color	

int setselectionmode(wbp w, char* s)	int setselectionmode(char* s)	Set 3D selection mode on or off. s equals "on" or "off". 3D selection mode determines whether input mouse clicks are mapped onto the object(s) in the scene that were clicked, denoted by identifiers set using WSection().	
	void selectonscreen(int x, int y);	Depending on the selection mode, either (a) selects the coordinates in the output window, or (b) selects an on-screen primitive.	The arguments x and y to this function are obtained in WindowProc, which is located in the calling C code. The selected coordinates or Primitive* are stored globally in Graphics.h as variable "selectedPrimitive". It is set with each picking operation.

Drawing shapes

Unicon function name	**UnionD3D function name**	**Explanation**	**Implementation comments**
void cube(double length, double x, double y, double z, int gen)	void cube(double length, double x, double y, double z, int gen)	Draw a cube on the current 3D window centered at (x,y,z). Flag gen==1 means: use the currently set texture. gen==0 uses materials.	
void cylinder(double radius1, double radius2, double height, double x, double y, double z, int slices, int rings, int gen)	void cylinder(double radius1, double radius2, double height, double x, double y, double z, int slices, int rings, int gen)	Draw a cylinder. Default points "up". gen==1 means use texture. Slices and rings govern how smooth/rounded the approximation should be.	
void disk(double radius1, double radius2, double angle1, double angle2, double x, double y, double z, int slices, int rings, int gen)	void disk(double radius1, double radius2, double angle1, double angle2, double x, double y, double z, int slices, int rings, int gen)	Draw a disk. radius1!=radius2 specifies an ellipse. Angle1 and angle2 units are degrees. Angle2 $<$ 360 degrees specifies a partial disk. slices and rings govern how smooth/rounded the approximation should be. gen==1 means use texture.	The number of horizontal, parallel rings is fixed to 2 in UniconD3D.
void sphere(double radius, double x, double y, double z, int slices, int rings, int gen)	void sphere(double radius, double x, double y, double z, int slices, int rings, int gen)	Draw a sphere. Slices and rings govern how smooth/rounded the approximation should be. gen==1 means use texture.	

void torus(double radius1, double radius2, double x,double y, double z, int slices, int rings, int gen)	void torus(double radius1, double radius2, double x,double y, double z, int slices, int rings, int gen)	Draw a torus. Slices and rings govern how smooth/rounded the approximation should be. gen==1 means use texture.	
void drawpoly(float *vertices, double x, double y, double z, int gen)	void drawpoly(float *vertices, double x, double y, double z, int gen)	Draw unfilled polygon using the current mesh mode, texture mode, etc.	Polygons are drawn using a mesh mode that simulates triangle fans.
void drawfilledpoly(float *vertices, double x, double y, double z, int gen)	void drawfilledpoly(float *vertices, double x, double y, double z, int gen)	Draw filled polygon using the current mesh mode, texture mode, etc.	Filled polygons are drawn using a mesh mode that simulates triangle fans.
	int drawstrng2d(char *s, double x, double y, double fontsize, char *font)	Draw text string s at location (x,y) using the current color	
int drawstrng3d(wbp w, double x, double y, double z, char *s)	int drawstrng3d(char *s, double x, double y, double z, double fontsize, char *font)	Draw text string s at location (x,y,z) using the current color	
int setmeshmode(wbp w, char* s)	int setmeshmode(char *s)	Set vertex interpretation for Polygon operations to one of: "points", "lines", "linestrip", "lineloop", "triangles", "trianglefan", "trianglestrip", "quads", "quadstrip", "polygon"	Of these, "points", "lines", "linestrip", "triangles" and "trianglestrip" are supported in UniconD3D.
int setlinewidth3D(wbp w, LONG linewid)	-	Set the line width for 3D lines.	Impossible in D3D unless you create a quad for each line and change the quad's thickness.

Matrix operations

Unicon function name	**UnionD3D function name**	**Explanation**	**Implementation comments**
int popmatrix()	int popmatrix()	Pop a matrix from either the projection or the modelview matrix stack	Both transformation matrix stacks always contain the identity matrix, so it can never be empty.
int pushmatrix()	int pushmatrix(float, float, float, float, float, float, float, float, float, float, float, float, float, float, float, float)	Push a matrix onto the current stack -- that is either the modelview or projection matrix stack	
int identitymatrix()	int identitymatrix()	Set the top of the current matrix stack to use the identity matrix.	
int setmatrixmode(char *s)	int setmatrixmode(char *s)	The matrix mode controls whether operations (e.g. push and pop) affect the "modelview" or the "projection" stack.	
int rotate(wbp w, dptr argv, int i, dptr f)	int rotate(float x, float y, float z)	Apply a rotate transformation to the top of the current matrix stack.	Both transformation matrix stacks always contain the identity matrix, so it can never

			be empty.
int scale(wbp w, dptr argv, int i, dptr f)	int scale(float x, float y, float z)	Apply a scale transformation to the top of the current matrix stack.	Both transformation matrix stacks always contain the identity matrix, so it can never be empty.
int translate(wbp w, dptr argv, int i, dptr f)	int translate(float x, float y, float z)	Apply a translate transformation to the top of the current matrix stack.	Both transformation matrix stacks always contain the identity matrix, so it can never be empty.
void applymatrix(wbp w, double a[])	void applymatrix(double a[])	Multiply the 4x4 matrix given in array-of-16-doubles a onto the top of the current matrix stack.	Both transformation matrix stacks always contain the identity matrix, so it can never be empty.

Material and texture operations

Unicon function name	**UnionD3D function name**	**Explanation**	**Implementation comments**
int settexcoords(wbp w, char* s)	int settexcoords(char *s)	Set the texture coordinates to be used for supplied vertices from alternating u,v pairs within a comma-separated string of numbers s.	
	void setcolor(double r, double g, double b)	Sets the color of vertices in subsequent colors to given RGB value.	
int texwindow2D(wbp w, wbp w2d)	int texwindow2D(wbp w1, wbp w2)	Set the current texture from the contents of a 2D window.	
int texwindow3D(wbp w1, wbp w2)	int texwindow3D(wbp w1, wbp w2)	Set the current texture from the contents of a 3D window.	
int settexture(wbp w, char* str, int len, struct descrip *f, int curtex, int is_init)	void settexture(char *texturename)	Set the current texture. w = 3D window on which we operate, str = filename or Unicon image string, len = length of str, *f = output param for display list entry, curtex = -1: new, >=0: replace current texture @ curtex, is_init = 0: new display list entry. 1: write on existing.	It looks like set texture will load an image with a given name from the drive and create a texture from it, with the option of setting it as the current texture, while bindtexture just sets a pre-existing texture as the current texture. In this case, they both do the same thing in UniconD3D, because textures are always created if they do not yet exist.

int setautogen(int i)	int setautogen(int i)	Turn automatic texture coordinate generation on or off.	Affects the use of automatically generated tex coords for all primitive types, including cubes, spheres and filled polygons.
int getmaterials(char* buf)	int getmaterials(char *buf)	Returns the current material properties	
int getlight(int light, char* buf)	int getlight(int light, char* buf)	Returns the current value of the specified light # in the result buf	
-	void addlight(float ambX, float ambY, float ambZ, float difX, float difY, float difZ, float difW, float specX, float specY, float specZ, float specW, float dirX, float dirY, float dirZ, float posX, float posY, float posZ, float attX, float attY, float attZ, float range, float spot)		Add a point light to the scene.
int determinematerial(char *temp, C_integer r, C_integer g, C_integer b, C_integer a)		Traverse the given list and set material properties. RGBA are given in X11-style 0-65535 range.	
int make_enough_texture_space (wcp wc)	-	Allocate space for textures.	
int TexDrawLine(wbp w, int texhandle, int x1, int y1, int x2, int y2)	-	Draw a 2D line directly on some texture memory. Probably can be made portable or adapted from ropengl.ri.	
int TexDrawRect(wbp w, int texhandle, int x, int y, int width, int height)	-	Draw a 2D rectangle directly on some texture memory. Probably can be made portable or adapted from ropengl.ri.	
int TexFillRect(wbp w, int texhandle, int x, int y, int width, int height, int isfg)	-	Draw a 2D filled rectangle directly on some texture memory. Probably can be made portable or adapted from ropengl.ri.	
int TexDrawPoint(wbp w, int texhandle, int x, int y)	-	Draw a 2D point directly on some texture memory. Probably can be made portable or adapted from ropengl.ri.	
int TexReadImage(wbp w, int texhandle, int x, int y,struct imgdata *imd)	-	Read an image into a texture.	
int TexCopyArea(wbp w, wbp w2, int texhandle, int x, int y, int width, int height, int xt, int yt, int width2, int height2)	-	Copy an area (from a window) directly into some texture memory.	

int setmaterials(wbp w, char* s)		Set the current material surface with which polgyons will be rendered when in non-textured or blended mode. Ignored in regular texture mode.	
void apply_texmodechange(wbp w)	void apply_texmodechange(bool useTextures)	Call the 3D system to turn the texture mode on or off. The texture mode has been set in w->context->texmode.	
void bindtexture(wbp w, texture t)	void settexture(char *texturename)	Apply/use a texture t as the current texture on subsequent 3D primitive(s)	See int settexture(wbp w, char* str, int len, struct descrip *f, int curtex, int is_init)
void applyAutomaticTextureCoords()	void applyAutomaticTextureCoords (int i)	Enable/use automatic texture coordinates for standard 3D primitives	Sets texture coordinates to be used for all subsequent primitives that are not filled polygons. See int setautogen(int i)
	void settransparency(float t)		Assigns transparency factor f to subsequent primitives.

Appendix E: Benchmark Code

This appendix contains the source code for the two benchmarks described in Chapter 6.

Benchmark 1

```
/*
 * Author: Fabian Mathijssen
 * Date: December 3rd 2014
 */
#include <Windows.h>
#include <windowsx.h>
#include "./../../UniconD3D (working copy)/UniconD3D/Main.h"
#pragma comment (lib, "Main.lib")

#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

HINSTANCE hInstanceCopy;
int nCmdShowCopy;
HWND hWnd;

int minCubes = 0;
int maxCubes = 85000;
int isTransparent = 0;

__int64 prevTime = 0,
        currTime = 0;
double  deltaTime = -1.0;

LRESULT CALLBACK WindowProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    if (msg == WM_LBUTTONDOWN)
        selectonscreen(GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));

    else if (msg == WM_KEYDOWN) {
        if (wParam == VK_ESCAPE) {
            DestroyWindow(hWnd);
            return 0;
        }
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}

void CreateD3DWindow(int resHor, int resVer, LPCSTR windowname) {
    WNDCLASSEX wc;
    ZeroMemory(&wc, sizeof(WNDCLASSEX));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstanceCopy;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.lpszClassName = "WindowClass";
    RegisterClassEx(&wc);

    RECT wr = { 0, 0, resHor, resVer };
    AdjustWindowRect(&wr, WS_OVERLAPPEDWINDOW, FALSE);
```

```
    hWnd = CreateWindowEx(
        WS_EX_LEFT, //default, left-aligned window
        "WindowClass",
        windowname,
        WS_OVERLAPPEDWINDOW,
        0,
        0,
        wr.right - wr.left,
        wr.bottom - wr.top,
        NULL,
        NULL,
        hInstanceCopy,
        NULL);
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow) {
    for (int i = minCubes; i < maxCubes; i+= 5000) {
        FILE *fp;
        if ((fp = fopen("Benchmark1.txt", "a")) == NULL) {
            printf("can't open file\n");
            return 0;
        }

        //Initialize high-performance timer
        __int64 countsPerSecond;
        QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSecond);
        double secondsPerCount = 1.0 / (double)countsPerSecond;

        hInstanceCopy = hInstance;
        nCmdShowCopy = nCmdShow;

        //Create window
        char *windowname = "Test";
        CreateD3DWindow(1800, 1000, windowname);

        //Open window
        ShowWindow(hWnd, nCmdShowCopy);

        //Create Graphics, Config and Controls objects before adding primitives
        UniconD3DMain(hWnd);

        //Set viewport independently from window resolution.
        initializeviewport(1800, 1000);

        //Initialize Direct3D
        UniconD3DInitGraphics(hWnd);

        //set color, set transparency, bind texture
        D3Dsettexture("crate.jpg");
        D3Dsetcolor(1, 0, 0);

        //set default transformations
        D3Dmatrixmode("modelview");
        identitymatrix();
        D3Dmatrixmode("projection");
        identitymatrix();

        //Set camera location
        SetCamera(0, 0, -10, 0, 1, 0, 0, 0, 1);

        //Init current time
        __int64 temp;
```

```
        QueryPerformanceCounter((LARGE_INTEGER*)&temp);
        currTime = temp;
        deltaTime = (currTime - prevTime) * secondsPerCount;
        prevTime = currTime;

        for (int k = 0; k < i; k++) {
            isTransparent = (isTransparent + 1) % 2;
            if (!isTransparent)
                settransparency(1.0f);
            else
                settransparency(0.5f);
            cube(1.0, rand()%1000 - 500, rand()%1000 - 500, rand()%1000 - 500, 1);
        }

        for (int j = 0; j < 60; j++) {
            //Refresh window
            D3Derasetocolor(1.0f, 1.0f, 1.0f);
            D3Dredraw3D();
            D3Dswapbuffers();

            //Measure time
                //Get current time
                __int64 temp;
                QueryPerformanceCounter((LARGE_INTEGER*)&temp);
                currTime = temp;

                //Calculate time since previous frame
                deltaTime = (currTime - prevTime) * secondsPerCount;
                prevTime = currTime;

                //Make deltaTime nonnegative
                deltaTime = (deltaTime < 0.0) ? 0.0 : deltaTime;

                //Print deltaTime
                fprintf(fp, "%d, ", i);
                fprintf(fp, "%f\n", deltaTime);
        }

        fclose(fp);
        UniconD3DCleanGraphics();
        CloseWindow(hWnd);
    }
    return 0;
}
```

Benchmark 2

```
/*
 * Author: Fabian Mathijssen
 * Date: December 3rd 2014
 */
#include <Windows.h>
#include <windowsx.h>
#include "./../../UniconD3D (working copy)/UniconD3D/Main.h"
#pragma comment (lib, "Main.lib")

#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

HINSTANCE hInstanceCopy;
int nCmdShowCopy;
HWND hWnd;

int minCubes = 0;
int maxCubes = 15000;
int isTransparent = 0;
int nrOfLights = 0; //nr of lights in environment, changed between benchmark runs

__int64 prevTime = 0,
        currTime = 0;
double  deltaTime = -1.0;

LRESULT CALLBACK WindowProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    if (msg == WM_LBUTTONDOWN)
        selectonscreen(GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));

    else if (msg == WM_KEYDOWN) {
        if (wParam == VK_ESCAPE) {
            DestroyWindow(hWnd);
            return 0;
        }
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}

void CreateD3DWindow(int resHor, int resVer, LPCSTR windowname) {
    WNDCLASSEX wc;
    ZeroMemory(&wc, sizeof(WNDCLASSEX));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstanceCopy;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.lpszClassName = "WindowClass";
    RegisterClassEx(&wc);

    RECT wr = { 0, 0, resHor, resVer };
    AdjustWindowRect(&wr, WS_OVERLAPPEDWINDOW, FALSE);

    hWnd = CreateWindowEx(
        WS_EX_LEFT, //default, left-aligned window
        "WindowClass",
        windowname,
        WS_OVERLAPPEDWINDOW,
        0,
        0,
        r.right - wr.left,
        wr.bottom - wr.top,
```

```
        NULL,
        NULL,
        hInstanceCopy,
        NULL);
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow) {
    for (int i = minCubes; i <= maxCubes; i += 1000) {
        FILE *fp0 = fopen("Benchmark0.txt", "a");
        FILE *fp1 = fopen("Benchmark1.txt", "a");
        FILE *fp2 = fopen("Benchmark2.txt", "a");
        FILE *fp3 = fopen("Benchmark3.txt", "a");
        FILE *fp4 = fopen("Benchmark4.txt", "a");
        FILE *fp5 = fopen("Benchmark5.txt", "a");
        FILE *fp6 = fopen("Benchmark6.txt", "a");
        FILE *fp7 = fopen("Benchmark7.txt", "a");
        FILE *fp8 = fopen("Benchmark8.txt", "a");

        //Initialize high-performance timer
        __int64 countsPerSecond;
        QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSecond);
        double secondsPerCount = 1.0 / (double)countsPerSecond;

        hInstanceCopy = hInstance;
        nCmdShowCopy = nCmdShow;

        //Create window
        char *windowname = "Test";
        CreateD3DWindow(1800, 1000, windowname);

        //Do not open window
        ShowWindow(hWnd, nCmdShowCopy);

        //Create Graphics, Config and Controls objects before filling area with prims
        UniconD3DMain(hWnd);

        //Set viewport independently from window resolution.
        initializeviewport(1800, 1000);

        //Initialize Direct3D
        UniconD3DInitGraphics(hWnd);

        D3Dsettexture("crate.jpg");
        D3Dsetcolor(1, 0, 0);

        //set default transformations
        D3Dmatrixmode("modelview");
        identitymatrix();
        D3Dmatrixmode("projection");
        identitymatrix();

        //Set camera location
        SetCamera(0, 0, -10, 0, 1, 0, 0, 0, 1);

        for (int k = 0; k < i; k++) {
            isTransparent = (isTransparent + 1) % 2;
            if (!isTransparent)
                settransparency(1.0f);
            else
                settransparency(0.5f);
            cube(1.0, 0, 0, 0, 1);
        }
```

```
//Add lights
for (int l = 0; l < nrOfLights; l++) {
    AddLight(8,
             1, 1, 1,            //amb
             1, 1, 1, 1,         //diff
             1, 1, 1, 1,         //spec
             0, -1, 0,           //dir
             0, 20, 0,           //pos
             0.1f, 0.1f, 0.1f,   //att
             1000.0f,            //range
             0.0f);              //spot
}

//Init current time
__int64 temp;
QueryPerformanceCounter((LARGE_INTEGER*)&temp);
currTime = temp;
deltaTime = (currTime - prevTime) * secondsPerCount;
prevTime = currTime;

for (int j = 0; j < 30; j++) {
    //Refresh window
    D3Derasetocolor(1.0f, 1.0f, 1.0f);
    D3Dredraw3D();
    D3Dswapbuffers();

    //Measure time
        //Get current time
        QueryPerformanceCounter((LARGE_INTEGER*)&temp);
        currTime = temp;

        //Calculate time since previous frame
        deltaTime = (currTime - prevTime) * secondsPerCount;
        prevTime = currTime;

        //Make deltaTime nonnegative
        deltaTime = (deltaTime < 0.0) ? 0.0 : deltaTime;

    //Print deltaTime
    if (nrOfLights == 0) {
        fprintf(fp0, "%d, ", i);
        fprintf(fp0, "%f\n", deltaTime);
    } else if (nrOfLights == 1) {
        fprintf(fp1, "%d, ", i);
        fprintf(fp1, "%f\n", deltaTime);
    } else if (nrOfLights == 2) {
        fprintf(fp2, "%d, ", i);
        fprintf(fp2, "%f\n", deltaTime);
    } else if (nrOfLights == 3) {
        fprintf(fp3, "%d, ", i);
        fprintf(fp3, "%f\n", deltaTime);
    } else if (nrOfLights == 4) {
        fprintf(fp4, "%d, ", i);
        fprintf(fp4, "%f\n", deltaTime);
    } else if (nrOfLights == 5) {
        fprintf(fp5, "%d, ", i);
        fprintf(fp5, "%f\n", deltaTime);
    } else if (nrOfLights == 6) {
        fprintf(fp6, "%d, ", i);
        fprintf(fp6, "%f\n", deltaTime);
    } else if (nrOfLights == 7) {
        fprintf(fp7, "%d, ", i);
        fprintf(fp7, "%f\n", deltaTime);
```

```
            } else if (nrOfLights == 8) {
                fprintf(fp8, "%d, ", i);
                fprintf(fp8, "%f\n", deltaTime);
            }
        }

        if      (nrOfLights == 0) fclose(fp0);
        else if (nrOfLights == 1) fclose(fp1);
        else if (nrOfLights == 2) fclose(fp2);
        else if (nrOfLights == 3) fclose(fp3);
        else if (nrOfLights == 4) fclose(fp4);
        else if (nrOfLights == 5) fclose(fp5);
        else if (nrOfLights == 6) fclose(fp6);
        else if (nrOfLights == 7) fclose(fp7);
        else if (nrOfLights == 8) fclose(fp8);
        UniconD3DCleanGraphics();
        CloseWindow(hWnd);
    }
    return 0;
}
```