Computational Optimizations for Real-Time 3D Rendering

A collection of suggested techniques for high-performance, hardware-accelerated 3D computer graphics

Antonios Gogios

M. IEEE Computer Society

M. ACM SIGGRAPH

Supervised by:

Dr. Konstantinos Veropoulos

A thesis submitted to Cardiff Metropolitan University
in partial fulfilment of the requirements for the degree of

Master of Science in Computing

2017

Contents

Abstract **5**

1 Introduction **6**
1.1 Topic description 6
1.2 Structure of this document 7
1.3 Previous work 7
1.3.1 Literature review 7
1.3.2 Author's work 26
1.4 Aims & objectives 28
1.4.1 Problem definitions 28
1.4.2 Research aims 28

2 Methodology **30**
2.1 Performance requirements 30
2.1.1 Framerate 30
2.1.2 Response time 31
2.2 Test scripts 32
2.2.1 Test data set 32
2.2.2 Test cases 34
2.3 Hardware 37
2.4 Testing environment 38
2.5 Testing procedure 39
2.6 Initial system configuration 40
2.6.1 The generic algorithm A1 40
2.7 Performance bottlenecks 41
2.8 Tested optimizations 42
2.8.1 The optimized algorithm A2 43
2.9 Conducted measurements 44
2.10 Evaluation process 47

3 Analysis **49**
3.1 Collective vertex/index buffers 49
3.2 Frequency-of-update constant buffers 52
3.3 Front-to-back rendering 55
3.4 Instancing 58
3.5 Frame limiter 62

3.6 Pre-transformed immobile objects 69

4 Conclusions **75**

5 Further research **79**

Bibliography **83**

APPENDIX **92**

A Table of measurements **93**

B Real-time rendering: Fundamental technical terms and concepts **95**

B.1 Renderer 95
B.2 GPU and real-time rendering 95
B.3 Polygons and polygon count 96
B.4 Frame rate and Frames per second (FPS) 98
B.5 Main loop structure of an interactive real-time rendering application . 99
B.6 Textures 99
B.7 GPU Buffers and draw calls 101
B.8 Shader programs 102
B.9 Pixel fragments 102
B.10 Depth culling 103
B.11 Overdraw 103
B.12 Shader complexity 104
B.13 Coordinate systems and spaces 106

List of Figures **108**

List of Tables **112**

Abstract

The aim of this study is to determine, test and present a set of suggested performance optimizations for hardware-accelerated, real-time 3D rendering systems. The study focuses on efficient GPU resource management and primarily addresses systems pertaining to 3D CAD software, interactive simulation and video game development. An exclusive GPU memory environment is also assumed, which is by far the most common at the time of writing. After a thorough series of comparative tests, the efficacy of each suggested technique is quantitatively measured and demonstrated. By collectively applying all suggested techniques, the tests performed indicate a **55.04%** increase in rendering speed as opposed to the initial, non-optimized configuration. System responsiveness is also increased by **9637** times. Detailed efficacy analysis for each individual technique allows readers to make informed decisions and prioritize their choices if considering to adopt any presented techniques in their own systems.

1. Introduction

1.1 Topic description

This study concerns the subject of determining computational optimizations that can be applied to the process of real-time, 3D (3-dimensional) rendering, which is a sub-field within the greater field of computer graphics research. Rendering is a computationally intensive process, something that is particularly true in the case of *3D* rendering, where 3 spatial dimensions of data are involved. This inherent computational complexity makes it a prime target for efforts to optimize it while retaining as much visual quality as possible, and has also led to techniques utilizing *dedicated* hardware (Graphics Processing Units or GPUs) to process such data with exponentially higher speed than conventional CPUs (Central Processing Units).

Such additional hardware components necessarily bring with them an increased overall *system complexity* and inter-component communicational overhead. Hence, to make the most of the system's capabilities, a rendering software developer must be able to address that increased complexity, and manage the system's resources with a good knowledge of the underlying system architecture and its distinctive characteristics and behaviour.

The focus of this study is on such *hardware-accelerated* 3D rendering, its optimal resource management, and on making the best possible use of such a system's configuration for achieving optimal rendering speed. It also adds the performance constraint of the rendering being *real-time* and thus necessitates that no less than 24 frames (see Möller, Haines and Hoffman, 2008; Poynton, 1996; also see Appendix, B.4) are rendered per second, imposing on the rendering software an elapsed time with an upper limit of 0.0416 seconds for rendering a single frame (ibid.).

Finally, since real-time 3D rendering is a fairly extensive field with many areas of distinct application and requirements, this study focuses on *interactive* real-time 3D rendering, and it is considered that the optimizations discussed more directly apply to the context of 3D CAD software, interactive simulation and video game development. Attention is given to considering as widely applicable optimization scenarios as possible, in the hope that the research's contribution will be relevant and possibly useful to an equally wide audience of real-time renderer developers.

1.2 Structure of this document

The rest of this introduction contains an overview of the field literature so far, and notes important milestones in computational techniques as well as in the software and hardware involved. A brief background on related previous personal work conducted by the author is also provided, followed by a delineation of the aims of this research.

The research methodology is thoroughly described is section 2, followed by section 3 which contains an in-depth analysis of all findings and observations. In section 4, the research conclusions are summarized, while section 5 suggests a number of areas of interest for possible further research on the selected topic.

The appendices have two main sections: Appendix A contains a full table with all the measurements for all test cases discussed throughout the study, while Appendix B contains descriptions of fundamental technical terms pertaining to real-time rendering that are utilized throughout this document. Unless the reader is already familiar with this subject, it is recommended to read Appendix B first in order to familiarize oneself with the terminology and concepts involved.

1.3 Previous work

1.3.1 Literature review

Early work

Early publications of relevance in the field include the work of Watkins (1970), which discusses the technique of scanline rendering, of Gouraud (1971), which introduces the Gouraud surface-shading model, and of Newell, Newell and Sancha (1972) which constitutes an early approach aimed at resolving the issue of 3D hidden surface removal.

Phong's 1975 PhD thesis (Phong, 1975) has been a very influential contribution in 3D computer graphics research, in which Phong proposes a 3D surface-shading model constituting an attempt to approximate the physical behaviour of light to a degree that, although simplified computationally (to make it more feasible for

use in 3D rendering), still maintains an amount of apparent photorealism that is superior to other contemporary models (Fig. 1.1). His shading model soon became a basis for subsequent research in 3D shading and rendering, and eventually proved particularly important for real-time rendering uses. His thesis also led to several other important publications that followed soon after.

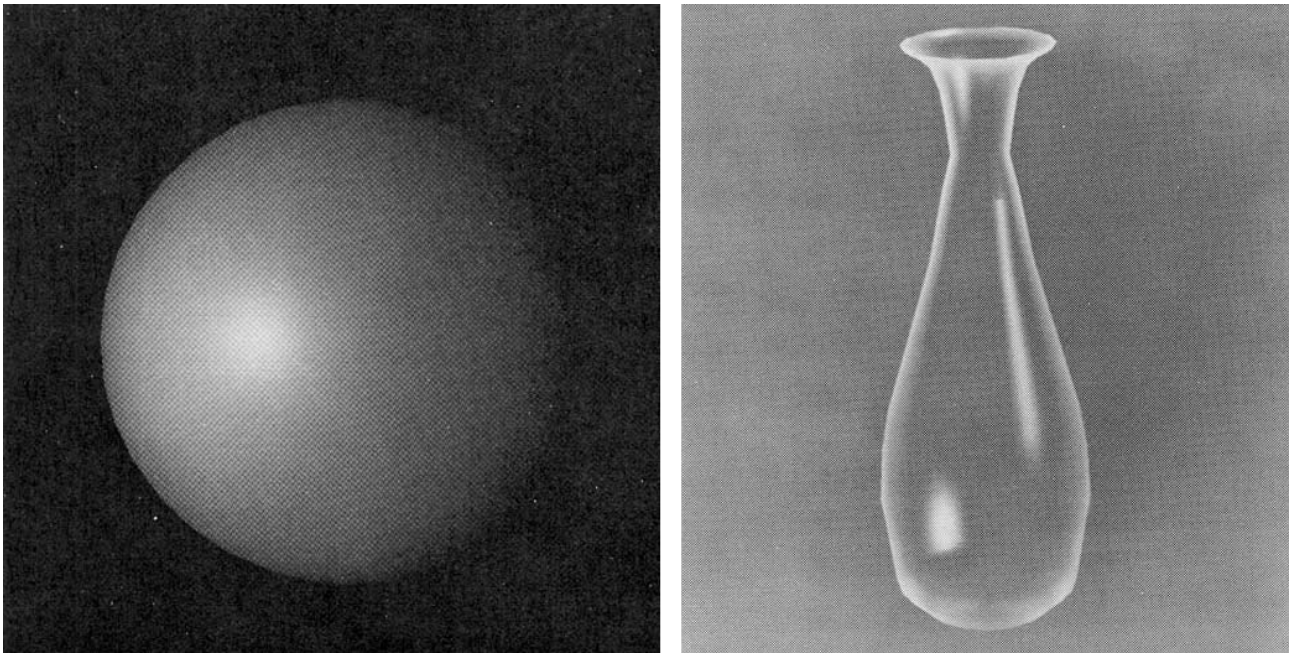

Figure 1.1: Rendering results using Phong's shading model (Phong, 1975).

Two years later, Blinn (1977) made a significant contribution to the field by suggesting his modified version of Phong's shading model. This model, usually referred to as the Blinn-Phong shading model or the "modified Phong" shading model, offered some additional computational efficiency compared to Phong's original model, while essentially retaining most of the (apparent) photorealism that Phong's produced. This model also became quite significant in the years that followed, and was eventually the default shading model used by the fixed-function pipeline of the real-time graphics APIs OpenGL and Direct3D, up to OpenGL 3.1 and DX10, respectively (Van Oosten, 2014).

Blinn made other important contributions in early 3D computer graphics research, such as refinements in the process of texturing (Blinn and Newell, 1976) – which was a technique established originally by the work of Catmull (1974) – as well as introducing the techniques of environment mapping (Blinn and Newell, 1976; see Fig. 1.2) and bump mapping (Blinn, 1978; see Fig. 1.3). While working for NASA's JPL (Jet Propulsion Laboratory), Blinn also participated in the creation of a number of 3D animations for JPL's Voyager program (see Blinn and Kohlhase,

1978 and Fig. 1.3).

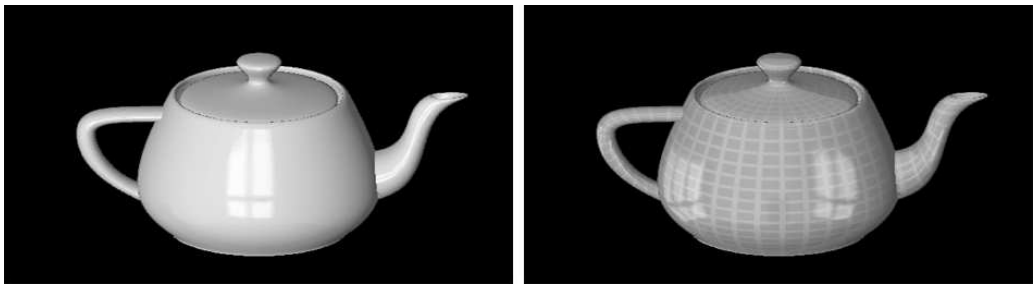

Figure 1.2: Images from Blinn and Newell's 1976 paper, which introduced the technique of environment mapping (Blinn and Newell, 1976).

Figure 1.3: Left: Image from Blinn's 1978 paper, which introduced the technique of bump mapping (Blinn, 1978). Right: Screenshot from a 3D animation by Blinn and Kohlhase which JPL released the same year (Blinn and Kohlhase, 1978).

In the same year, Williams introduced shadow mapping (Williams, 1978). Two years later Whitted introduced the significant technique of ray tracing (Whitted, 1980; see Fig. 1.4), with Appel having already laid a foundation for it by introducing ray casting (Appel, 1968). Both of these techniques – particularly ray tracing – offered increased apparent photorealism although involved a considerably higher computational cost.

Several of the techniques discussed so far had far-reaching effects and applications, but were not directly applicable in a real-time context upon their original publication. Due to the optical phenomenon of the persistence of vision (Möller, Haines and Hoffman, 2008; Poynton, 1996), for real-time rendering to occur it is a requirement that a minimum of 24 consecutive images (or frames) are rendered per second. If this is not maintained, the human eye no longer sees continuous motion, but a series of static images. Hence, real-time rendering requires a minimum framerate (see Appendix B.4) of 24 frames/sec to occur, which was not computationally viable by the hardware capabilities of the time. These techniques were used with non-real-time framerates initially, although as we will see, when the hardware

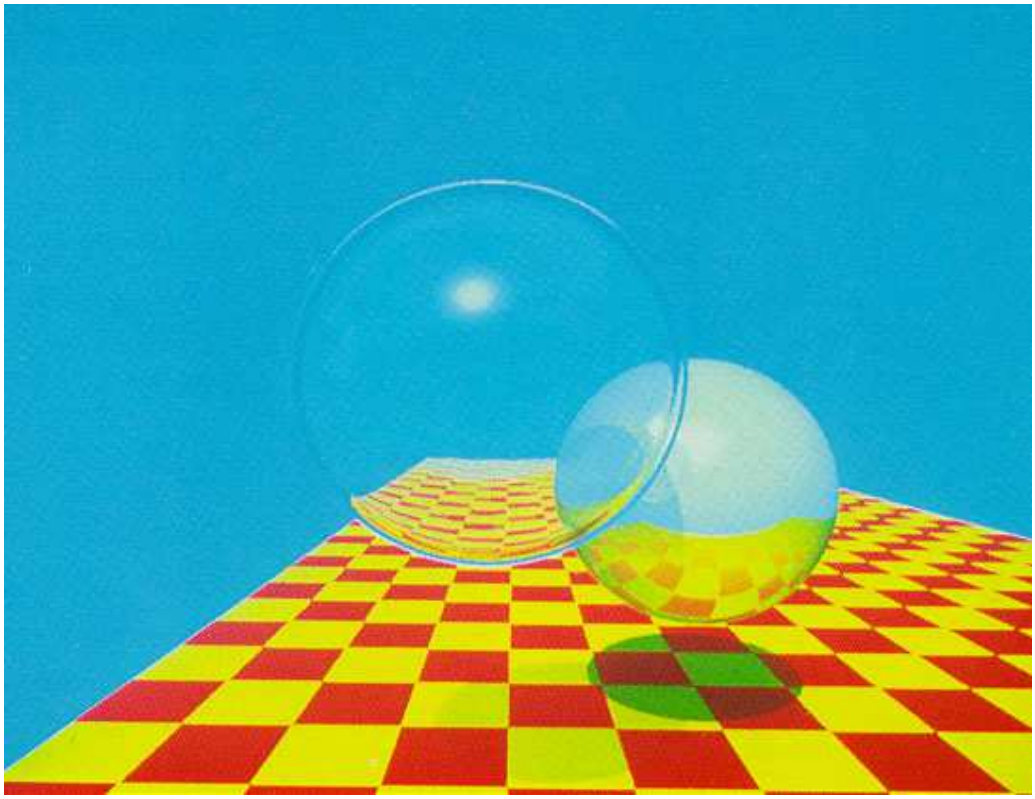

Figure 1.4: Image from Whitted's 1980 paper which introduced ray tracing (Whitted, 1980).

eventually allowed it, they became the basis for real-time 3D rendering techniques that followed.

Real-time implementations and further advancements

The period between 1970 and 2010 involves much work that resulted in the real-time implementation of techniques and models previously discussed, and which led to further advancements both in the software and hardware involved. To illustrate this, a number of chronologically-ordered real-time rendering applications are mentioned as indicative examples of their contemporary systems' rendering capabilities. The choice of examples is based on the applications' capacity to be computationally demanding for their respective rendering systems, and primarily involves 3D video gaming and CAD software.

An example of early "nearly-real-time" 3D rendering is the video game "Spasim" (Bowery, 1974; see Fig. 1.5), which included *wireframe* 3D content only (i.e. composed of points and lines, but no solid surfaces), due to the hardware limitations of the time. Another example from about 10 years later, which achieved even higher frame rate but still not actually a minimum of 24 frames/sec is the "Freescape"

game engine (Incentive Software, 1987), which became popular for featuring 3D *solid* surfaces (Fig. 1.6). It was used in a number of video games developed by the same company, and eventually led to the 3D CAD software "3D Construction Kit" (Incentive Software, 1991; see Fig.1.7), which allowed the user to generate 3D content and use Freescape's game engine features to also add custom gameplay and game-related rules to the content.

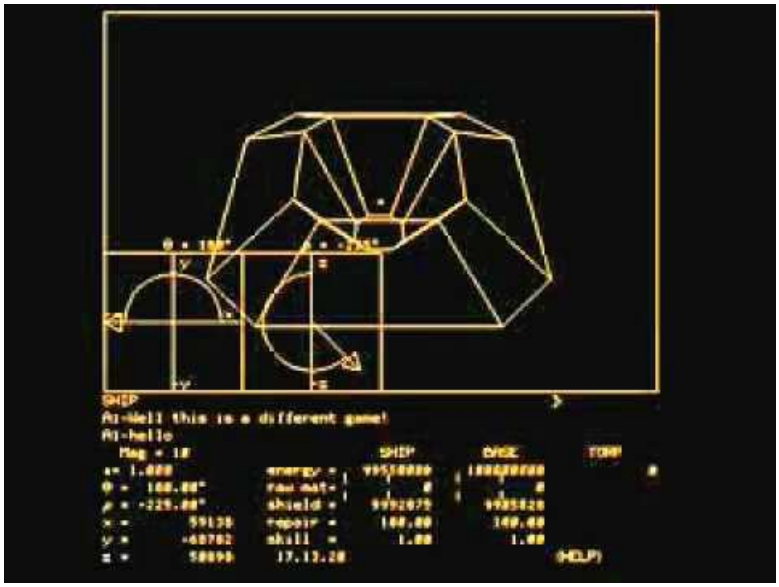

Figure 1.5: "Spasim", an example of early "nearly-real-time" 3D rendering, featured the wireframe technique (Bowery, 1974).

Figure 1.6: The Freescape 3D engine featured solid 3D surfaces (Incentive Software, 1987; Teque Software Development, 1990).

As hardware capabilities increased, achieving a real-time frame rate for non-trivial 3D content became possible. Polygon count (see Appendix B.3) increased (Fig. 1.8, 1.10 and 1.11), and eventually, the allowable shading calculations' complexity started increasing too, giving rise to various techniques for better manipulating and making the most of the available potential of the hardware.

Phong-derivative shading models were the norm in all cases involving real-time 3D rendering until the late 2000s, because of the model's shading computational ef-

Figure 1.7: The 3D CAD software and game engine 3D Construction Kit (Incentive Software, 1991).

Figure 1.8: "Quake II: The Reckoning" (Xatrix Entertainment Inc., 1998) (left) and "Unreal" (Epic MegaGames, Digital Extremes and Legend Entertainment, 1998) (right), competitors at the time. Compare with the current state of Unreal Engine, shown in Fig. 1.16.

ficiency. Various additions were made over the years, becoming industry-standard techniques that were expected to be addressed in any shading model utilized by a real-time renderer, such as texturing, dynamic lights and shadows, normal and specular texture maps, ambient occlusion, and several others. Shading approach variations such as *deferred* shading appeared in use (Hargreaves and Harris, 2004; Fernando and Pharr, 2005; also see Fig. 1.9) – which utilized ideas originally introduced in Deering et al (1988) and Saito and Takahashi (1990). Furthermore, progress was steadily made in real-time ray-tracing implementations (Harris, 2008; see Fig. 1.12).

Figure 1.9: Forward (left) and deferred (right) shading comparison for the same scene (Fernando and Pharr, 2005).

Video games like "Crysis" (Crytek, 2007; also see Fig. 1.13) – which was commended for its real-time graphics' quality at the time (Adams, 2007) – and "Call of Duty: Modern Warfare 3" (Infinity Ward and Sledgehammer Games, 2011; see Fig. 1.13) still essentially used Phong-derivative shading models. During this period however, a new type of shading model gained prominence, and soon became an industry-standard for real-time: the Physically Based Rendering model, or PBR.

The PBR shading model

As mentioned earlier, the capabilities of hardware increased dramatically over time, allowing for more real-time computational complexity. This naturally led to more extensive shading models – and gradually – models that no longer assumed Phong-like foundations, but rather based their calculations a lot more directly on the *physical* behaviour of light, because they were now allowed to by contemporary hardware. These are all models that employ ray tracing (Pharr and Humphreys, 2010). Due to the models' basis on physics, that family of models gradually be-

Figure 1.10: Wireframe and solid views of a character from two installments of the video game series "Tomb Raider" (Core Design Ltd., 1996; Crystal Dynamics, 2007), showing the differences in polygon count, texture resolution, and shading model.

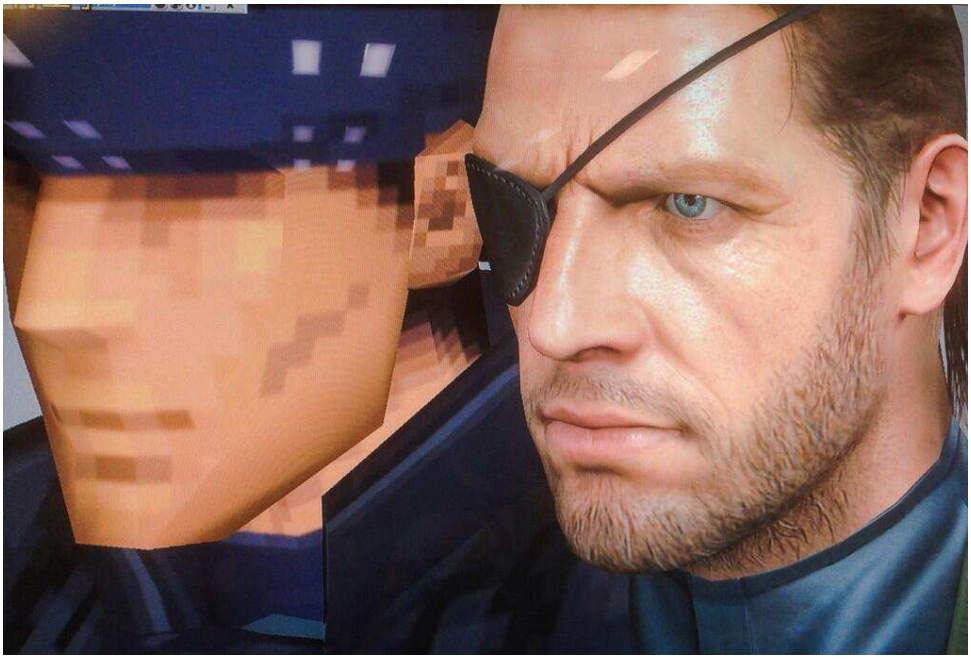

Figure 1.11: Comparison of a character from two installments of the video game series "Metal Gear" (Konami Computer Entertainment Japan, 1998; Kojima Productions, 2014).

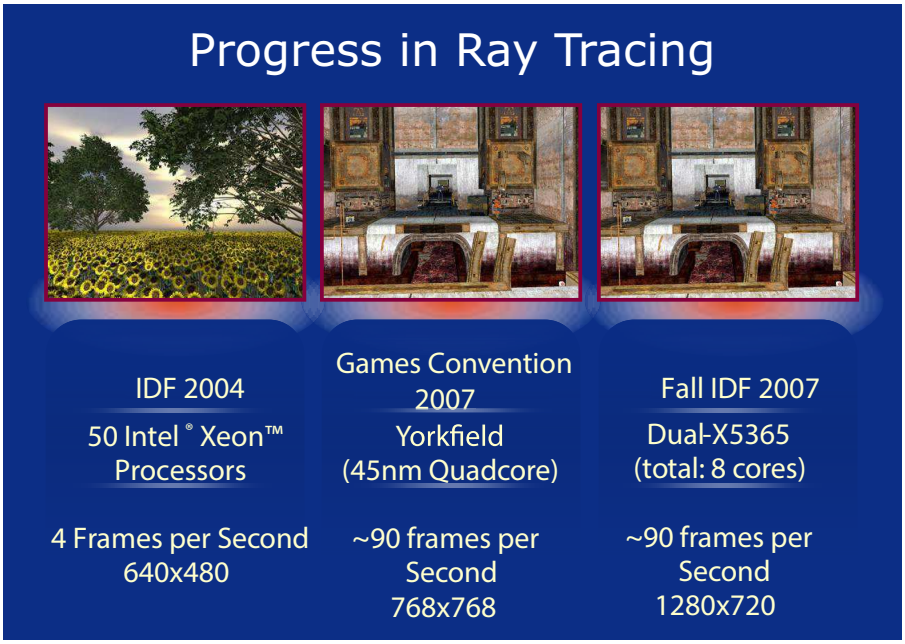

Figure 1.12: Comparative framerates, resolutions and types/numbers of processors used in ray tracing benchmark scenes between 2004 and 2007, indicating the increasing feasibility of ray tracing for real-time use (Harris, 2008). By 2007, ray tracing achieved 90 frames per second for this test scene, rendering at a 1280 x 720 resolution.

Figure 1.13: Crysis (Crytek, 2007) (left), and Call of Duty: Modern Warfare 3 (Infinity Ward and Sledgehammer Games, 2011) (right).

came known as "Physically-Based" (ibid.). Initially, such models were only available in CPU-rendering implementations (and thus not real-time), and were used for producing high-quality, photorealistic images that might take several hours to compute. An example of such a CPU-based PBR renderer is "pbrt" (ibid.; see Fig. 1.14).

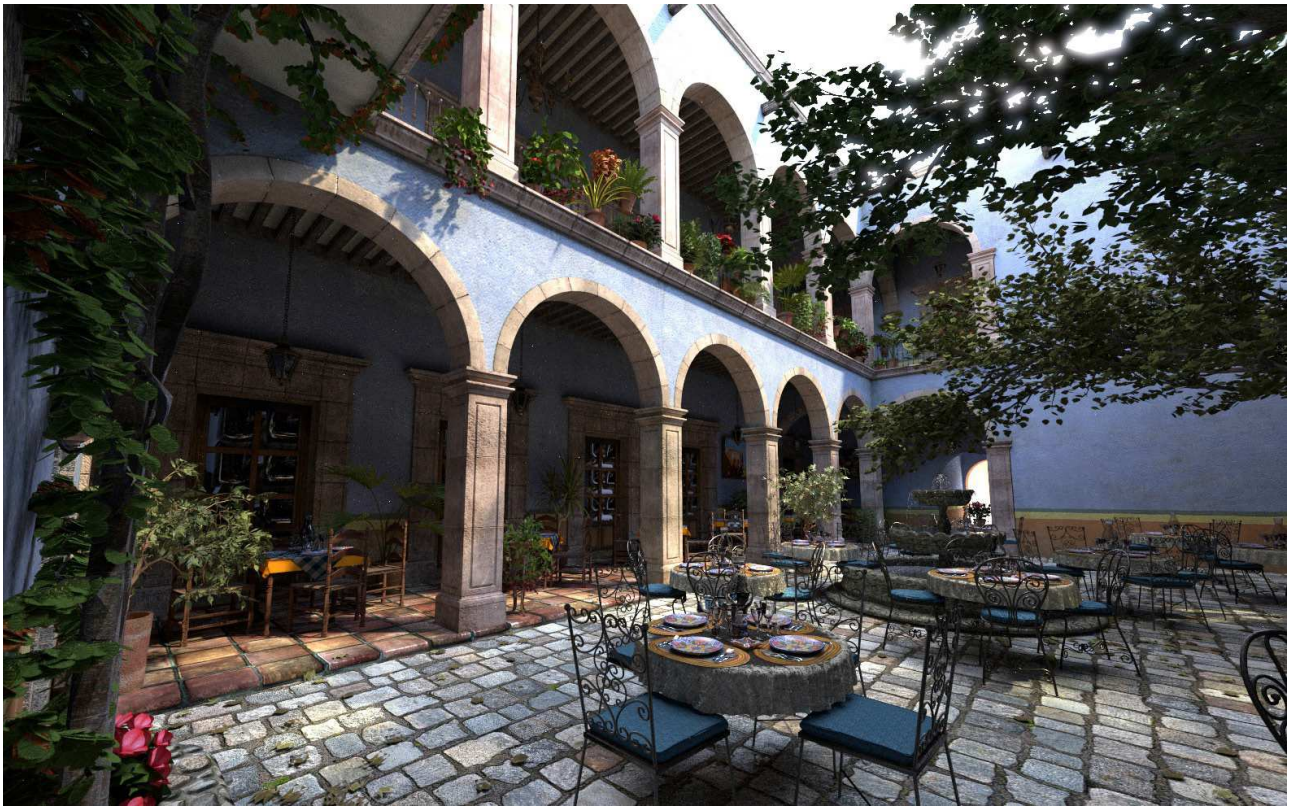

Figure 1.14: A CPU-rendered frame by the open-source PBR renderer "pbrt" (Pharr and Humphreys, 2010).

In recent years, hardware capabilities made rendering with PBR models possible in real-time (two examples are shown in Figures 1.15 and 1.16). PBR has thus fast become the new standard shading model for photorealistic real-time rendering.

Reviewing main changes in real-time rendering techniques

Having briefly covered a period of about 5 decades of 3D and real-time rendering, by now it can be observed that the allowable amount of pixels that are processable per frame, along with the permissible per-frame shading calculations, have increased substantially. The same applies for the number of polygons that can be rendered in real-time for a given frame (see Fig. 1.10). This is primarily due to utilizing more powerful hardware, and the result is the exponentially increased scene complexity that can be observed in the previously shown figures.

Figure 1.15: A real-time rendered PBR frame from the video game "Uncharted 4" (Naughty Dog, 2016).

In terms of actual shading computational approaches, the main difference is the transition from the simpler Phong-derivative models (mostly dividing light components to "ambient", "diffuse" and "specular" terms) to a fully supported real-time PBR model. While the Phong-derivative models tend to rely on mostly arbitrary and non-standardized values and formulas, with the main aim to reach *apparent* photorealism, PBR models follow more physically accurate formulas, strictly enforce energy conservation, involve several additional material attributes, and also rely more on systematic generation of standardized reference tables for those values (Fig. 1.17).

In terms of implementing these models and designing software that effectively manages system resources, the utilized techniques necessarily vary, depending on the underlying hardware and its configuration. Hardware acceleration has been used from very early on, as will be discussed below. However, the main issue in harnessing hardware acceleration is always to perform the best possible resource management, since dedicated hardware brings with it the overhead of module interfacing, data copying, or – in the case of shared memory models – the overhead and complexity of synchronizing and securing that shared access.

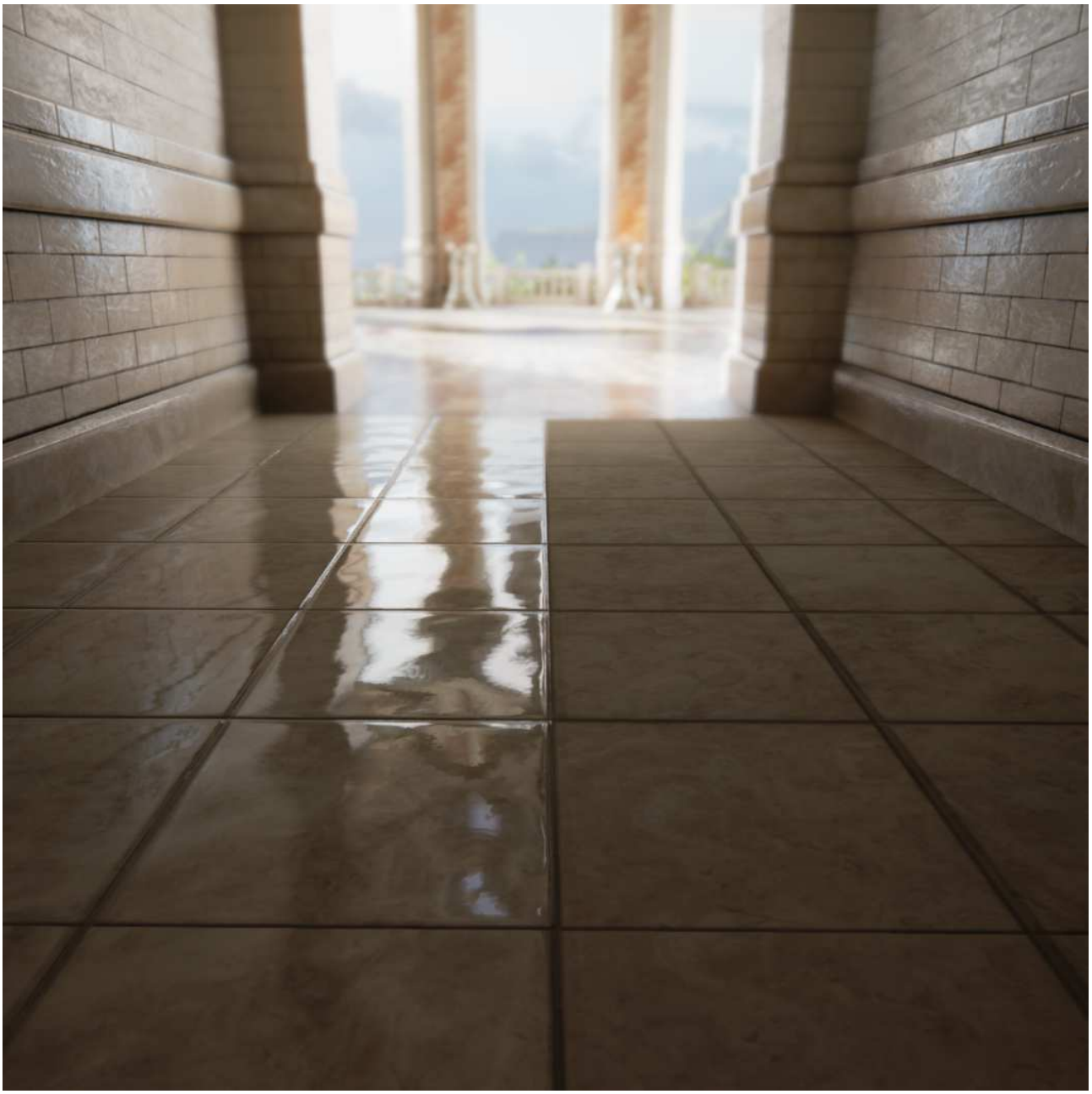

Figure 1.16: A screenshot from a 3D scene in Unreal Engine 4's real-time PBR-shaded environment (Walker, 2014).

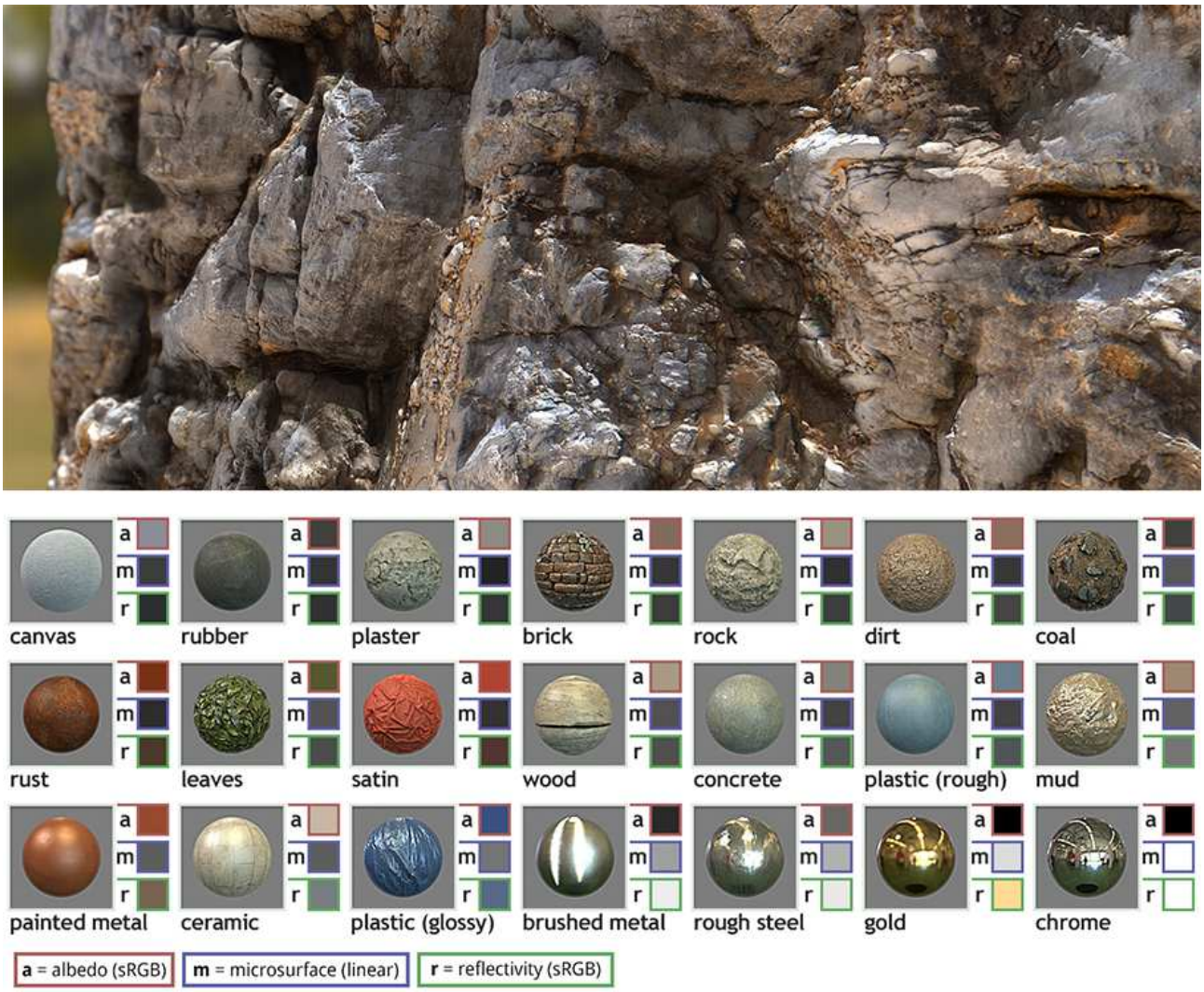

Figure 1.17: Top: Real-time PBR render performed by the material-authoring software "Substance Designer" (Allegorithmic, 2017). Bottom: Example of 3D-scanned materials with standardized PBR value tables (Wilson, 2015).

Recent developments and publications

A number of fairly popular (Schauerte, 2014) annual international conferences, which focus on computer graphics research and publish a large number of papers on the subject, are the ACM Special Interest Group on Computer Graphics and Interactive Techniques (ACM SIGGRAPH, 2017), the Symposium on Interactive 3D Graphics and Games (I3D, 2017) and IEEE's Visualization and Computer Graphics conference (IEEE, 2017). The transactions of these conferences thus serve as a primary resource elucidating recent developments in the field.

Another noteworthy set of publications, specifically focusing on real-time 3D rendering, is the GPU Gems series (Fernando, 2004; Fernando and Pharr, 2005; Nguyen,

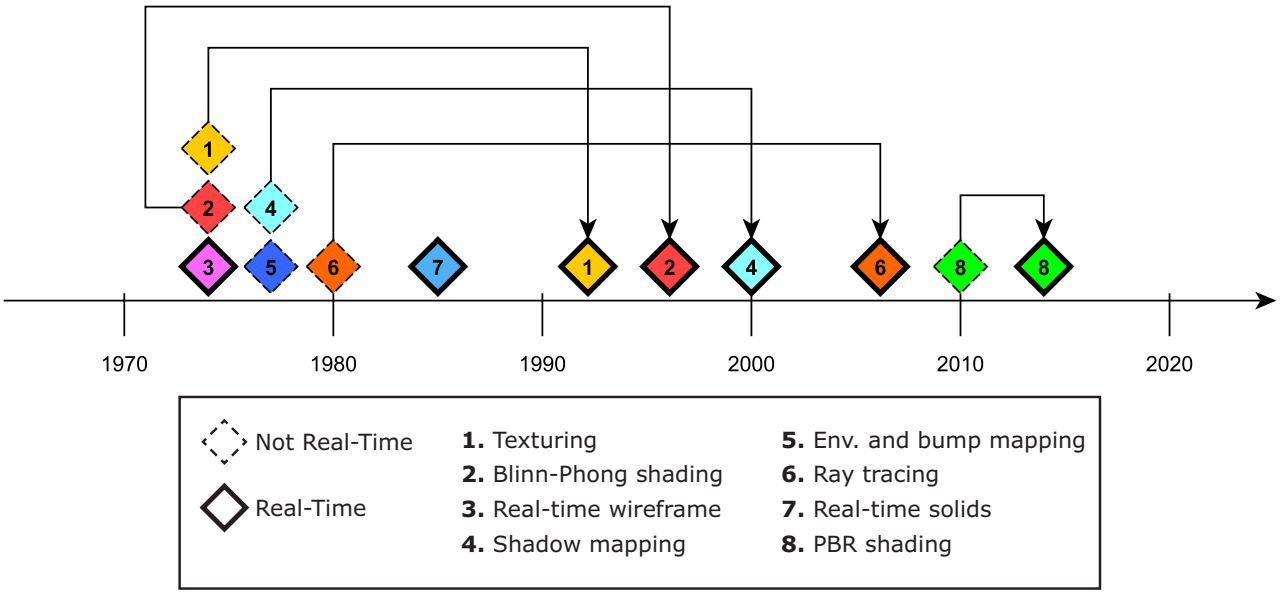

Figure 1.18: A timeline summarizing major milestones of the field.

2008). This is a collection of white papers and articles on real-time 3D rendering which focuses on discussing high-performance real-time software architecture, 3D transformation and shading algorithms, as well as overall GPU resource and memory management. It also covers effectively using the GPU to perform General Purpose Computing tasks.

Although not strictly constituting an academic resource/publication, one other important set of publications worth mentioning are those by the international Game Developers' Conference (GDC, 2017), which is widely considered the largest conference focusing exclusively on video game development worldwide (Nvidia Corporation, 2017c; Advanced Micro Devices, Inc., 2017b; Khronos Group, 2017b). The conference features speakers from various real-time 3D graphics development backgrounds, often under the auspices of a group or company, and tends to focus on discussing instructive case studies of recent projects, highlighting development challenges met and choices made, or discuss emerging workflows and contemporary good development practices.

A review of graphics processing hardware

In terms of hardware, and specifically the availability and use of dedicated hardware for graphics computation (i.e. GPUs), these have been utilized since the 1970s by different vendors and in various vendor-specific configurations (Bauer, 2015). They were initially used for the faster production of 2D images, and later also for 3D

(ibid.). Their growing capacity to efficiently process greater amounts of data made it possible for more complex shading models and higher amounts of polygons to be utilized.

The earliest dedicated hardware for processing graphics occurred on arcade video game systems of the 1970s, with Fujitsu's MB14241 (Fig. 1.19) perhaps being the first dedicated "GPU" (ibid.).

Figure 1.19: The MB14241 was used to accelerate 2D sprite drawing and supported RGB colour. It was used in video games such as Gun Fight (Taito, 1975), and Space Invaders (Taito, 1978).

In PCs, no dedicated hardware was used for graphics until the 1980s, with all graphics computations performed in software by the CPU. One of the first dedicated hardware units for processing graphics for PCs was the NEC uPD7220 High-Performance Display Adapter (Fig. 1.20), released in 1981 for NEC's APC personal computers (Stengel, 2016; Bauer, 2015).

Figure 1.20: The NEC uPD7220 was used for drawing 2D points, lines and arcs to the computer's bitmap display. It was the first graphics hardware to qualify as a LSI ("Large Scale Integration") due to it containing tens of thousands of transistors in a single chip (Stengel, 2016; Bauer, 2015).

Other hardware worth mentioning is IBM's Color Graphics Adapter for the IBM

PC, released in 1981, and Intel's iSBX 275 Video Graphics Controller, released in 1982 (Bauer, 2015).

Moving to more recent time periods, Nvidia's GeForce 256 (Fig. 1.21), released in 1999 (Nvidia Corporation, 2016a) and promoted by the manufacturer as "the world's first GPU", popularized the term "GPU" to refer to such dedicated graphics-processing hardware. It supported the Direct3D 7.0 and OpenGL 1.3 APIs. The GeForce series became very popular, and also helped further standardize development workflows and tools for real-time graphics programming (ibid.). The subsequent GeForce 3 series added programmable shader support (vertex and pixel shaders) and support for Direct3D 8.0, while the GeForce 8 series was first to support USA (Unified Shading Architecture) (Nvidia Corporation, 2017a).

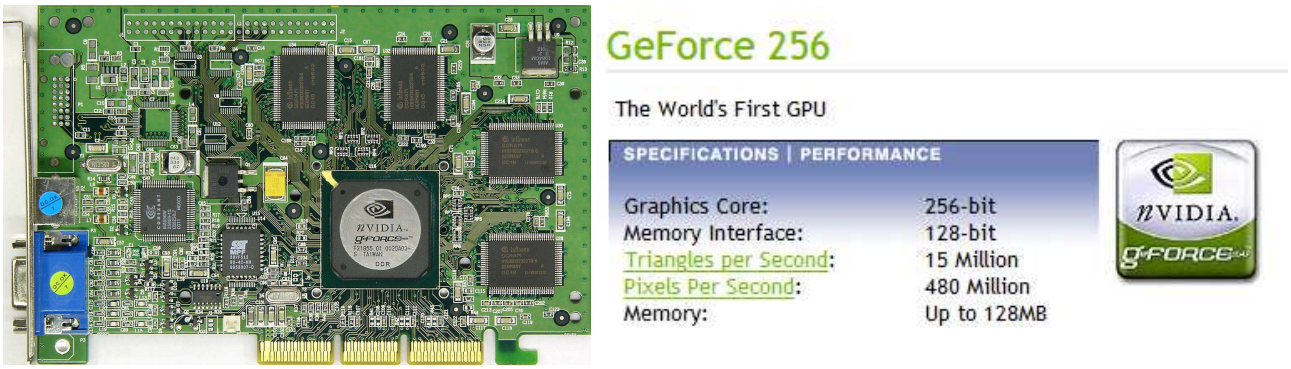

Figure 1.21: A screenshot from the GeForce 256 information page at the manufacturer's website (Nvidia Corporation, 2016a).

By this time, the GPUs discussed are highly parallel processing units, containing a large number of SIMD (Single Instruction Multiple Data) processor cores, where the programmable shaders constitute and specify the instruction(s) to be executed, and the input vertex and pixel data get distributed across those SIMD cores to be processed *in parallel*. This is the reason why such GPU architectures (Fig. 1.22) have also proven very appropriate for General Purpose Computing and Parallel Computing uses (Nvidia Corporation, n.d.; Harris, 2008).

It should be noted at this point that there are other graphics hardware configurations than *dedicated* GPUs. It is possible to have *integrated* GPUs, in the CPU or motherboard, which can be assigned a portion of system memory for *exclusive* use (partitioned memory), and it is also possible to have an integrated processing unit *share* memory with the CPU (see Figures 1.23 and 1.24). In this later case, CPU accessible data is also directly accessible by the GPU, alleviating the need to copy all data to GPU-visible memory (see Appendix B.7), and making the system's behaviour

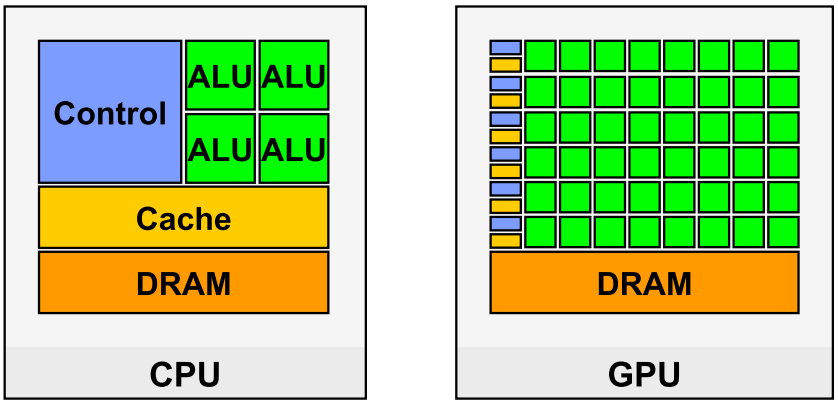

Figure 1.22: Due to its SIMD and data parallelism approach (as opposed to task parallelism that is characteristic of modern CPUs), a contemporary GPU is capable of containing a much higher number of ALUs in the same area than a contemporary CPU, thus managing to contain very high processing power (Nvidia Corporation, n.d.).

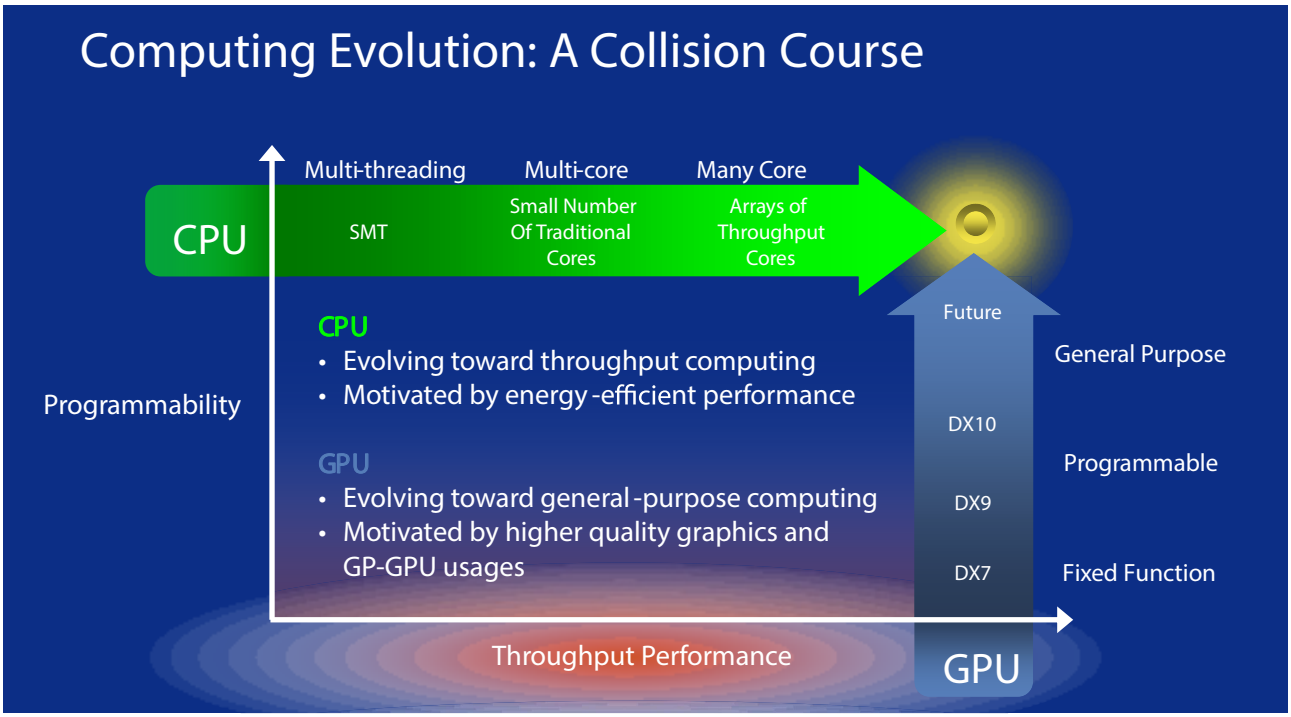

Figure 1.23: CPU and GPU evolution, moving towards eventual integration (Harris, 2008).

more uniform and more similar to CPU-based programming (Advanced Micro Devices Inc., 2017a). Note that the video game consoles XBox One and PlayStation 4 both use a GPU that corresponds to this third category of shared memory (Taylor, 2013; Shimpi, 2013; see Fig. 1.25), which the manufacturer, Advanced Micro Devices, Inc. (commonly known as AMD), also refers to as an "APU" or "Accelerated Processing Unit" due to the CPU and GPU being integrated in the same chip (Advanced Micro Devices Inc., 2017a).

Another related area of interest for real-time 3D graphics development which emerged in recent years is that of Heterogeneous Computing, along with the quite more recent Heterogeneous Systems Architecture (HSA) specification (HSA Foundation, 2016). A *heterogeneous* computing system is one that utilizes more than one type of processor, and HSA specifies a development and execution context that allows not only sharing memory but also *tasks* across heterogeneous processor types. The specific types of processors of the underlying system may be unknown to the developer of the executing program, and the program's tasks are appropriately scheduled for execution by the HSA platform. AMD's recent "APU"s mentioned previously make a good and relevant example of such a system, by combining the CPU and GPU not only in terms of shared memory, but in accordance with the HSA Foundation's specifications, of which AMD is a founding member.

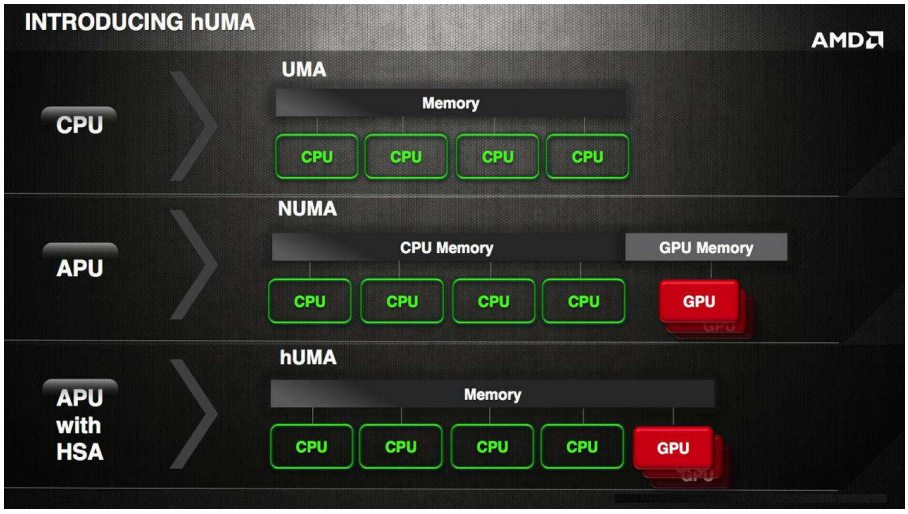

Figure 1.24: A diagrammatic illustration of a Unified Memory Access (UMA) architecture, a Non-unified Memory Access (NUMA) architecture, and what AMD refers to as a Heterogeneous Unified Memory Access (hUMA) architecture (Advanced Micro Devices Inc., 2017a).

Similar approaches are seen from other CPU/GPU manufacturers as well. For example, Nvidia's Tegra series (Fig. 1.26), for use with mobile devices, has similar

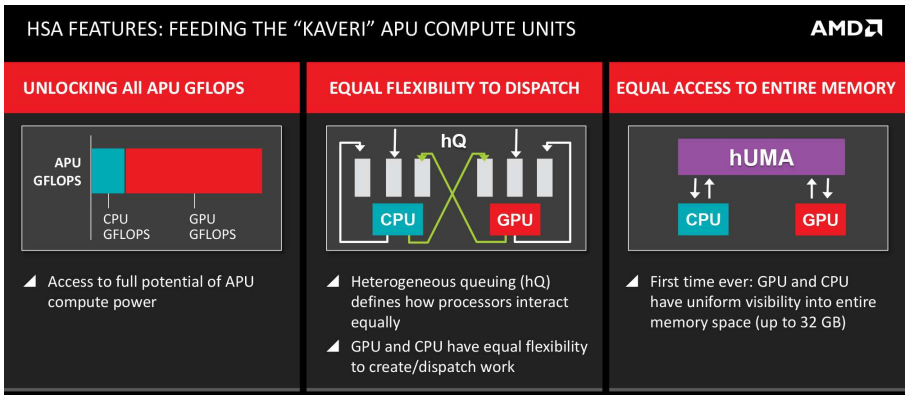

Figure 1.25: According to this 2013 presentation from AMD's annual developer conference (Smith, 2013), the "Kaveri" APU allows both the CPU and GPU to have uniform visibility into the entire system's memory, through the use of hUMA (Advanced Micro Devices Inc., 2017a).

features of multi-core CPU and GPU integration in a singe chip (Nvidia Corporation, 2017b).

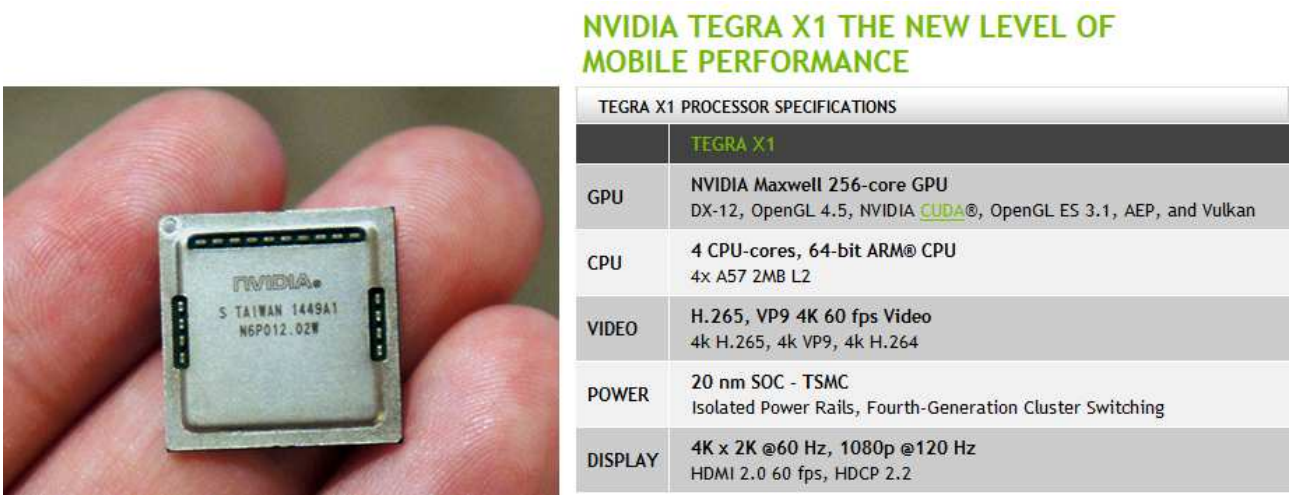

Figure 1.26: Nvidia Tegra X1 specifications and size (Nvidia Corporation, 2017b).

Other recently released hardware by different manufacturers with similar heterogeneous core integration principles include Intel's Haswell and Broadwell microarchitectures, Qualcomm's Snapdragon 810 and Apple's A7 and A8 (Bauer, 2015).

As noted already, this research focuses on GPUs with *exclusive* memory (partitioned or physically discrete), being significantly more common in use at the time of writing, and thus making the contribution of this research more widely applicable.

1.3.2 Author's work

The author also had the opportunity to work on some real-time rendering projects in previous years, a contributing factor in deciding to pursue this research topic, since it directly relates with questions and difficulties that the author had faced during performing work on those projects.

.OBJ 3D File Loader / Viewer (C++ / OpenGL), 2009-10

A personal educational endeavour constituting a real-time 3D viewer application for Microsoft Windows. The viewer loaded and displayed 3D data stored in Wavefront .OBJ files, allowing them to be interactively inspected by the user in real time (Fig. 1.27). A small portable library of linear algebra and 3D math functions was also implemented as part of it, providing all the required functionality to display and transform the imported 3D data.

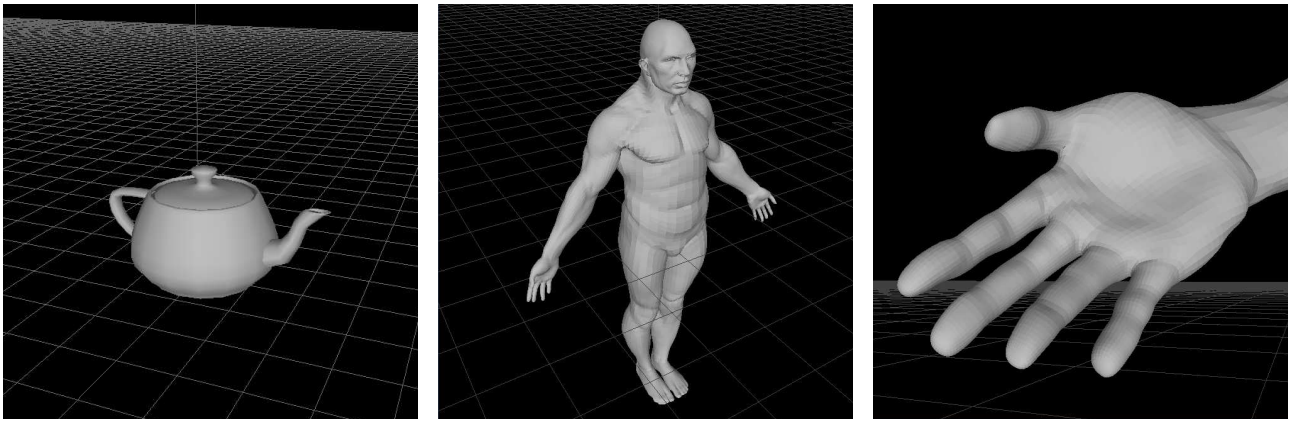

Figure 1.27: Indicative screenshots of the Wavefront .OBJ loader/viewer application, developed in 2010.

Real-time 3D Engine project (C++ / DirectX), 2010 - *(ongoing)*

An ongoing endeavour which so far has led to the development of **i)** a portable 3D math utilities and collision detection library in standard C++, **ii)** a renderer that conforms to a Phong-derivative shading model, **iii)** a dynamic, "open-world" scene-loading system using discrete spatial sectors, **iv)** a AI path network generation and path-planning system, and **v)** a set of custom file types, importers and exporters for efficient handling of the engine's data (see Fig. 1.28 and 1.29).

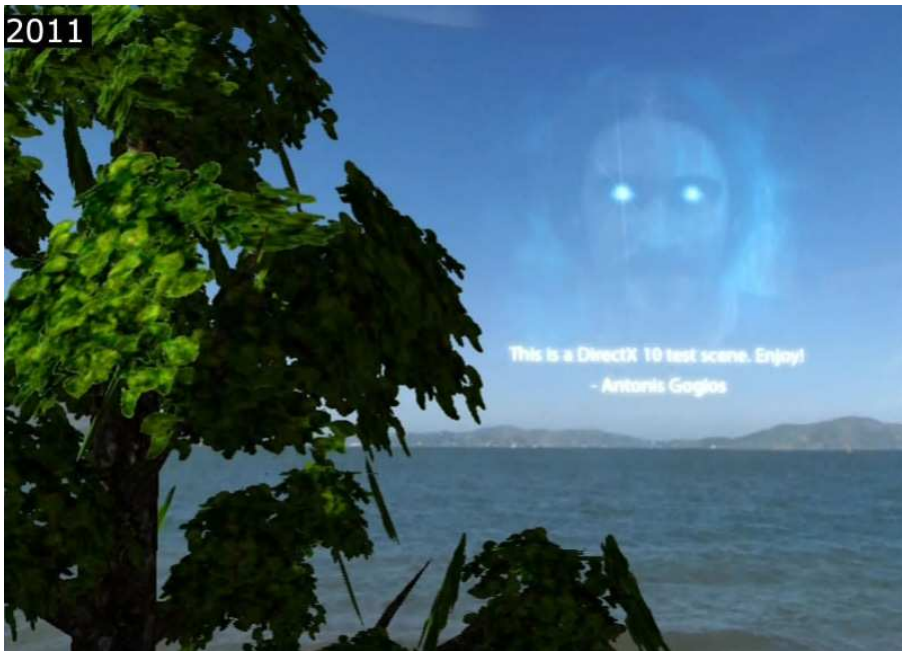

Figure 1.28: By 2011, the developed real-time 3D engine supported a Phong-derivative shading model fully, along with alpha-masking, depthmap shadows, and normal and specular texture maps.

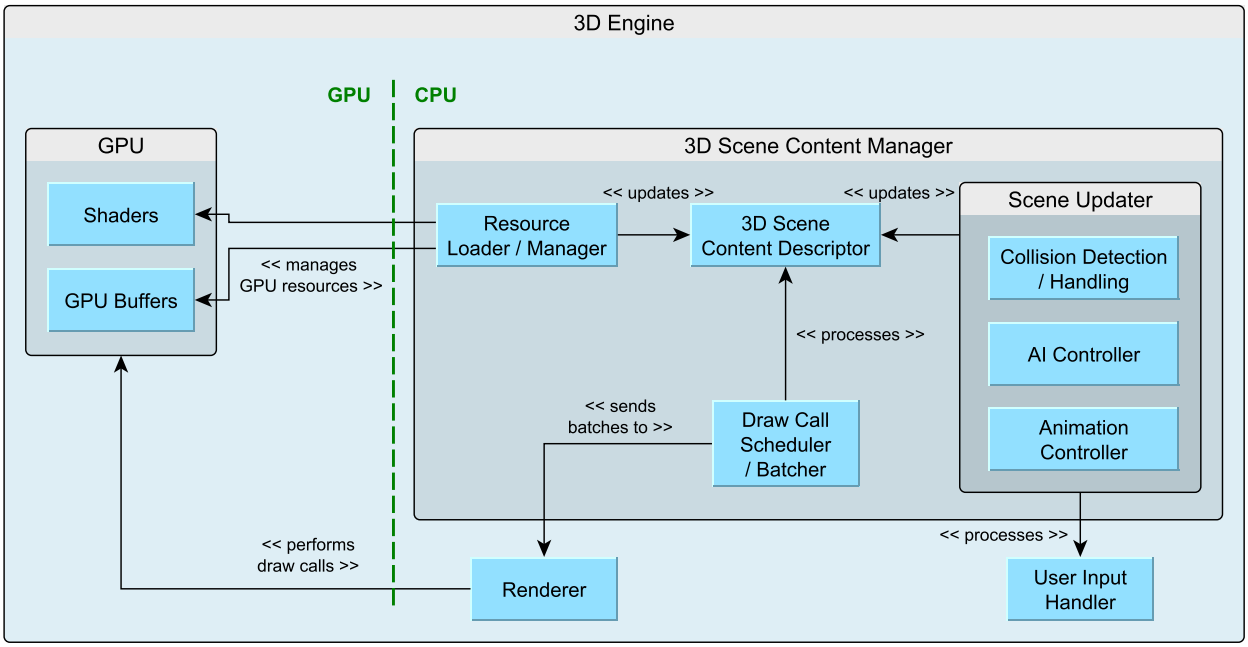

Figure 1.29: A diagrammatic overview of the developed 3D engine's main system components.

1.4 Aims & objectives

1.4.1 Problem definitions

This study focuses on determining key performance optimizations in the field of real-time 3D rendering – particularly when dedicated hardware acceleration with exclusive GPU memory is involved. Performance is of pivotal importance for such systems, although there are problems that always need to be addressed:

1. **Increased system complexity:** The dedicated hardware used increases the heterogeneity of the system, and in this way also increases its overall complexity (see section 1.3.1). System component interaction – such as CPU-GPU interaction and synchronization – also introduces overhead.
2. **Overhead caused by exclusive memory:** Exclusive GPU memory introduces the need for additional data copying to be performed as part of interfacing with the GPU (see Appendix B.7). Data to be processed needs to be copied into the GPU address space, while the results require to be copied back to system memory. This increases the overhead of the interfacing process.

To effectively make the most of such a system and deliver the optimal performance, the developer must address these problems in an informed and appropriate manner. Furthermore, there are several factors requiring the constant review and refinement of previously determined solutions to these problems. System architectures change, making past solutions suboptimal when applied to newer systems.

1.4.2 Research aims

The primary aims of this study are:

1. Identify common performance bottlenecks in contemporary real-time 3D rendering;
2. Determine key optimization areas to address them;
3. Determine the most appropriate techniques to resolve those bottlenecks.

As it has been pointed out, this study more directly addresses managing the type of 3D data sets characterizing the fields of 3D CAD software, interactive simulation and video game development, without this excluding applicability to other areas as well.

2. Methodology

This study employs a quantitative comparative method of analysis that is based on the process of "*Performance Optimization and Tuning Testing*" as delineated in Liu (2009). According to Liu, performance optimization testing for software typically consists of the following steps:

1. Extract performance requirements for the software in question
2. Develop appropriate test cases
3. Decide on appropriate hardware for testing
4. Set up the testing environment
5. Form a detailed procedure about how your tests are conducted
6. Begin testing with an initial set of configuration parameters
7. Analyze performance bottlenecks
8. Implement optimizations at the application level
9. Get reliable quantitative data for all test cases
10. Use the collected measurements to arrive at recommendations for further optimization and tuning

The way each of these steps have been applied by the author in the course of this study is discussed below.

2.1 Performance requirements

2.1.1 Framerate

As noted in section 1.1, this study focuses on real-time, hardware-accelerated 3D rendering. In the same section, it has been pointed out that for rendering to qualify as real-time, it needs to comply with the lowest framerate allowing persistence of

vision, which is 24 FPS. Interactive rendering is preferred to comply with the higher limit of 60 FPS, which creates higher animation fluidity and is known to provide less exertion to the user in an interactive context (Möller, Haines and Hoffman, 2008). For this reason, 60 FPS is a common performance requirement in the field of video game development (ibid.). Since this study focuses on applications pertaining to the fields of 3D CAD software, interactive simulation and video game development (see section 1.4), a minimum framerate of **60 FPS** is adopted as a requirement.

As for an *upper* FPS limit, no such limit is set for the initial configuration and measurements, although appropriate upper limits are discussed in the analysis (see section 3.5).

2.1.2 Response time

Beyond the system's framerate, another fundamental aspect of software performance that must be considered here is response time. An interactive real-time rendering system contains a central loop structure which constantly iterates, receiving user input and outputting rendered frames (a full description of the process can be found in Appendix B.5). In a single-threaded context, the iterations of this loop constitute a useful measure for reliably representing the system's response time, since they directly correspond to the frequency of its I/O processing capability. Naturally, if frame rendering is unconditional, the achieved FPS will also equal the number of iterations per second. However, if rendering is conditional, these values can vary substantially, as will be demonstrated in the analysis.

Maximizing responsiveness is of central importance to an interactive real-time rendering system, since in many cases the effective and timely processing of high-reflex user input is of pivotal importance for the software's success. For example, a low degree of responsiveness is extremely noticeable and can make use of 3D CAD or video game software very frustrating for the user, due to the high level of interactivity involved.

Also, particularly in a real-time rendering context, maximizing responsiveness is an always desired and never wasted effort because such increased responsiveness can represent a "raw resource" for the software involved to sustain future expansions: The execution speed gained through optimization can be directly re-invested in increasing the complexity of the computations, and be translated into features

such as more physically accurate shading, higher polygon count, more 3D assets, and more complex user interaction.

As treated in section 2.1.1, 60 FPS has been set as the minimum acceptable framerate. This is a requirement that also indirectly sets the minimum number of acceptable iterations per second to **60**. As the analysis will show however, even a fully non-optimized contemporary system maintains a frequency of several thousand iterations per second. Hence, this study adopts the more realistic performance requirement of measuring the iterations per second achieved by the initial configuration (see section 2.6), and then attempting to *maximize* that through resolving all observed performance bottlenecks (see section 2.7).

2.2 Test scripts

2.2.1 Test data set

Before considering the applicable test cases, it is important to first determine the characteristics of a representative data set to be used in the tests, and to also obtain such a data set.

Current field literature discussing 3D asset creation for interactive real-time use adheres to *modularity* and *reusability* (Fig. 2.1) as the main principles that should guide the asset creation process (Perry, 2002; Burgess and Purkeypile, 2013; Pluralsight LLC, 2012; Stephens, 2011, 2014; Kinney, 2014). With these principles in mind, the desired 3D content should be systematically broken down to more manageable objects/pieces, to be individually constructed by the 3D artists involved in the project (ibid.). The resultant 3D object data would then be used as modular components to assemble a larger 3D scene or "world".

Using this approach, a 3D data set was assembled from the 9 objects listed in Table 2.1:

Figure 2.1: Modularity and reusability in contemporary 3D content creation. Top: A typical 3D modular "kit", composed of tens of individual components, capable of being variously assembled to compose more complex environments (3DRT, n.d.). Bottom: Screenshot from a demonstration of 3D modular asset production workflows (Pluralsight LLC, 2012).

Name	Triangle count	Material count	Times used
Stanford bunny (reconstructed)	12394	1	1
Stanford dragon (reconstructed)	49512	1	1
Teapot	1232	1	5
Ground	19200	1	4
Wall segment	274	1	64
Pyramid	6	1	3
Sphere A	30	1	8
Sphere B	1140	1	8
Cylinder	80	1	3

Table 2.1: The test data set's 3D objects.

The resultant 3D scene contains in total 97 instantiations of these 9 objects, totaling 172020 triangles. It is noted that the Stanford bunny and dragon are low-polygon versions of their originals (Stanford Computer Graphics Laboratory, 2017). As illustrated in Fig. 2.2, the instantiated objects' placement is such that facilitates tests that examine the effects of overdraw (see Appendix B.11), since the position and orientation of the camera creates a high-overlap area in the centre of the frame. The modularity and high number of instantiations also serve in making this an appropriate data set for demonstrating the effects of the GPU's "Instancing" rendering mode (as discussed in section 3.4).

2.2.2 Test cases

The test cases employed are designed to address efficient GPU resource management. This area of focus was chosen because it is one that necessarily applies to a wide range of real-time rendering application types. By avoiding to make specific assumptions about the content and calculations involved in the scene, the contribution of the study becomes more relevant and applicable to a wider range of situations. For example, in a contemporary video game development context, shading is frequently particularly demanding, due to attempting to accurately represent environments with a high degree of photorealism. In contrast, 3D CAD software, while also real-time and highly interactive, usually has much simpler shaders because its focus is different. The same applies for real-time 3D simulation, where the highest complexity most likely is in the simulated effect or phenomenon. Hence, different application types can make widely different usage of GPU resources, al-

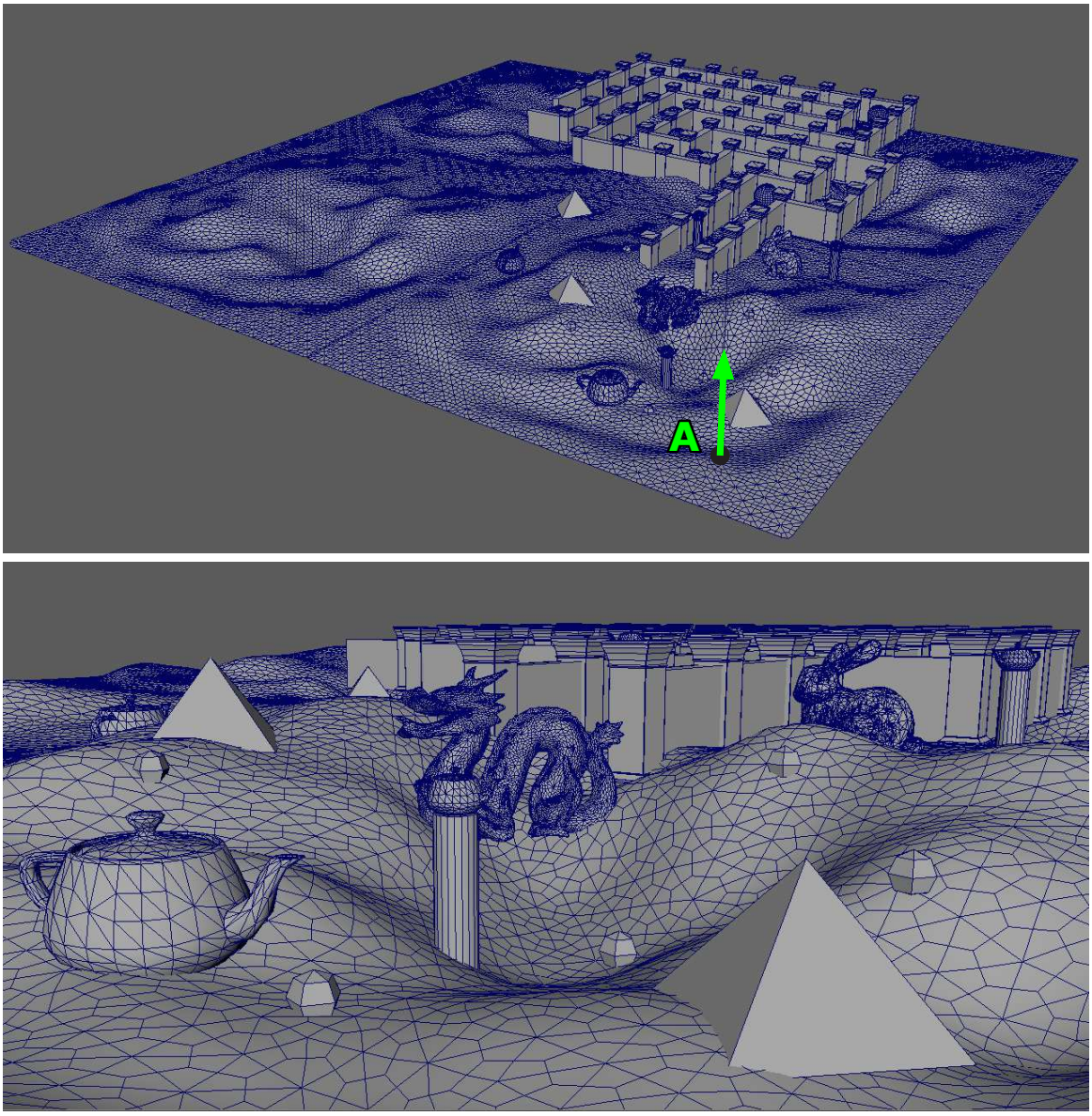

Figure 2.2: Screenshots of the test data set used in the study. Top: The virtual camera is positioned at "A", and oriented in the direction of the vector. Bottom: The test data set, as seen from the point of view of the virtual camera.

though efficient *management* of those resources is pivotally important in all these situations.

There are 14 test cases employed:

- **TC-01:** This test case represents the initial system configuration (discussed in section 2.6), where no optimizations are employed and initial measurements are taken.
- **TC-02:** Computationally identical with TC-01 except a collective vertex and collective index buffer is used.
- **TC-03:** Computationally identical with TC-02 except a frame limiter is used, with an upper limit of 60 FPS.
- **TC-04:** Computationally identical with TC-02 except 5 constant buffers are used instead of a collective one.
- **TC-05:** Computationally identical with TC-02 except front-to-back sorting is applied to rendered objects.
- **TC-06:** Computationally identical with TC-02 except back-to-front sorting is applied to rendered objects.
- **TC-07:** Computationally identical with TC-02 except immobile objects are pre-transformed to World Space.
- **TC-08:** Computationally identical with TC-02 except instancing is enabled.
- **TC-09:** Computationally identical with TC-08 except a frame limiter is used, with an upper limit of 60 FPS (as in TC-03).
- **TC-10:** Computationally identical with TC-08 except 5 constant buffers are used instead of a collective one (as in TC-04).
- **TC-11:** Computationally identical with TC-08 except front-to-back sorting is applied to rendered objects (as in TC-05). Sorting is applied independently to each batch of instances.
- **TC-12:** Computationally identical with TC-08 except back-to-front sorting is applied to rendered objects (as in TC-06). Sorting is applied independently to each batch of instances.

- **TC-13:** Computationally identical with TC-08 except immobile objects are pre-transformed to World Space (as in TC-07).
- **TC-14:** All optimizations are enabled for a cumulative effect: Collective vertex/index buffers, 5 constant buffers instead of one, instancing, front-to-back sorting per instance batch, a frame limiter with an upper limit of 60 FPS, and pre-transformation of immobile objects to World-Space.

The rationale and effects of each of these test cases is thoroughly covered in section 3. As the above description demonstrates, the focus of the tests is on GPU buffer management and minimizing the overhead caused by graphics API calls.

Also, for reasons mentioned above and due to the focus of these tests, the shaders (see Appendix B.8) employed are intentionally kept very simple (Fig. 2.3):

- Solid mode with back-face culling enabled
- Diffuse shading according to the Blinn-Phong shading model (see section 1.3.1)
- A single directional light, no shadows
- Texturing is used to sample the diffuse colour of each object per-pixel. A single 2048 x 2048 texture is employed for all models

2.3 Hardware

A system with exclusive GPU memory and DirectX 11 support has been used:

- **GPU:** NVIDIA GeForce GTX 750 Ti
- **Graphics clock:** 1058 MHz
- **Memory data rate:** 5400 MHz
- **Memory interface width:** 128-bit
- **Memory bandwidth:** 86.40 GB/sec
- **Dedicated video memory:** 2048 MB GDDR5

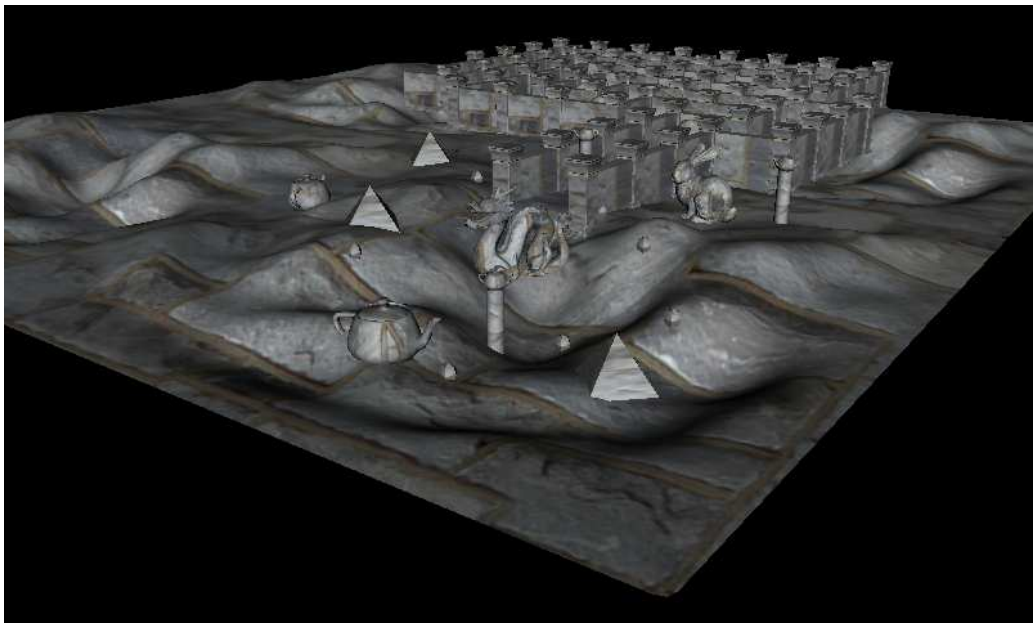

Figure 2.3: The data set after import into the testing environment, and shaded by it.

- **System video memory:** 0 MB
- **Shared system memory:** 2048 MB
- **Direct3D feature level:** 11_0
- **CPU:** Intel i5-6500 (3.20 GHz)
- **System memory:** 16 GB

2.4 Testing environment

The testing environment is a real-time rendering application developed for the purposes of this study. It incorporates multiple implementations that collectively represent the test cases mentioned in section 2.2.2. Based on the given user-defined configuration at the time of initialization, the application uses the appropriate implementation and renders the test data set from a specifically defined camera position and orientation (Fig. 2.2). After sampling and processing all relevant measurements, it outputs the test case's results in a log file.

Underlying software used by the testing environment is:

- **OS:** Windows 7 Professional SP1 64-bit

- **Direct3D API version:** 11

- **GPU driver version:** 355.82

2.5 Testing procedure

The procedure begins with the application initializing and receiving a configuration description provided by the user. The description is made to correspond to a given test case, informing the application regarding the desired configuration and allowing it to create and manage the needed CPU and GPU resources in the specified way.

Once initialized, the application begins rendering the data set and initiates the sampling process for the conducted measurements (see section 2.9). Among the settings defined in the configuration description are the "seconds to skip" and the "seconds to measure". The "seconds to skip" correspond to an amount of seconds to wait after real-time rendering is initiated but before sampling actually begins, in order to allow the application to achieve a more stable condition before measurements are taken. The "seconds to measure" represent the amount of time that sampling should last for a given session.

For the values listed in Appendix A and analyzed in this study, the seconds to skip were set to 10 and the seconds to measure were set to 300 (five minutes). This configuration was adopted due to the high amount of variance observed in the initial measurements that were conducted. This type of variance is most likely the result of the operating system's preemption combined with the complexity of the graphics API's internal operations, which can behave differently across successive API calls. To reliably filter out this variance prior to analysis, samples were initially taken for each application iteration (see Appendix B.5 for a full description of the central loop structure) for the duration of the sampling process, and then post-processed in order to acquire their arithmetic mean value. This operation's results form the measurements analyzed in this study.

2.6 Initial system configuration

The initial system configuration corresponds to test case TC-01, and is the fully non-optimized state initially tested. As discussed in section 2.6.1 below, TC-01 is configured to represent a real-time implementation of an otherwise generic rendering algorithm. Although employing hardware acceleration, it does not take into consideration or address hardware-accelerated rendering's distinctive bottlenecks (see section 2.7), capabilities and peculiarities, thus resulting in a suboptimal, somewhat naive implementation – the particulars of which are treated extensively in the analysis (section 3).

TC-01 uses 4 vertex and 4 index buffers, a single constant buffer, renders frames unconditionally, does not sort objects prior to rendering, and does not use GPU instancing. This initial system configuration will also be referred to as "A1" ("Algorithm-01") in this study, as opposed to "A2" which represents TC-14 and incorporates all the optimizations treated in the analysis.

2.6.1 The generic algorithm A1

The A1 algorithm represents and applies the basic principles of the rendering process as specified in Appendix B. After application initialization and loading of the 3D content is complete, it simply follows the abstract main loop of **i)** Process user input, **ii)** Update scene, and **iii)** Render scene (described in Appendix B.5) until the exit condition is met. In pseudocode, it amounts to:

[Initialization] :
1. Initialize application
2. Load 3D content
3. Execute **[Per-loop]**

[Per-loop] :
1. Process user input
2. Update scene based on input
3. Sequentially draw all 3D objects
4. Check exit condition, repeat **[Per-loop]** if false

What makes it a generic algorithm is that it does not address its implementation

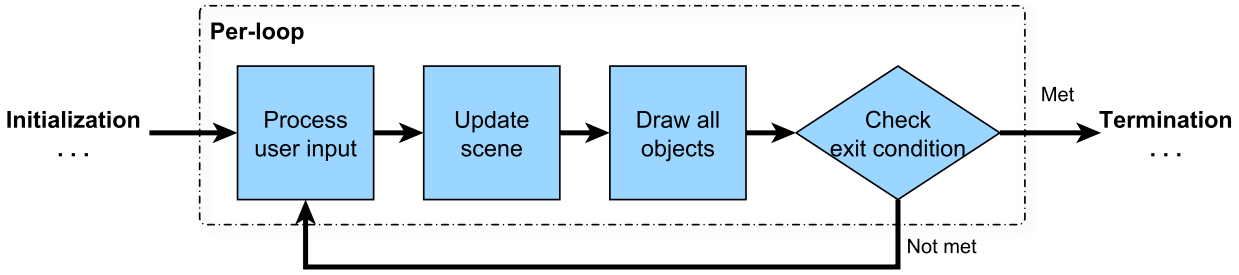

Figure 2.4: A diagrammatic overview of the generic algorithm A1.

context in more specific terms than required, but only performs the steps of the abstract main loop from a very high-level standpoint. Hence, although it is implemented in a real-time and hardware-accelerated graphics context, it does not take any specific features of that context into account, resulting in a suboptimal implementation that ignores all the areas that optimizations specific to the real-time rendering context could be made. In turn, A1 embodies the performance bottlenecks this study aims to resolve, listed below.

2.7 Performance bottlenecks

The performance bottlenecks addressed in the present study, and around which the remaining test cases have been designed, are the following:

1. **GPU buffer management:** The inefficient configuration of GPU buffers – in terms of their *number*, their *size* and their *contents* – is a significant source of overhead (Luna, 2012; McDonald, 2012). Several of the test cases are designed to illustrate and examine how different configurations can impact performance.

2. **Overdraw:** Overdraw involves the full recomputation of a given pixel's value during rendering (see Appendix B.11), and is a factor that can affect performance (Luna, 2012). The test cases examine ways that overdraw can be reduced, increasing frame computation speed.

3. **Graphics API calls:** API calls like these are required for purposes like managing/updating GPU resources and issuing rendering commands (i.e. performing "draw calls"). These quite often involve high overhead (McDonald, 2012), and it is desirable to minimize them.

4. **3D transformations:** As noted in Appendix B.13, transformations occur across a number of different coordinate spaces as a frame is rendered. Since these involve a high number of floating point multiplications, it is desirable to minimize them wherever possible.

2.8 Tested optimizations

To address the performance bottlenecks listed in 2.7, the optimizations to be tested are as follows:

1. Collective vertex/index buffers, addressing the overhead of managing multiple buffers and reducing the required graphics API calls to do so. This optimization is analyzed in section 3.1.

2. Frequency-of-update constant buffers, distributing buffer content across several smaller buffers (as opposed to a collective constant buffer) and grouping data members in accordance with their frequency of update. This optimization is analyzed in section 3.2.

3. Front-to-back sorting of objects prior to rendering, which reduces the overdraw while rendering the frame. This is analyzed in section 3.3.

4. Instancing, a GPU rendering mode that dramatically reduces API draw calls for a given frame. Analyzed in section 3.4.

5. Employing a frame limiter, which conditionally skips rendering of a frame if the specified frame limit is met, and in this way radically reduces the amount of API calls performed in a given second. Analyzed in section 3.5.

6. Pre-transformation of strictly immobile objects, allowing them to skip per-frame World-Space transformation and be only transformed once when they are loaded. Analyzed in section 3.6.

These optimizations are collectively enabled in test case TC-14, which also represents the optimized algorithm referred to as "A2" (as opposed to "A1", represented by TC-01).

2.8.1 The optimized algorithm A2

The A2 algorithm addresses the system's increased complexity (see section 1.1), and more importantly the system's areas characterized by high computational overhead – and pays attention in their optimal management. It contains all optimizations treated in this study. In pseudocode and from a sufficiently high-level standpoint, the algorithm amounts to:

[Initialization] :
1. Initialize application
2. Load 3D objects
3. Pre-process loaded content for GPU
4. Create GPU buffers
5. Execute **[Per-loop]**

[Per-loop] :
1. Process user input
2. Update scene based on input
3. Conditionally render frame
4. Check exit condition, repeat **[Per-loop]** if false

Areas of particular interest for this study are:

- In **[Initialization], step 2**, the object loader makes a clear distinction between 3D objects that are strictly immobile (e.g. the ground), and objects that are not. The first are *pre-transformed* to World Space coordinates (see Appendix B.13) during loading. This is done so that subsequent per-frame (and *per-vertex*) World-Space transformation can be avoided for those objects, with the clear assumption that their World-Space position will never change. This is part of the optimization analyzed in section 3.6.

- In **[Initialization], step 3**, the pre-processing performed involves the aggregation of the loaded 3D object data into collective vertex and index buffers. Individual objects can be referred to through their indices relative to the buffer's beginning (analyzed in section 3.1).

- Also in **[Initialization], step 3**, the instance batches and contents of the instancing buffer to be used are assembled. The "instancing buffer" referred to

is a vertex buffer containing per-instance data, required for using the GPU's instancing rendering mode (analyzed in section 3.4).

- In **[Initialization], step 4**, among the GPU buffers (see Appendix B.7) created (i.e. vertex, index and instancing buffers), the algorithm creates a number of constant buffers that represent the frequency of updates that their content has (analyzed in section 3.2).
- In **[Per-loop], step 3**, the algorithm makes use of a *frame limiter* (a rendering system component implemented as part of the testing environment), to determine if the elapsed time since last rendering a frame necessitates rendering another one. This depends on the specified *frame limit*, which the author defines as a value representing the desired maximum framerate (see Appendix B.4) to be adhered to by the application. This optimization is analyzed in section 3.5.
- Also in **[Per-loop], step 3**, and provided that a new rendering is required (which is determined by the frame limiter), the algorithm sorts the objects of each batch to be rendered in *front-to-back order* from the point of view of the camera. This is done to reduce overdraw (see Appendix B.11), and is analyzed in section 3.3.
- Lastly, once again in **[Per-loop], step 3**, while the frame is rendered, the algorithm checks each batch prior to rendering it. If the contents are classed as strictly immobile (and thus pre-World-Space transformed, as discussed in [Initialization], step 2), a vertex shader is used (see Appendix B.8) that assumes that the object rendered *already is* in World Space, rather than Local Space (see Appendix B.13), skipping the World-Space transformation.

The above is also summarized in a diagrammatic overview of the algorithm, in Fig. 2.5 (depicting the [Initialization] part) and Fig. 2.6 (depicting the [Per-loop] part).

2.9 Conducted measurements

The measurements conducted during the sampling process are intended to reliably represent the test cases' performance level in accordance with the requirements listed in section 2.1, both for framerate and response time. The measurements are the following:

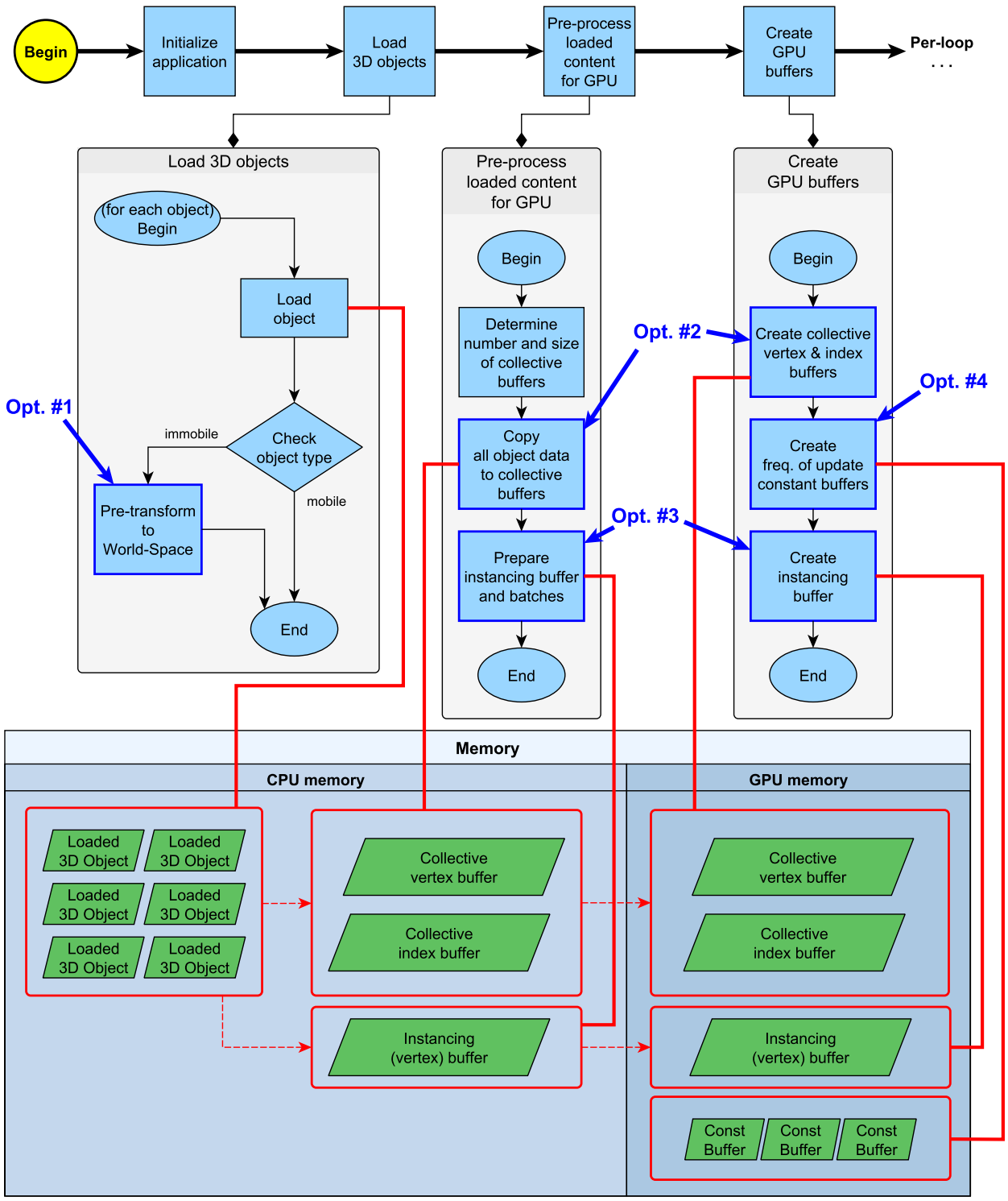

Figure 2.5: Overview of the "initialization" section of the optimized algorithm. Attention is drawn to areas where optimizations (noted as "Opt.") have been performed: 1) Pre-transformation of immobile objects to World-Space (see Appendix B.13), 2) use of collective vertex/index buffers (see Appendix B.7), 3) use of GPU instancing (see section 3.4) , and 4) creation of constant buffers whose number is determined by their frequency of update.

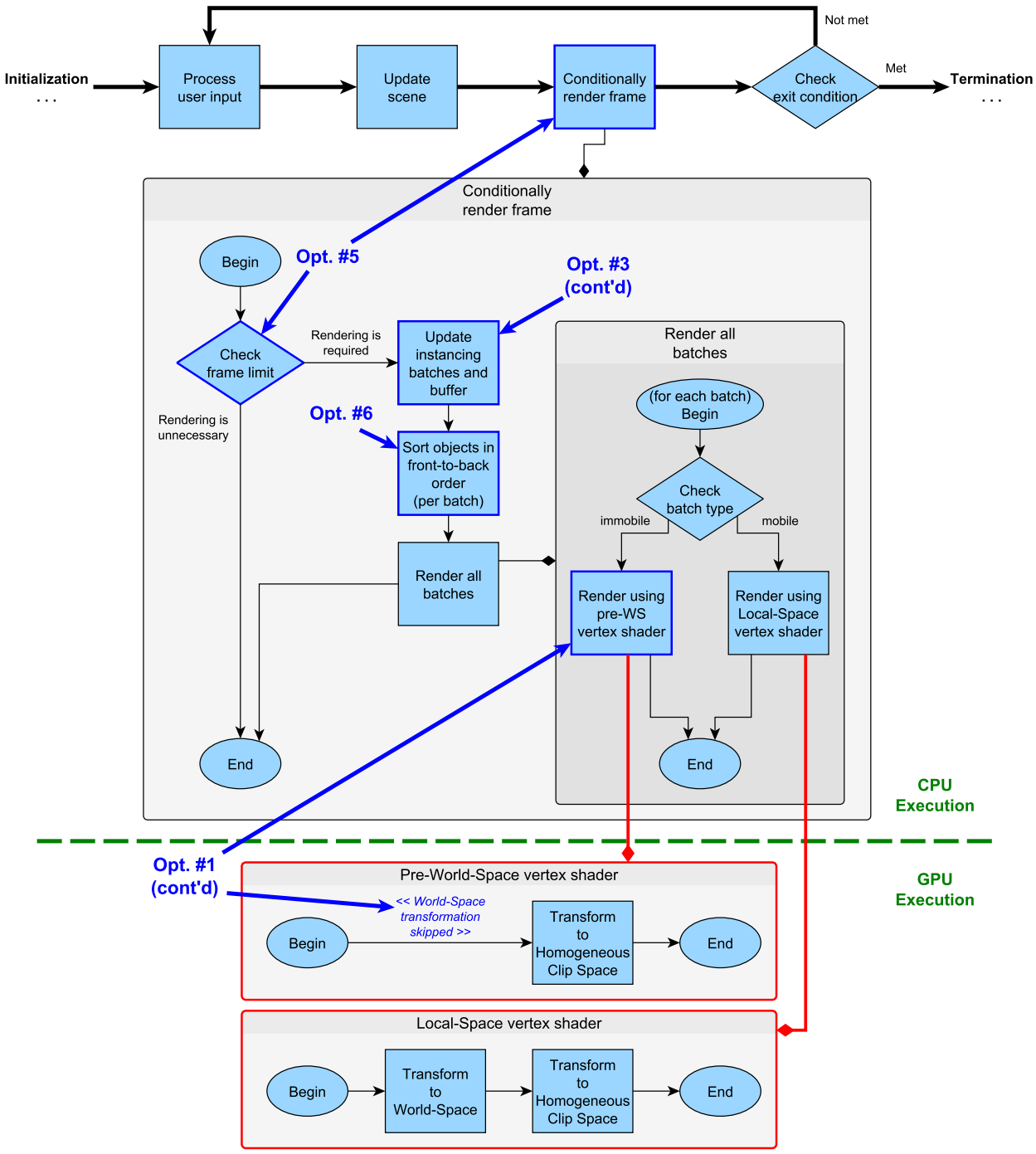

Figure 2.6: The "per-loop" section of the optimized algorithm. Again, attention is drawn to areas where optimizations have been performed, and noted as "Opt.". Opt. #5 constitutes a frame limiter for conditional rendering, and Opt. #6 the front-to-back sorting of objects prior to rendering. Opt. #1 and #3 continue from what was noted in Fig. 2.5. This diagram also contains parts of the algorithm that execute on the CPU and others that execute on the GPU, respectively noted as such. Lastly, the reason for using a *separate* vertex shader instead of a conditional flow for immovability inside a *single* vertex shader is explained in section 3.6, along with the entire optimization #1.

1. **Draw calls per frame:** The amount of API draw calls performed for a given frame.

2. **Frames per second (FPS):** As discussed in the Appendix B.4, this value represents the amount of rendered frames for a given second.

3. **Average frame-rendering time:** The average amount of time, in milliseconds, for a frame to be rendered with a given system configuration (i.e. test case).

4. **Iterations per second:** The amount of iterations of the central loop of the application (see Appendix B.5) for a given second. Since the application is single-threaded, this measure reliably represents response time.

2.10 Evaluation process

The evaluation process used to determine the efficacy of the optimizations is detailed comparison of the measurements for the evaluated test cases. Generally speaking, the average frame-rendering time is the most important value to observe, since a reduction indicates that the current configuration has a positive effect in improving performance. Correspondingly, an increase generally indicates that the current configuration has a negative effect in improving performance.

In addition, since iterations per second represent the response time of the application, an increase generally denotes a positive effect for performance, since more I/O iterations get completed in a given second. A decrease denotes the opposite.

Draw calls per frame are not directly used to determine performance, although are mentioned wherever it is relevant to demonstrate the effect on performance (directly determined by avg. frame-rendering time and iterations per second) that different amounts of API calls can have. Fewer draw calls are generally preferred, provided they achieve the desired rendered result.

Frames per second have a requirement to not be less than 60, and apart from this their increase provides no benefit because it is not visually appreciable to a viewer/user. However, FPS are often listed in the analysis to provide a fuller appreciation of the varying factors involved when comparing test cases and their configurations.

It is noted that the above is merely a general skeleton of the evaluation process employed, and does not necessarily represent all criteria considered in the analysis.

Hence, the reader is directed to the analysis section (section 3), where all considerations for each reached conclusion are treated and discussed in full.

3. Analysis

3.1 Collective vertex/index buffers

The GPU's vertex and index buffers contain 3D data used for rendering primitives (see Appendix B.3). After their creation, such buffers need to be "*bound*" to the rendering pipeline in a specific GPU input slot, so that the data can be used for rendering (MSDN, 2016; Luna, 2012).

In the context of the DirectX API, the exact number of GPU input slots available depends on the capabilities of the specific GPU in question and is measured through so-called Direct3D "feature levels" that represent discrete grades of GPU capacity (MSDN, 2016). For DirectX 11 and DirectX 12, the different feature levels all offer a maximum of 32 available input slots, while DirectX 10 offers 16 input slots (ibid.).

The number of GPU vertex and index buffers that can be created by an application is not specifically constrained by the API, and primarily depends on the amount of memory available to the GPU. However, it is generally suggested to keep it relatively low (Luna, 2012), since binding buffer resources to the input slots for rendering use involves computational overhead, and also having data separated across several buffers will eventually require consecutive binding to occur on a per-frame basis. APIs like DirectX, OpenGL and Vulkan offer functionality to utilize sub-portions of buffers in rendering computations (MSDN, 2016; Shreiner et al., 2013; Luna, 2012; Schott and Bishop, 2016; Sellers and Kessenich, 2016). Hence, even though the maximum available input slots offered by GPUs increase over the years, it is always best to try and collect one's data in larger, collective buffers. In some cases it is also possible to achieve data aggregation allowing multiple heterogeneous *buffers* to be stored in a single memory allocation (Schott and Bishop, 2016; Sellers and Kessenich, 2016).

In this section, the importance of avoiding buffer-switching overhead is highlighted, and what is proposed is to minimize the number of vertex and index buffers employed. If possible, 3D data should first be loaded in full, then copied over to the GPU in a collective vertex and collective index buffer, after indexing of individual objects' positions (i.e. offsets, see Fig. 3.1) within those buffers has been conducted by the application through CPU-based pre-processing (see section 2.8.1). Afterwards,

the application can bind these buffers once, and render the entire scene's content by accessing sub-portions of those two buffers (ibid.).

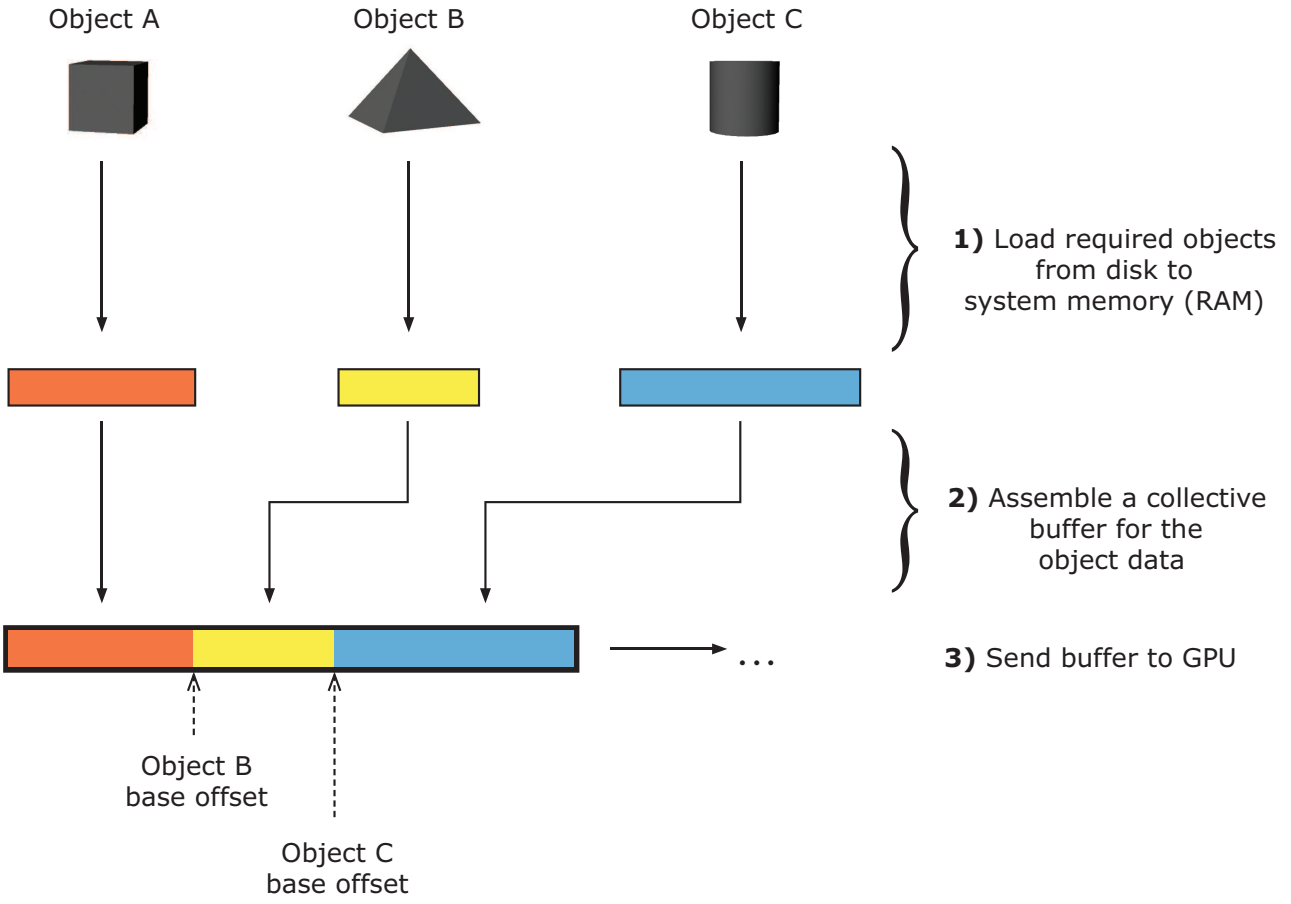

Figure 3.1: Diagrammatic overview of using collective GPU buffers. Minimizing the number of buffers in this way also minimizes the overhead caused by the need to switch among, and re-bind resources to the GPU pipeline.

Clearly, there are several situations where, due to the requirements and restrictions imposed on certain types of 3D applications, the use of only a *single* vertex and index buffer can prove untenable. This is particularly true for interactive applications. Here are some indicative examples:

- It is not uncommon for the exact content of a 3D scene to be impossible to be known in advance, but rather to be determined dynamically based on user input – e.g., in "open world" types of environments, where the scene is divided in discrete sections that are loaded and displayed *sequentially* as the player/viewer approaches them, and where the user's input may dynamically *request* content.
- The structure of the utilized 3D data may also be non-uniform – e.g. data with

different type and number of vertex attributes, etc. – often requiring that it is stored in different containers.

- In certain cases it may even be *desirable* to make the 3D data non-uniform in order to conserve overall GPU memory (Fernando and Pharr, 2005, ch. 5).

Despite the fact that use of only a single vertex/index buffer may be inapplicable in such scenarios, the suggestion made in this section is to nevertheless try to *minimize* the number of such buffers, because the API calls required for their management are a source of high overhead. The following analysis of relevant test cases demonstrates this effect.

Test case analysis

Two test cases were specifically designed for testing this assumption (Table 3.1). TC-01, representing the generic algorithm A1 in its entirety, utilizes 4 vertex and 4 index buffers to store the scene's data. The first vertex/index pair contains the Stanford dragon (see section 2.2.1), the second the Stanford bunny, the third contains the ground, and the fourth contains all other models. As the frame is computed, it is required to switch among these buffers, until all objects are rendered. TC-02, involves exactly the same configuration and computations as TC-01, with the only difference being that it uses a collective vertex and collective index buffer, which contain *all* scene data. This eliminates the need to re-bind buffers between draw calls, thus reducing the number of API calls required to compute the frame.

	TC-01	**TC-02**
FPS	3441	3676
Avg. frame time (ms)	0.290503	0.271727

Table 3.1: TC-01 and TC-02 comparison.

The taken measurements listed in Table 3.1 demonstrate that TC-02 rendered **6.46%** faster than TC-01 (also see Appendix A for a collective table listing all measurements). This difference is to be directly attributed to the absence of the overhead caused by switching among vertex and index buffers. Hence it demonstrates that it is optimal to employ the smallest possible amount of such buffers.

This observation, although small in itself in terms of direct performance gains, is significant in other ways that will be made clear in the course of this study.

Subsequent test cases attempt various sorting approaches for the order the objects get rendered in. The underlying notion is to minimize computations and context re-setting operations between draw calls, so objects with identical resource dependency contexts are sorted adjacently in order to avoid setting that context more times than is required. Buffers are one such resource, as are textures, material attributes, etc. . These can often create conflicting requirements during sorting, creating an inability to sort the objects efficiently because of their heterogeneous resource dependencies.

As a result, apart from its direct performance gains, collective buffers usage also increases *homogeneity* in resource dependency between objects, which will boost the effectiveness of subsequent sorting techniques discussed in the sections to follow. For this reason, collective buffers usage is employed in all subsequent test cases.

3.2 Frequency-of-update constant buffers

For hardware-accelerated rendering systems using exclusive GPU memory, data requires to be copied to GPU memory before it can be used in GPU-based calculations (see Appendix B.7). The destination of such copying is GPU buffers. However, it is often the case that the data involved is not constant, but must be updated during the course of the application session. Common examples of such data can be a light's direction vector or its intensity, a rotation angle value or an object's colour, and so on.

A GPU buffer with data meant for shader program input (see Appendix B.8) is called a "constant buffer" in DirectX API terminology (MSDN, 2016). Also do note that, despite their name suggesting invariable content, constant buffers are primarily used to store shader-accessible variables, and thus require very frequent updates. The word "constant" derives from its contents' invariability during the course of a draw call's execution (ibid.).

Updating a portion of data placed in a constant buffer requires the update of the *entire* constant buffer (ibid.). No individual members of the buffer can be updated independently. This is a feature aimed at aiding overall performance, provided the application utilizes it conscientiously and the developer understands how it operates. Misuse of this feature can severely affect performance due to the overhead associated with interfacing with the GPU (McDonald, 2012).

In such a context, it is always desirable to minimize interaction with the GPU due to such overhead. According to McDonald (2012), inefficient GPU resource management introduces what he refers to as CPU-GPU "Sync Points" into a real-time rendering application. These sync points are caused when the CPU requires that the GPU completes work before CPU work can be resumed (ibid.), thus increasing total frame rendering time. The same source mentions that a single sync point has the potential of possibly halving an application's frame rate – it is thus particularly important to manage GPU resources effectively.

According to guidelines commonly given when discussing real-time rendering systems (as in McDonald, 2012; MSDN, n.d. b; Luna, 2012; and others), it is suggested to group data in constant buffers based on their *frequency of required updates*. This is done so that the performance impact of the updates is minimized, an effect that will be thoroughly examined in the subsequent test case analysis for this section. At the same time, the data should conscientiously be grouped so that the total number of constant buffers used is relatively small (McDonald, 2012; Luna, 2012), thus reducing the *number* of updates required. Usually, the suggested number of constant buffers is about 5 (ibid.). It should be noted here that this particular number is of course a rule of thumb based on mostly empirical observations, involving real-time applications such as video games mainly rendering 3D environments representing physical landscapes, realistic lighting conditions, and so on. It is clear that a real-time rendering application with a different purpose could require a different amount of constant buffers, making this determination ultimately dependent on the scope and purpose of the application at hand.

In any case, it remains certain that in any given real-time rendering context, one should attempt to distribute data into constant buffers based on their frequency of updates, while at the same time trying to keep the number of constant buffers relatively low. Let us now examine some test cases that can demonstrate this.

Test case analysis

Two test cases are examined in this section, TC-02 and TC-04. TC-02, as mentioned in section 3.1, employs only the collective vertex/index buffer optimization. In terms of constant buffer usage, TC-02 uses a collective constant buffer containing all data requiring such a buffer. This buffer is updated once per object, so that the subsequent draw call can have access to all shader constants needed by that object.

TC-04 computationally replicates TC-02 with the exception that, instead of using a collective constant buffer, it distributes its content into 5 smaller constant buffers. Each buffer's content is chosen in accordance with the "frequency of updates" approach described earlier in this section. As demonstrated in Fig. 3.2, the total size of the data members remains the same, it is only the distribution of those members across buffers that changes.

Since TC-04 employs 5 distinct constant buffers, it updates them in appropriate intervals. Buffers with lower frequency of update than "per-object" get pushed back accordingly in the pipeline:

- "Immutable CB" is updated only once at the application's initialization
- "Per-frame CB" is updated once per frame
- "Per-camera CB" is updated once per frame (only one camera is employed)
- "Per-object CB" is updated once per object (97 updates)
- "Per-material CB" is updated once per object (97 updates)

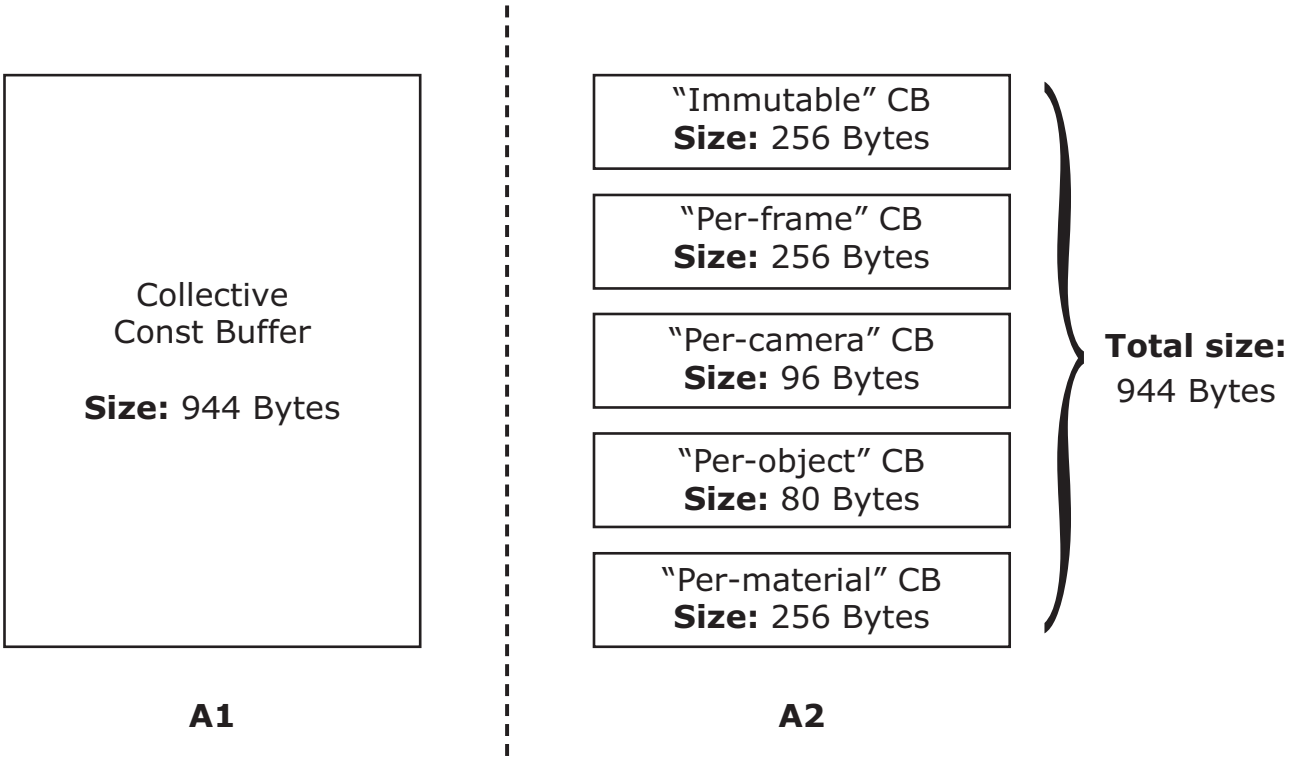

Figure 3.2: Comparison of the constant buffers employed by the generic algorithm A1 and the optimized algorithm A2.

	TC-02	TC-04
FPS	3676	3797
Avg. frame time (ms)	0.271727	0.262851
Draw calls/frame	97	97
Const. buffer updates/frame	97	196
Total bytes updated/frame	91568	32944

Table 3.2: TC-02 and TC-04 comparison.

As shown in Table 3.2 and Fig. 3.2, for a given frame, TC-02 performs 97 updates of its single, collective constant buffer, with a size of 944 bytes. On the other hand, TC-04 performs 196 updates, since its buffers get updated with varying frequencies. Note that these updates are excessive and could be easily sorted and batched further in order to reduce update API calls for even better performance than what TC-04 achieves here, but they intentionally were not in order to simply observe the difference by employing the "frequency of updates" principle without any additional changes or applied improvements.

TC-04 may perform more than twice TC-02's update API calls, although it still updates **64%** less bytes per frame than TC-02 does. This difference in the *size* of the updates is the reason for the performance difference we see, which amounts to a render speed increase by **3.26%** (see Table 3.2).

Hence, in this case, due to the unconditional full updating of constant buffers, using a collective buffer was not an optimal approach, as it was in the cases discussed in section 3.1, which involved vertex and index buffers. With regard to constant buffers, employing the "frequency of updates" principle to determine their number has the best results.

3.3 Front-to-back rendering

As demonstrated in Appendix B.11, given a set of 3D objects, the order that the objects get sequentially rendered in matters. A 3D scene's content must be translated into a series of draw calls (described in Appendix B.7) in order for a frame of that scene to be rendered. The order of the draw calls affects which objects get rendered first and which afterwards. When depth culling (see Appendix B.10) is enabled, subsequent draw calls often overwrite pixel fragments calculated previously. This

occurs if the new fragment has a lower depth value than the old fragment, thus representing an object that is closer to the camera.

In consequence of the above, it can be suggested that sorting the draw calls so that the rendered objects are in front-to-back order (that is, so that an *ascending* order of the rendered objects' distance from the camera is achieved), the pixel fragments that result from a draw call will likely not be overwritten by the next draw call – at least not while depth culling is enabled. This is because the previous pixel fragments will fully occlude the new ones, resulting in the new ones to skip full computation and be discarded.

Test case analysis

In order to test the efficacy of this optimization of sorting draw calls in front-to-back order, a series of test cases are examined. TC-05 and TC-06 are both computationally identical to TC-02, with the only difference that they apply distance-based sorting to the scene's objects prior to rendering. TC-05 applies front-to-back sorting relative to the camera, while TC-06 applies back-to-front sorting. Table 3.3 contains the relevant measurements for them:

	TC-05	**TC-06**
FPS	3772	3671
Avg. frame time (ms)	0.26461	0.271991

Table 3.3: TC-05 and TC-06 comparison.

Due to the increased overdraw caused by back-to-front sorting, TC-06 has a decreased rendering speed by **2.78%** compared to TC-05 (see Table 3.3). TC-05 manages to avoid the recomputation of a large portion of the frame's pixels by successfully culling pixel fragments of occluded objects during the depth culling phase.

By examining a sample frame of TC-05 in Visual Studio's Graphics Analyzer utility (Microsoft Corporation, 2015), the pixel history of a pixel in a high-overlap area of the frame indicated an overdraw of **x1** – corresponding to the object occupying that pixel that was closest to the camera across that ray. An equivalent frame of TC-06 indicated an overdraw of **x10** for the same pixel, since all objects from the background all the way to the foreground passed the depth culling phase successfully (Fig. 3.3).

TC-05 also demonstrates a **2.61%** increase in rendering speed compared to TC-

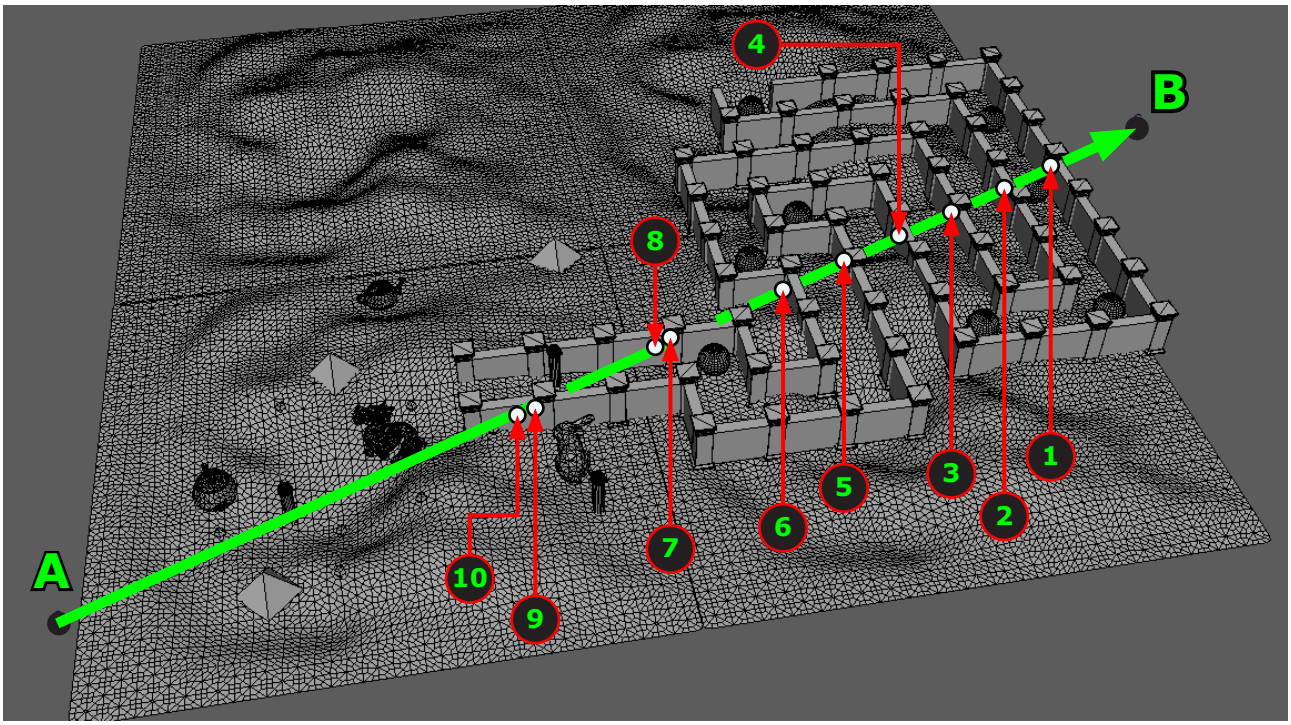

Figure 3.3: The camera is positioned at A, looking towards B. All these 10 pixel fragments successfully passed the depth culling test during TC-06, and were fully computed in succession. In TC-05, due to front-to-back ordering, only the pixel fragment #10 passed, and the others were skipped and discarded.

02. This is to be expected by now, and again to be attributed to TC-05's front-to-back sorting prior to rendering, since TC-02 and TC-05 are otherwise identical. In general, an unsorted rendering order is expected to achieve a speed between those achieved by front-to-back and back-to-front rendering orders, respectively – as is the case between TC-02, TC-05 and TC-06 in the tests conducted here. As will be shown subsequently, a situation where this is not always the case is when front-to-back sorting is combined with instancing. This will be analyzed in section 3.4, along with additional test cases demonstrating the effect.

It is clear from the above analysis that the minimal overdraw achieved through front-to-back sorting makes this distance-based sorting approach preferable, and that it is more efficient than simply employing unsorted rendering. Also, it has been demonstrated that a back-to-front rendering order serves as a performance inhibition due to its maximizing overdraw.

3.4 Instancing

Instancing is a GPU rendering mode that allows instructing the GPU to render a *batch* of objects, instead of one, through the performance of a single draw call (see Appendix B.7). It requires utilizing an additional type of buffer that contains per-instance data, which the GPU uses in order to *instantiate* the drawn object in different configurations (ibid.). In situations where 3D objects are used multiple times in a scene, it can be an effective way to reduce draw calls and the overhead they involve. Its usage is well documented (MSDN, 2016; Luna, 2012; Fernando and Pharr, 2005; and others) so it is unnecessary to elaborate more regarding its particulars here. Instead, at this point we will consider a number of test cases to determine the performance benefits that instancing can provide to the data set used in this study. The effects of *combining* instancing with previously discussed techniques will also be investigated.

Test case analysis

	TC-02	**TC-08**
Draw calls/frame	97	9
FPS	3676	4193
Avg. frame time (ms)	0.271727	0.238032

Table 3.4: TC-02 and TC-08 comparison.

Through instancing, TC-08 manages to reduce the required number of draw calls dramatically, lowering them by **90.72%** compared to TC-02 (Table 3.4, Fig. 3.4). It also achieves a rendering speed increase of **12.4%** compared to TC-02. This degree of achieved performance improvement naturally depends on the content of the given scene, which lent itself to instancing because most objects were utilized multiple times. Instancing efficiency increases when more objects can be batched into a single draw call, as is the case here. It is however very common to have 3D scenes that can benefit from employing instancing, due to the reasons discussed in section 2.2.1, and the fact that 3D assets are commonly designed with modularity and reusability as guides.

An additional point to be made regarding TC-08, is that it is slightly more efficient than TC-11, as Table 3.5 demonstrates. This is one of the cases where front-to-back sorting can have worse performance than unsorted rendering, as noted in section

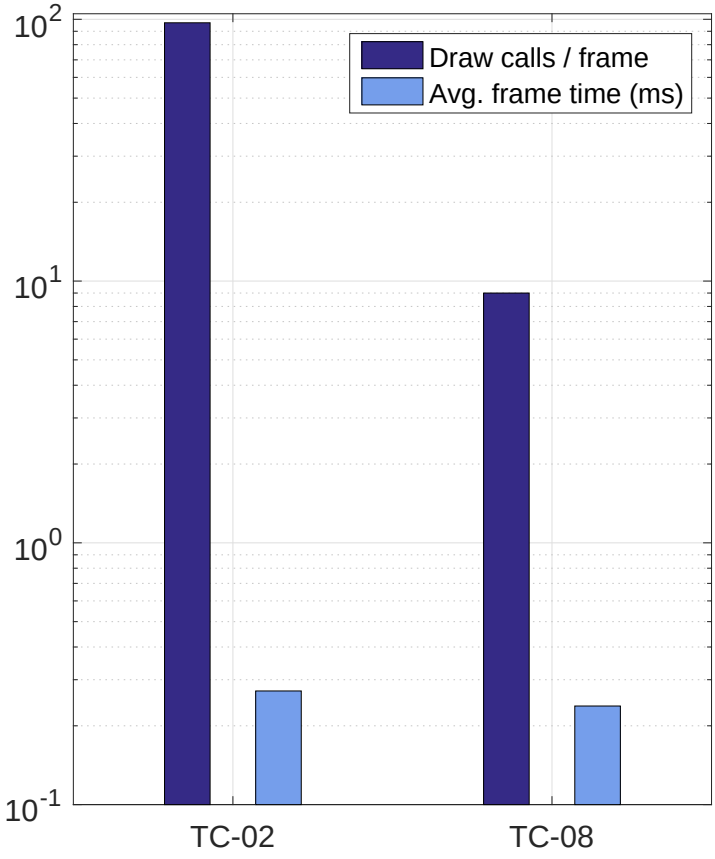

Figure 3.4: Comparison between TC-02 and TC-08. TC-08 demonstrates a 12.4% rendering speed increase compared to TC-02. This is to be attributed to the use of instancing, which lowered required draw calls by 90.72%.

3.3. It is thus time to re-examine the effect of different distance-based sorting methods when used in conjunction with instancing.

	TC-08	**TC-11**	**TC-12**
FPS	4193	4163	3975
Avg. frame time (ms)	0.238032	0.239366	0.251118

Table 3.5: TC-08, TC-11 and TC-12 comparison.

TC-11 employs instancing, and then sorts all instance batches internally in front-to-back order. TC-12 does the same, but instead uses a back-to-front order. TC-08 employs instancing with no distance-based sorting applied. As expected, TC-12 is suboptimal due to excessive overdraw, and is **4.9%** slower than TC-11. TC-11 does

not have the optimal performance among these three test cases however, contrary to what would be expected after what was treated in section 3.3. Instead, front-to-back sorted TC-11 has a **0.56%** *slower* rendering speed compared to the unsorted TC-08, which is the most efficient among the three in this case.

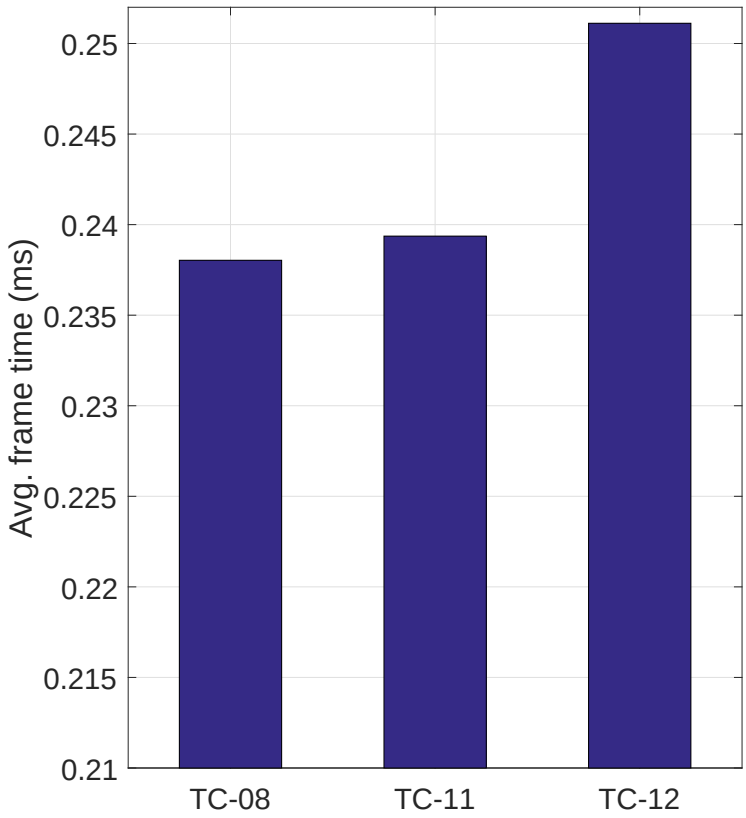

Figure 3.5: Comparison between TC-08, TC-11 and TC-12. Due to the effect of "instancing-induced overdraw", the unsorted TC-08 achieved slightly better performance than the front-to-back sorted TC-11. Back-to-front sorted TC-12 is expectedly suboptimal, due to excessive overdraw.

The reason for the reduced performance of TC-11 is the *introduction* of overdraw due to instancing itself. This "instancing-induced overdraw" effect is explained below.

Instancing-induced overdraw

Instancing requires objects to be batched based on the criterion of being able to be rendered together in a single draw call. The objects may or may not be sorted *inside* the batch, but in all cases the batches remain independent of one another, and are sequentially rendered. Hence, even after internal sorting, a subsequent

batch can *overwrite* pixels drawn by a previous batch, thus introducing overdraw. This is illustrated with an example in Fig. 3.6.

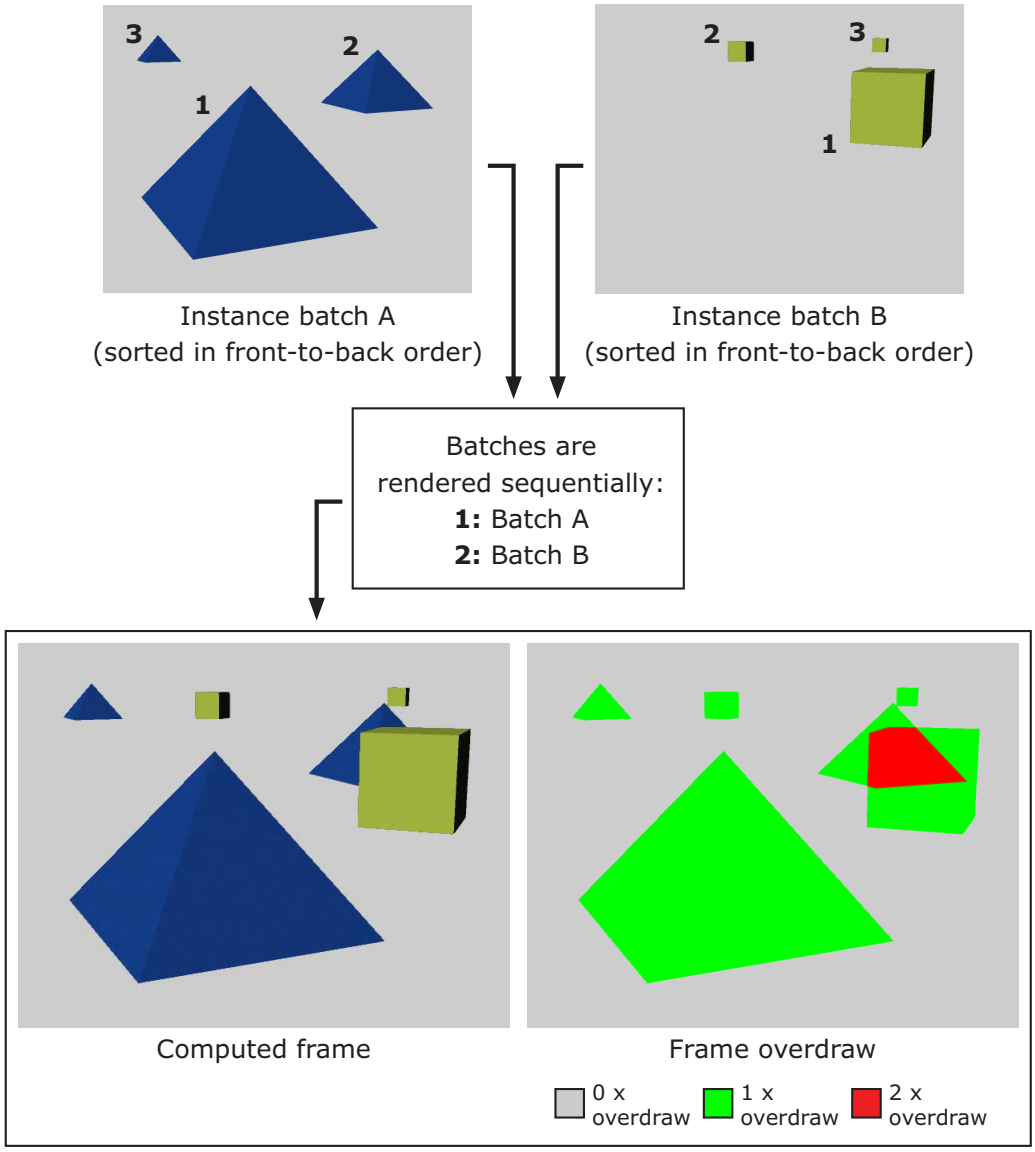

Figure 3.6: This example scene is composed of two batches, A and B, internally sorted in front-to-back order. The batches get sequentially rendered, with A first, and B second. The order the objects get rendered in is A:1, A:2, A:3, B:1, B:2, B:3. Even though the batches are internally sorted in front-to-back order, object B:1 overwrites some of A:2's pixels, because B:1 is closer to the camera than A:2 is. This overdraw could have been avoided if instancing was not used. Instancing is very effective for improving performance however, so this type of overdraw is acceptable.

Due to this effect, TC-11 and TC-12 both have overdraw that is introduced by instancing. In particular, this partially reduces TC-11's speed, since front-to-back sorting can only be applied relative to each batch, rather than to the scene contents as a whole. It is this reduction that makes TC-11 slightly less efficient than

the unsorted TC-08 in this case. However, the *combined* effect of instancing and front-to-back sorting is generally more effective than using any of these techniques individually (as has been done in TC-05 and TC-08 respectively). The reason for this is the following:

- The order resulting from distance-based sorting is dependent on the camera's relative position to the scene objects. As the camera and/or objects move, the objects get re-sorted accordingly on a per-batch basis. Hence, the result of sorting is highly variable through the progression of a typical application session. This makes situations like TC-11 transient, whenever they may occur.
- The alternative of using unsorted instancing is naturally unpredictable and equally likely to result in performance such as that demonstrated by the back-to-front sorted TC-12 (see Fig. 3.5), which is clearly suboptimal.

Hence, in the majority of cases, using both front-to-back sorting and instancing will be optimal, and even in cases when front-to-back sorting will introduce a degree of inefficiency, the effect will be slight (-0.56% in the case of TC-11) and dependent on the current positions of the camera and the objects of the scene.

3.5 Frame limiter

As discussed in Appendix B.5, a real-time rendering application involves a main loop structure that renders frames continuously, throughout the application session. The achieved framerate must comply with the minimum of 24 frames per second in order for the output to meet the persistence of vision limit and qualify as real-time (Möller, Haines and Hoffman, 2008; Poynton, 1996).

The idea behind the optimization suggested in this section is the following: Contemporary systems are quite capable of maintaining very high framerates for solid 3D rendering of typical complexity data sets, and have been for some time now (see section 1.3.1). This leads to the observation that it is very likely that the framerate achieved by a real-time renderer will – at least at times – be exponentially higher than the 24 FPS minimum limit. It is suggested by the literature that higher FPS limits are desirable for interactive rendering that does not cause exertion to the user, such as 60 FPS (Möller, Haines and Hoffman, 2008). However, from approx. 75 FPS

and upward, the eye can no longer determine any display differences (ibid.). Hence, when rendering at framerates considerably higher than this, a large portion of the invested computational effort is unnecessary, and effectively wasted. Additionally, in many situations, after first being fully computed, such superfluous frames eventually get discarded because the monitor used as output cannot support such a high refresh rate. Hence, it is suggested that, by actively limiting the maximum frames rendered for a given second through explicitly assigning and enforcing an *upper* limit, that wasted computational effort can be avoided, and instead redirected to performing other tasks.

Test case analysis

As all test cases discussed so far demonstrate (also see Appendix A), the achieved FPS for the tested data set range between 3000 to 4000. This corroborates what was discussed above, indicating that there is much room for optimization in reducing such an unnecessarily high frame rate.

In consequence of this behaviour, a "frame limiter" component was developed to be used in conjunction with the A2 algorithm. It is cursorily noted that this frame limiter is entirely independent from specifying DirectX API's `DXGI_SWAP_CHAIN_DESC::BufferDesc.RefreshRate` during swap chain creation, although naturally this too should correspond to the desired frame limit.

When the frame limiter is enabled, the system no longer allows the unconditional performance of a render, but first checks the elapsed time since the last rendering of a frame (Fig. 3.7 and 3.8), and only renders the frame if the specified *frame limit* is not exceeded. For example, for achieving and maintaining a maximum frame rate of 60 FPS, this would require that a render should occur *if and only if* the elapsed time `dt` satisfies the following condition:

```
if(dt >= (float)(1/60)){RenderScene();}
```

This could be further generalized for any desired frame rate `n`:

```
int n=60; // desired maximum limit of frames per second
if(dt >= (float)(1/n)){RenderScene();}
```

At this point, two test cases are considered to observe the effects of this frame limiter on the application: TC-02, as discussed in section 3.1, only employs a collective vertex/index buffer. TC-03 performs the same computations as TC-02, except for

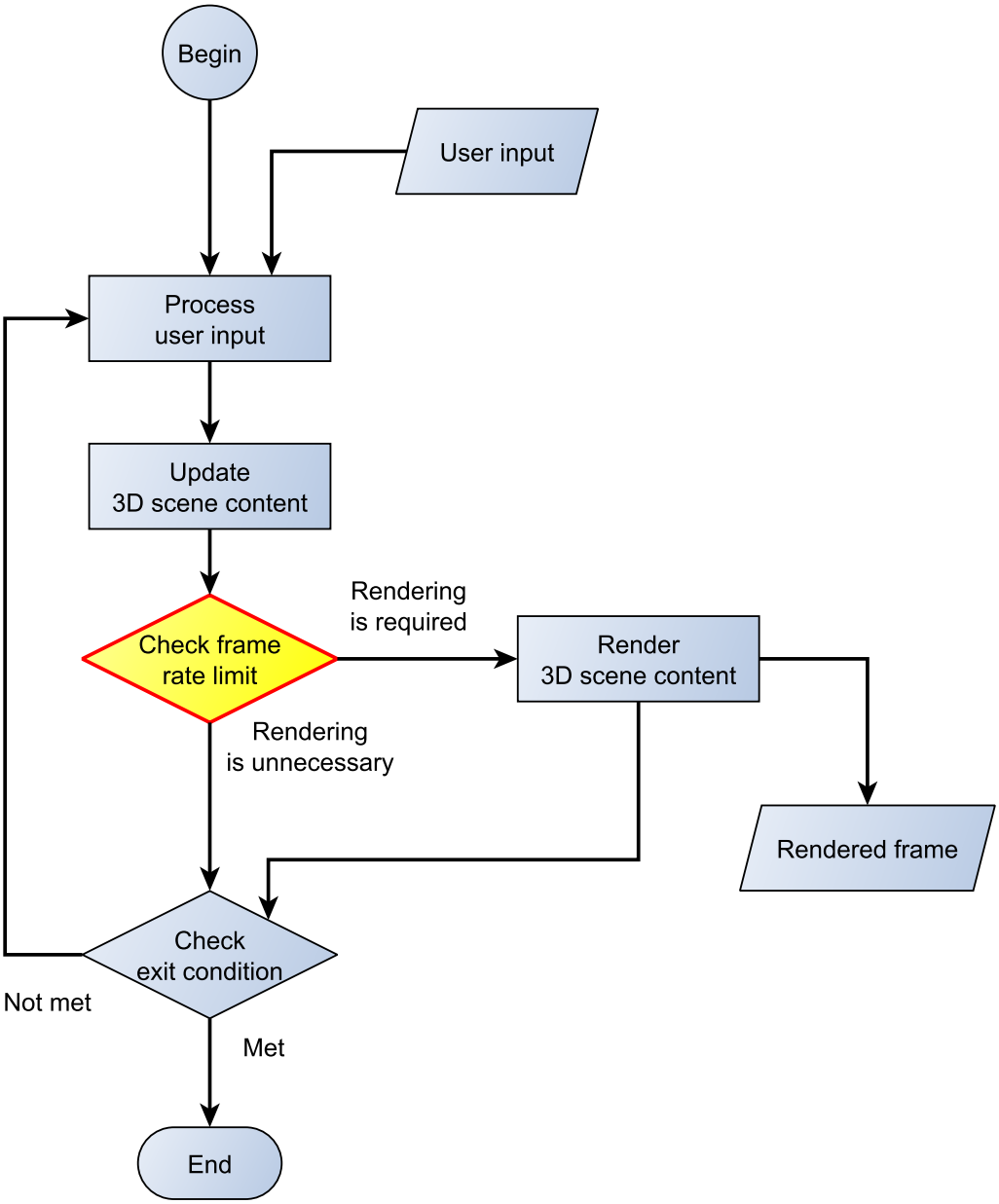

Figure 3.7: A flowchart summarizing the effect of adding the frame limiter to the application's main loop (compare with Fig. B.2). Instead of unconditionally rendering, the frame rate limit is first checked, allowing skipping of unnecessary frames.

```
bool bExitConditionMet = false;
float dt; //elapsed time (in seconds) since RenderScene() was last called
int n = 60; // desired maximum limit of frames per second

while ( bExitConditionMet != true )
{
      //process user input
      //...

      //update scene content, including dt
      //...

      if ( dt >= (float)(1/n) ) //conditionally render the scene
      {
            RenderScene(); //render frame
      }
}
```

Figure 3.8: A short code segment representing the main loop with the frame limiter added, corresponding to the flowchart of Fig. 3.7.

the addition of the frame limiter.

	TC-02	**TC-03**
FPS	3676	60
Iterations per sec.	3676	32792276
Avg. frame time (ms)	0.271727	0.303235

Table 3.6: TC-02 and TC-03 comparison.

Table 3.6 indicates that by limiting the framerate to the maximum perceptible rate of 60, the responsiveness of the application (represented by the iterations per second) increased by more than **8920** times (Fig. 3.9). Visually, this makes no difference in the perceived fluency of the animation, although the significant processing power conserved in this way can be applied to other useful tasks, increasing the efficiency of the application, and possibly its functionality as well.

A very interesting aspect of TC-03 is that it has a decreased rendering speed of **11.59%** compared to TC-02. This apparent "anomaly" may initially appear like an erroneous measurement, considering that the only thing modified between TC-02 and TC-03 is the frequency of issuing the draw calls. It has been resilient to scrutiny and test repetition however. TC-08 and TC-09 also initially may seem to suggest the "anomaly" is likely due to an error, since the effect of the frame limiter appears to

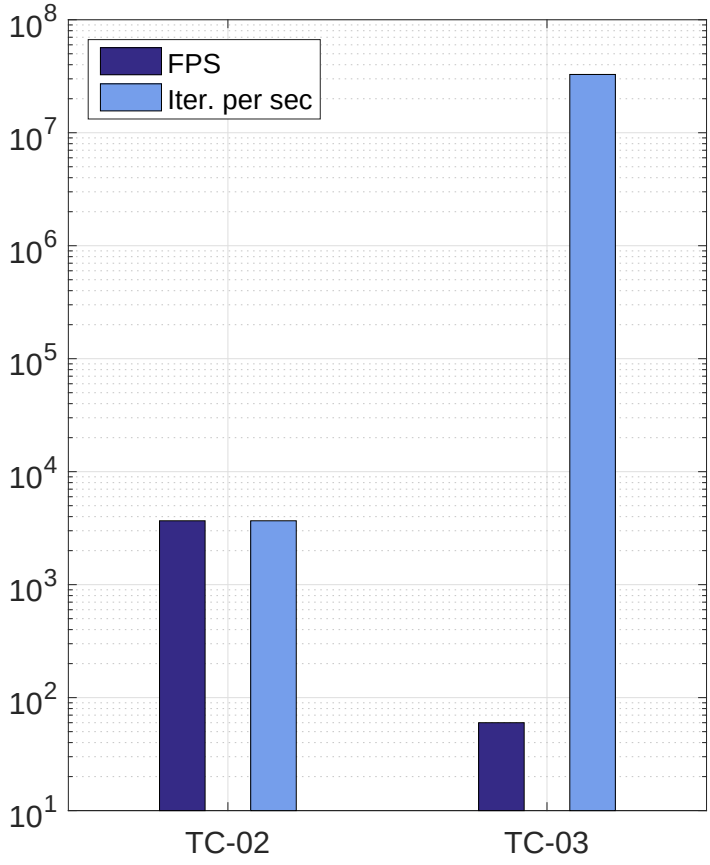

Figure 3.9: A chart demonstrating how the skipping of unnecessary frames in TC-03 exponentially boosts system responsiveness.

clearly increase rendering speed (specifically, by **32.83%**) when used in conjunction with instancing (see Table 3.7 and Fig. 3.10).

	TC-08	**TC-09**
FPS	4193	60
Avg. frame time (ms)	0.238032	0.159883

Table 3.7: TC-08 and TC-09 comparison.

To investigate the causes of TC-03's decreased speed further, let us examine some additional tests demonstrating the effect that modifying the frame limit (originally set at 60 FPS, both for TC-03 and TC-09) has on the application's performance (Fig. 3.11). These tests are listed in Table 3.8 below.

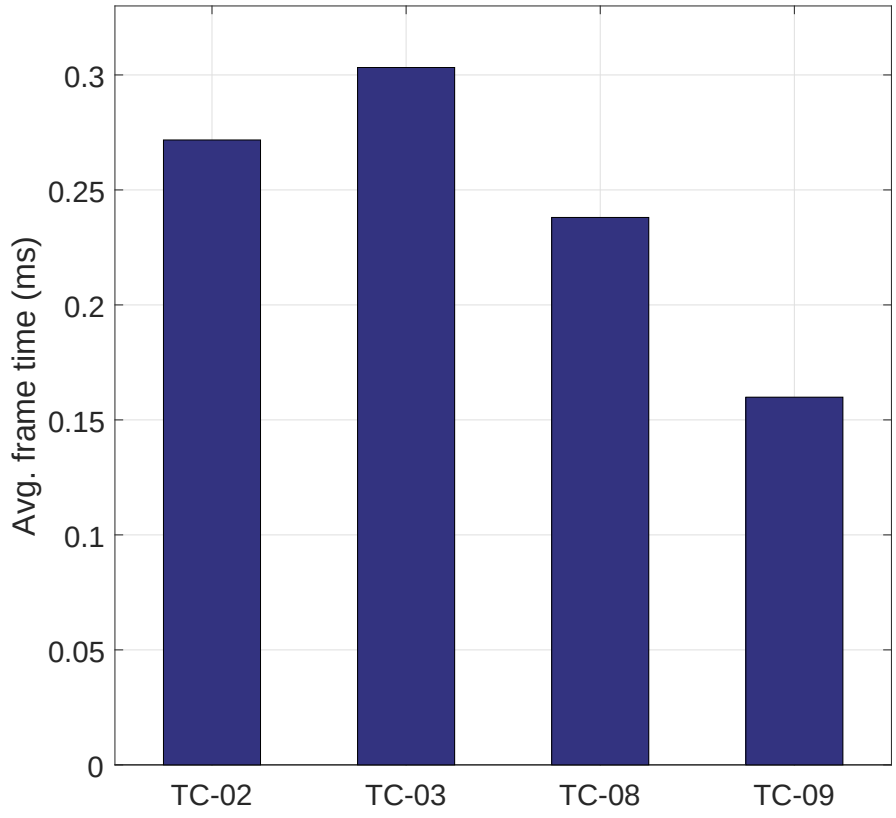

Figure 3.10: Chart comparing TC-02, TC-03, TC-08 and TC-09, and the apparently anomalous speed decrease of TC-03 compared to TC-02.

Frame limit	**Avg. frame time (ms)**	**Instancing enabled**	**Draw calls/frame**
5	0.425524	No	97
20	0.347363	No	97
60	0.303235	No	97
200	0.298149	No	97
1000	0.297752	No	97
5	0.259747	Yes	9
20	0.179845	Yes	9
60	0.159883	Yes	9
200	0.14692	Yes	9
1000	0.137616	Yes	9

Table 3.8: Additional tests demonstrating the effect of modifying the frame limit.

These additional tests greatly clarify the relationship between TC-02, TC-03, TC-

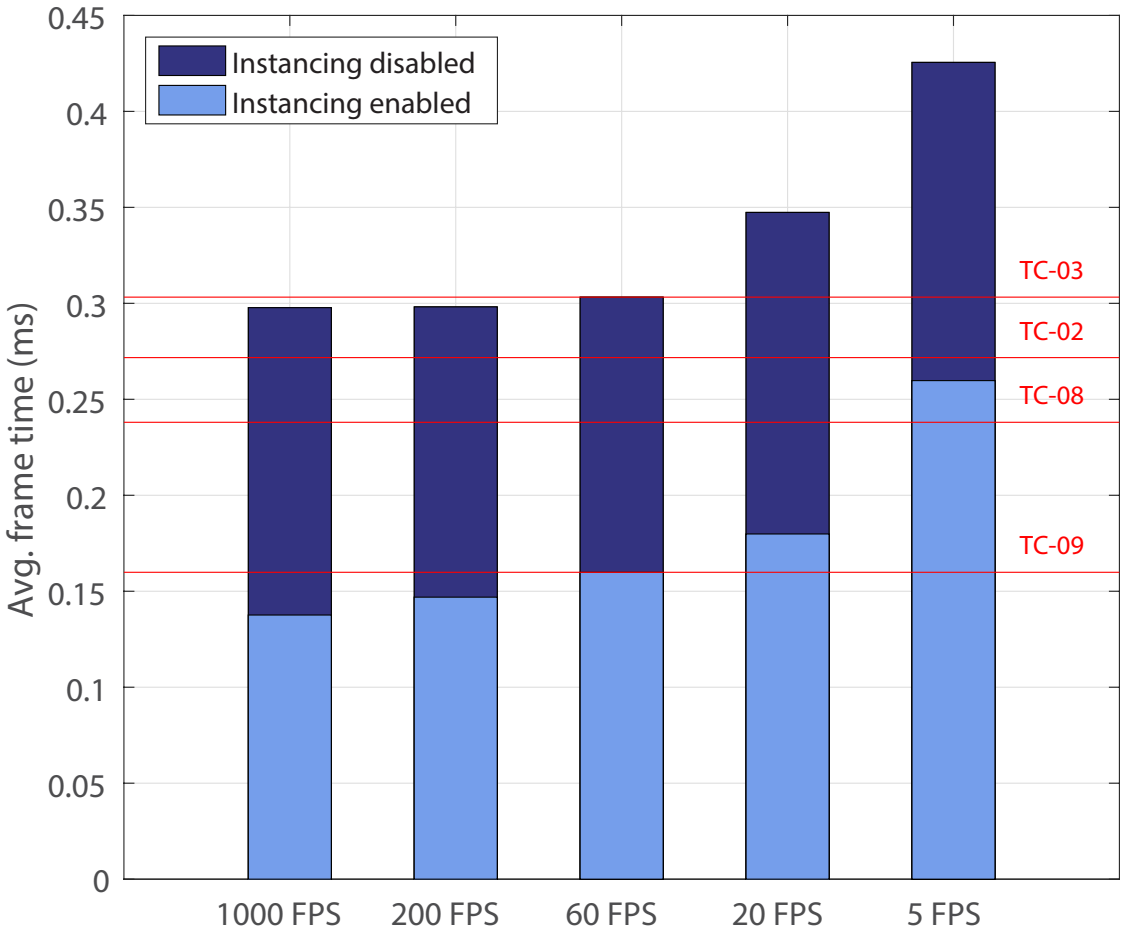

Figure 3.11: Chart demonstrating the effect adjusting the frame limit has on rendering speed. Both with instancing enabled and disabled, a very regular proportional decrease in rendering speed is observed as the frame limit decreases.

08 and TC-09. What Table 3.8 and Fig. 3.11 demonstrate is a very regular and hardly anomalous effect originally hidden in the frame rendering speed relationship observed between these test cases in Fig. 3.10. Apparently, the frame limit specified has a strong effect on rendering speed, causing it to proportionally decrease as the frame limit *decreases*.

Furthermore, the following observations can be made at this point:

1. The only factor that changes in these tests is the *frequency* of issuing draw calls

2. The great difference in the number of draw calls (**90.72%** less) between en-

abling and disabling instancing does not seem to significantly affect the pattern's regularity, but only offset it along the Y-axis (Fig. 3.11)

Based on these observations, the author's interpretation of this behaviour is that the effect observed is likely related to cache misses. Most likely, increased elapsed time between rendering a frame makes the GPU more liable to subsequent cache misses because the application's GPU data is gradually replaced by other data in cache memory.

Having discussed the indirect effects of applying and adjusting the upper frame limit, let us now conclude this section by assessing the frame limiter's overall efficacy in improving performance. The frame limiter has a very significant influence in response time (Fig. 3.9) which makes it clearly one of the most effective techniques discussed in this study. The effect it has on rendering speed is proportional to the specified frame limit, making higher frame limits render faster than lower ones. After considering that **a)** the lowest possible framerate enabling real-time rendering is 24 FPS, **b)** 60 FPS is desirable for interactivity, and **c)** ~75 FPS is the upper limit that is visually distinguishable, the rendering speed decrease caused by the limiter is not particularly significant in this range, but the benefits in increased response time are quite noteworthy (Fig. 3.9). Furthermore, as demonstrated by TC-09, at 60 FPS, the frame limiter is actually beneficial to rendering speed as long as it is combined with instancing (discussed in section 3.4). Hence, the effectiveness of the frame limiter is clearly demonstrated here.

3.6 Pre-transformed immobile objects

In a real-time rendering context, 3D transformation calculations are normally performed on the GPU (MSDN, 2016; Luna, 2012), and usually in the vertex shader program (see Appendix B.8). Such calculations are fairly computationally intensive, involving 4x4 square matrix multiplications, and represent a large portion of the overall computational load of the 3D rendering process. Through them, a 3D object's coordinates get transformed from Local Space (see Appendix B.13) to World, View, Projection, and eventually reach all the way to Window Space for it to get rendered. This process is repeated every time a frame is rendered.

It is often the case however, that a 3D scene contains objects that are strictly immobile. Common examples of such objects in a hypothetical urban scene can

include the ground or architectural elements. When *known* to be strictly immobile, the World Space coordinates of such objects are guaranteed to never change and could be treated as constant. This can eliminate their per-frame World-Space transformation, which can alleviate some of the computational overhead that 3D transformations involve.

What is proposed in this section is to treat strictly immobile objects separately from potentially mobile ones within the rendering system: Immobile objects are explicitly specified as such during the creation/design of the 3D scene's content. They are pre-transformed *once* to World Space at application initialization. In contrast, potentially mobile objects are in Local Space, getting transformed to World Space on a per-frame basis as normal. For the remainder of this section, strictly immobile objects will be referred to as "pre-WS" ("pre- World-Space") objects, and potentially mobile ones as "OS" ("Local/Object Space") objects.

In the optimized algorithm A2 (see section 2.8.1), "pre-WS" transformation is implemented in the following manner: The 3D object is initially loaded by the CPU. Once determined that it is "pre-WS", it gets transformed *once* to World Space by the CPU. Then the object (now already in World Space) is copied to the GPU and a "pre-WS"-specific vertex shader is used when rendering such objects. Instead of performing the transformation from Local to World Space, the "pre-WS" vertex shader skips this computation and treats the data it has as being already in World Space. See section 2.8.1 and Figures 2.5 and 2.6, which can complement this description and offer a diagrammatic overview of these features.

As noted, A2 uses two separate vertex shaders to handle "pre-WS" and "OS" objects respectively. An alternative approach possibly suggested is to use a *single* vertex shader with a conditional World Space transformation, e.g. through passing a boolean flag representing immobility to the vertex shader. There are some points to make in favor of using separate shaders however:

- To make a boolean flag accessible to the shader, this would likely require GPU constant buffer updating (see Appendix B.7 and section 3.2) to copy the boolean value to the GPU on a per-frame and per-object basis, so that the shader can examine it[1]. Remember that constant buffers are necessar-

[1]An alternative, and possibly superior approach in favor of using a boolean flag could be to embed this flag in the *instancing* buffer (i.e. the vertex buffer containing per-instance data). This would require no additional buffer updates apart from updating the instancing buffer itself prior to rendering the frame. This was not tested in this study, although it would likely perform even better than A2's current pre-WS implementation of separate shaders does.

ily updated in full (see section 3.2), so this boolean flag makes full per-object constant buffer updates *required*. By contrast, the separate shader approach allows potentially skipping per-object constant buffer updates altogether (provided no other required data is stored in that constant buffer). This update skipping is employed in the A2 algorithm.

- Using additional shaders requires additional API calls per frame in order to sequentially bind those shaders to the pipeline. This is an overhead that A2's pre-WS implementation does involve, although batching of pre-WS objects is also performed, so that the shader is switched only *once* per frame for all specified pre-WS objects, thus minimizing the overhead.

- Shader programs have additional limitations, such as the number of allowable nesting levels and the total number of contained instructions (MSDN, n.d. a). It is generally suggested to keep them as short and simple as possible (McDonald, 2012), and to avoid unnecessarily complex and overly-conditional flows (ibid.). Given that a system's requirements may necessitate shaders of already reasonably high complexity, it may be beneficial to avoid adding even more conditional execution.

- Shaders are very small programs, occupying comparatively little GPU memory (MSDN, 2016; Luna, 2012; McDonald, 2012). Hence, code reduplication for shaders is generally a negligible issue, provided it is conscientiously employed.

Hence, the main advantage offered by the use of a separate shader is the flexibility of optionally skipping per-object constant buffer updates. This introduces the overhead of re-binding shaders, which A2 minimizes by batching pre-WS objects separately, as noted above.

Before beginning the test case analysis, it is important to also mention some limitations of pre-transforming immobile objects:

1. Since objects are treated as *already* in World-Space, this makes them not easily instantiated more than once in a given scene (note that the "instantiation" referred to here is unrelated to using GPU instancing as described in section 3.4). Hence, strictly immobile objects that are also desired to be multiply instantiated (like a modular architectural element, e.g. a column) cannot benefit from this technique, because additional instantiations would require retrans-

formation in World Space. This is possible of course, although nullifies the benefits and purpose of employing pre-transformation in the first place.

2. For pre-transformation to be effective, the added cost of employing it must be less than the cost of actually computing those immobile objects' World-Space transformations every frame. This makes pre-WS usage inefficient when applied to a small portion of the total scene's polygon count (see the test case analysis for exact percentages tested).

Considering that 3D assets are usually designed with modularity and reusability as guides (see section 2.2.1), the inability to instantiate pre-WS objects multiple times certainly limits their applicability. Even so, this technique can prove useful in usage scenarios where a scene contains a moderate portion (in terms of total polygon count) of single-instantiation, strictly immobile objects.

Test case analysis

The test data set employed in these tests (described in section 2.2.1) contains two objects intentionally added for testing pre-transformation in World Space. These are the single instantiations of the Stanford bunny and Stanford dragon models (Stanford Computer Graphics Laboratory, 2017), both reconstructed to a lower polygon count for this study. Having a total triangle count of 61906 triangles, they amount to **35.98%** of the total triangle count of the tested scene. This ratio proved sufficient to demonstrate clear efficiency with this technique and is utilized by the test cases discussed below and listed in Table 3.9.

	TC-02	**TC-07**	**TC-08**	**TC-13**
FPS	3676	3775	4193	4221
Avg. frame time (ms)	0.271727	0.264411	0.238032	0.23638

Table 3.9: Comparison of TC-02, TC07, TC-08 and TC-13.

Four test cases are examined here. TC-02 which employs only collective vertex/index buffers (originally mentioned in section 3.1), TC-07 which is identical to TC-02 but also uses pre-WS objects, TC-08 which uses collective vertex/index buffers and instancing (originally mentioned in section 3.4), and TC-13 which is identical to TC-08 but also uses pre-WS objects.

As Table 3.9 and Fig. 3.12 demonstrate, pre-WS objects have proven effective both with and without being combined with instancing. When used without instancing,

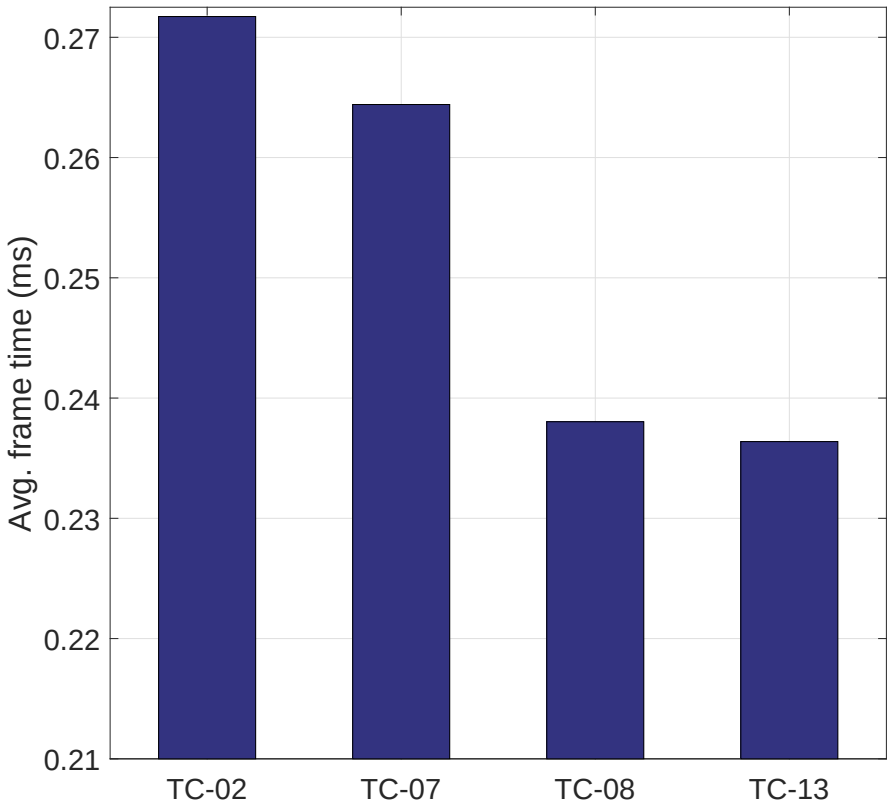

Figure 3.12: Chart demonstrating the effect of pre-WS objects usage on rendering speed. Both with instancing disabled (TC-07) and enabled (TC-13), the achieved rendering speed is higher than without using pre-WS (TC-02 and TC-08).

a **2.69%** rendering speed increase was observed (by comparing TC-02 and TC-07). When used with instancing, this technique was still slightly more efficient, with a rendering speed increase of **0.69%** (by comparing TC-08 and TC-13).

Hence, the above demonstrates that provided the limitations of pre-WS objects are not prohibitive, employing this technique can increase rendering performance. It is also repeated here that the above test cases involve a 3D data set where pre-WS objects correspond to **35.98%** of the total triangle count of the scene, and therefore data sets with an even higher percentage of such immobile objects would benefit proportionally more by this technique.

On the other hand, from the above it follows that data sets with a progressively lower percentage of immobile objects would benefit progressively less, until employing pre-transformation would provide no performance benefits at all. In connection with this observation, it is noted that initial measurements performed for this study

tested pre-transformation with lower triangle count versions of the Standford bunny and dragon, which only amounted to **15.78%** of the total triangles of the scene. This percentage was low enough to make pre-transformation of immobile objects partly inefficient: It resulted to a rendering speed increase of **3%** for TC-07 compared to TC-02, but to a **1.098%** *decrease* for TC-13 compared to TC-08.

4. Conclusions

This study has successfully demonstrated the following:

- Using collective vertex/index buffers is more efficient than distributing data in more buffers of this type. In the performance optimization tests conducted, the use of collective vertex/index buffers improved rendering speed by **6.46%**, as demonstrated in section 3.1.
- A small number of constant buffers determined by the frequency of updates of the contained data is more efficient than using a collective constant buffer. The conducted tests demonstrated a **3.26%** increase in rendering speed when this was applied, as discussed in section 3.2.
- Sorting opaque objects to be rendered in front-to-back order relative to the camera position improves rendering speed by reducing overdraw. Back-to-front sorting has the opposite effect. With front-to-back sorting, the conducted tests indicated a **2.78%** increase in speed compared to back-to-front sorting, as discussed in section 3.3.
- Use of the GPU's instancing rendering mode has a very strong effect in increasing rendering speed due to the reduction of API draw calls required. In the conducted tests, its use increased frame rendering speed by **12.4%**, as discussed in section 3.4.
- When combined with instancing, front-to-back sorting can be less effective due to being applied on a per-batch basis. It is still generally advised to use both front-to-back sorting and instancing simultaneously if possible. This is discussed in section 3.4.
- Rendering frames with a rate higher than the viewer can distinguish is suboptimal. Instead, a frame limiter should be employed, with an upper limit no higher than 75 FPS. This has no visual difference for the viewer, but skipping unnecessary frames can exponentially improve system response time. In the tests performed, using a frame limiter with a limit of 60 FPS increased response time by more than **8920** times, as discussed in section 3.5.
- The frame limit specified when using a frame limiter has a proportional influence to frame-rendering speed. Hence, lower frame limits result in lower

rendering speed. In the conducted tests and with a limit of 60 FPS, use of the limiter demonstrated a **11.59%** decrease to rendering speed when instancing was disabled. When instancing was also enabled however, the limiter instead had the effect of a **32.83%** *increase* in rendering speed. This behaviour is discussed in section 3.5.

- When the rendered scene contains objects that are strictly immobile, it can be efficient to pre-transform them and treat them separately than potentially mobile objects, even if this leads to using a small number of additional shaders and API calls. This technique does not allow those immobile objects to be instantiated multiple times, but can be effective if they are high-polygon-count, single-instantiation objects. In the tests conducted, employing such immobile objects comprising **35.98%** of the scene's total polygon count brought about a **2.69%** increase in rendering speed when GPU instancing was disabled and **0.69%** when it was enabled. This is discussed in section 3.6.

By cumulatively employing all optimizations treated in this study, the conducted performance tests demonstrated a frame rendering speed increase of **55.04%** and a responsiveness increase by **9637** times (Fig. 4.1 and 4.2).

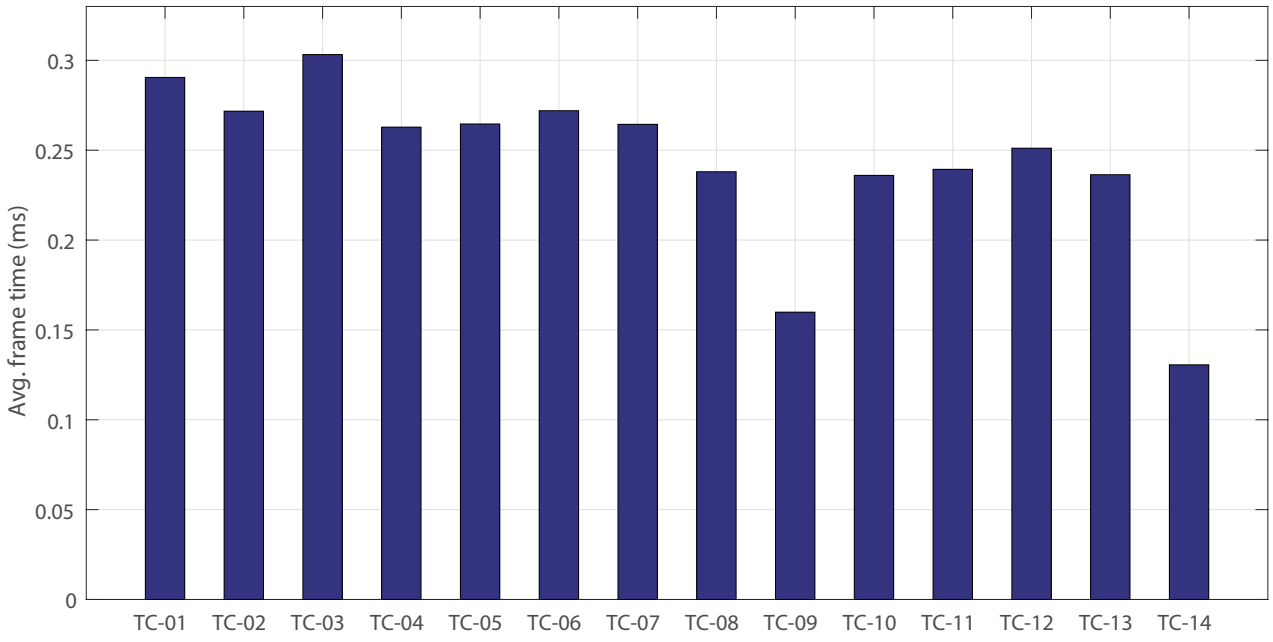

Figure 4.1: Collective chart comparing the rendering speed of all test cases examined. TC-14, employing all optimizations discussed, demonstrates a 55.04% speed increase compared to TC-01, which is the initial configuration.

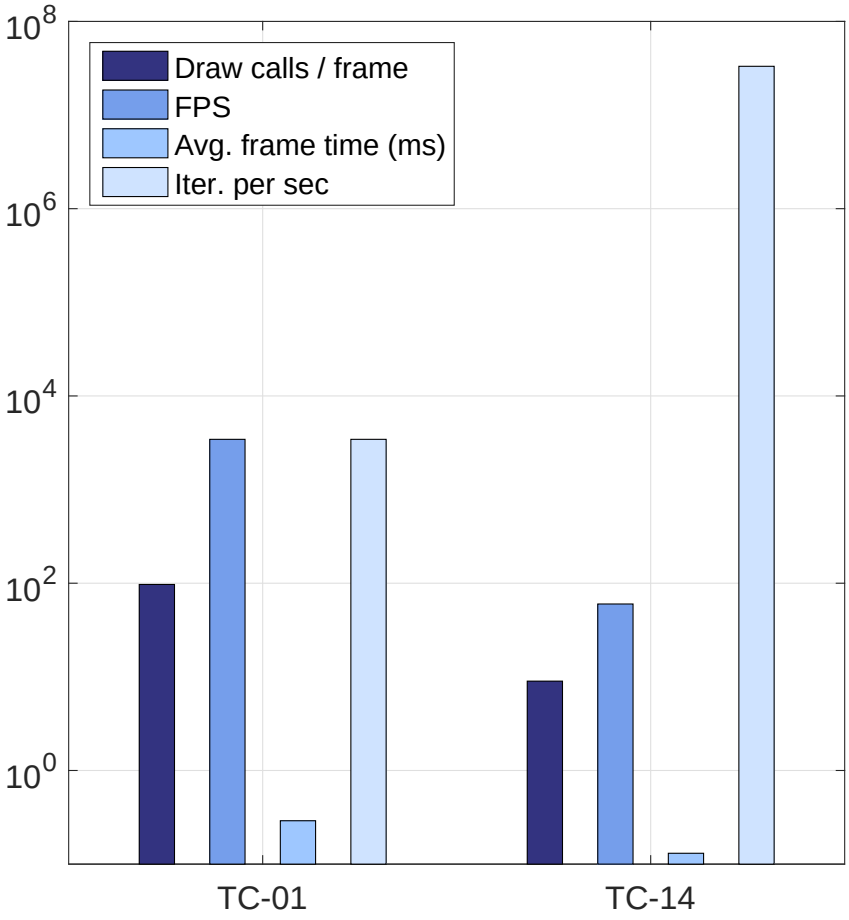

Figure 4.2: Collective chart summarizing the study's results by comparing the initial configuration TC-01 with the optimized TC-14. TC-14 demonstrates a reduction in draw calls, an adhered-to upper FPS limit of 60, a 55.04% rendering speed increase and a response time increase by 9637 times.

5. Further research

There are several areas where additional research could be performed after what has been demonstrated through the current study. Some important areas worth considering are the following:

- **Multithreading:** This study employed a single-threaded testing environment in order to somewhat narrow the scope of the test scripts and the variety of the test cases to be analyzed. Concurrency is most commonly employed in contemporary systems, however. A multi-threaded implementation can allow much higher efficiency, such as by allowing user input to be processed in a separate thread, or by performing multi-threaded rendering (Fig. 5.1).

- **Unified CPU-GPU memory address space:** This study intentionally focused on resource management in a system with a discrete GPU memory address space (see sections 1.3.1 and 1.4). However, it would be interesting to see the results of the same test cases applied to a system with a unified CPU-GPU memory address space – such as the ones discussed in the hardware review of section 1.3.1. Such a system would not necessitate the copying of data to and from GPU memory (see Appendix B.7), making the areas of likely overhead to differ, and thus perhaps indicate different optimization approaches.

- **LODs and occlusion culling:** Techniques like LODs (Luebke et al, 2003; Watson, Walker and Hodges, 2004) and occlusion culling (Yoon, Salomon and Manocha, 2003) – as illustrated in Figures 5.2 and 5.3 respectively – are commonly used methods for improving real-time rendering performance. They could be used in conjunction with what is discussed here in order to determine their joint efficacy and performance benefits.

- **Triangle reordering:** The pre-processing of 3D data prior to rendering so that the triangles composing them can be reordered (Fig. 5.4) is a subject that has been discussed in several studies (Han and Sander, 2016; Sander, Nehab and Barczak, 2007; Storsjö, 2008). The aim is to reorder the triangles so that optimal use of the GPU's vertex cache is made and a minimal amount of overdraw occurs (ibid.). Note that this refers to *self*-overdraw of the object and thus cannot be addressed by the rendering order of the objects discussed in section 3.3. Here, it is the order of the objects' *triangles* that matters. Such

techniques could be employed in conjunction with what has been discussed in this study, to further improve rendering performance.

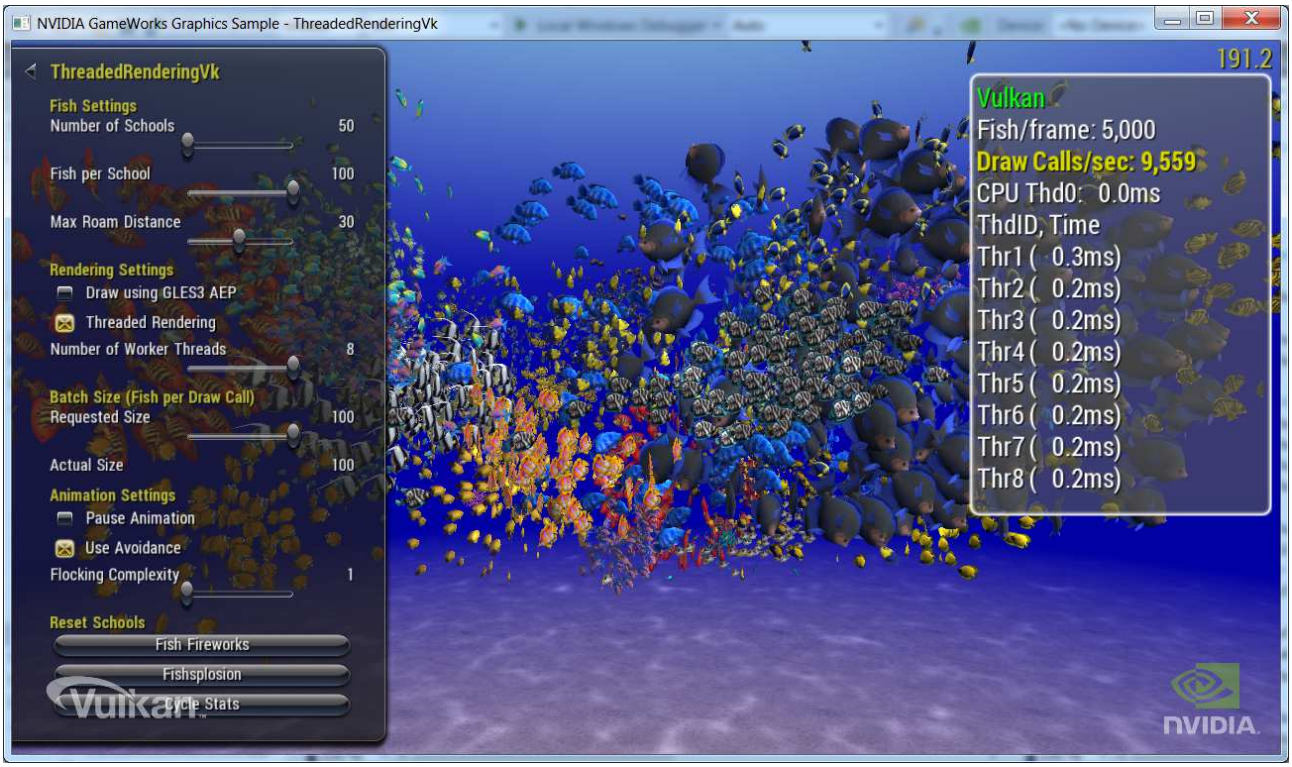

Figure 5.1: A Nvidia sample showcasing threaded rendering using Vulkan (Nvidia Corporation, 2016b).

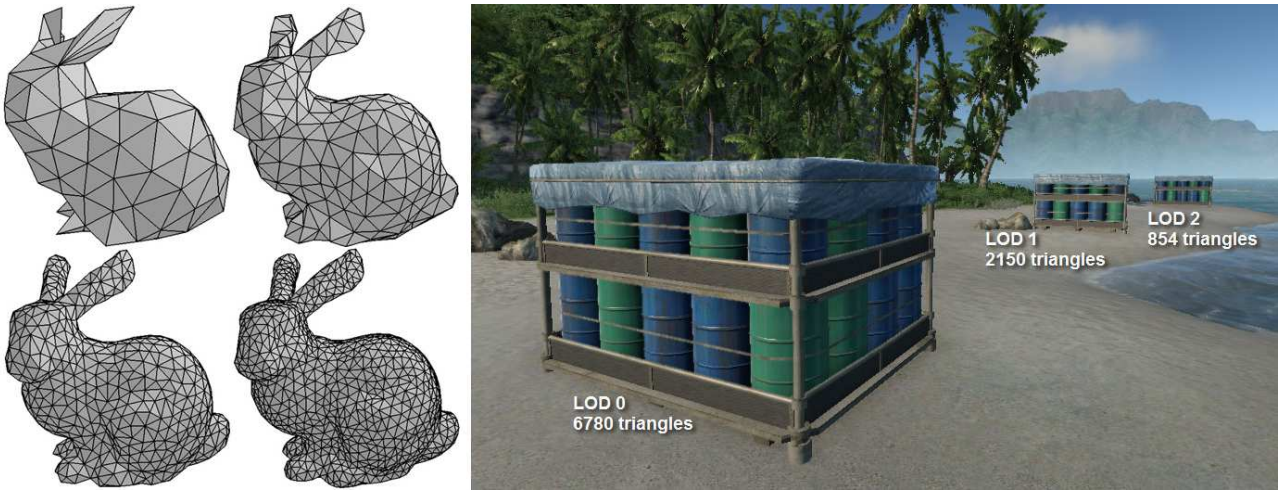

Figure 5.2: LODs ("levels of detail") can improve performance by rendering progressively lower polygon-count (and texture resolution, etc.) versions of objects when their placement indicates that the substitution will be unnoticeable to the viewer/user. Typically, heuristics are employed to control the LODs (Watson, Walker and Hodges, 2004). Left: Various possible LODs for the Stanford bunny model (Stanford Computer Graphics Laboratory, 2017). Right: Example of a chain of distance LODs for a 3D object (CRYTEK GmbH., 2016).

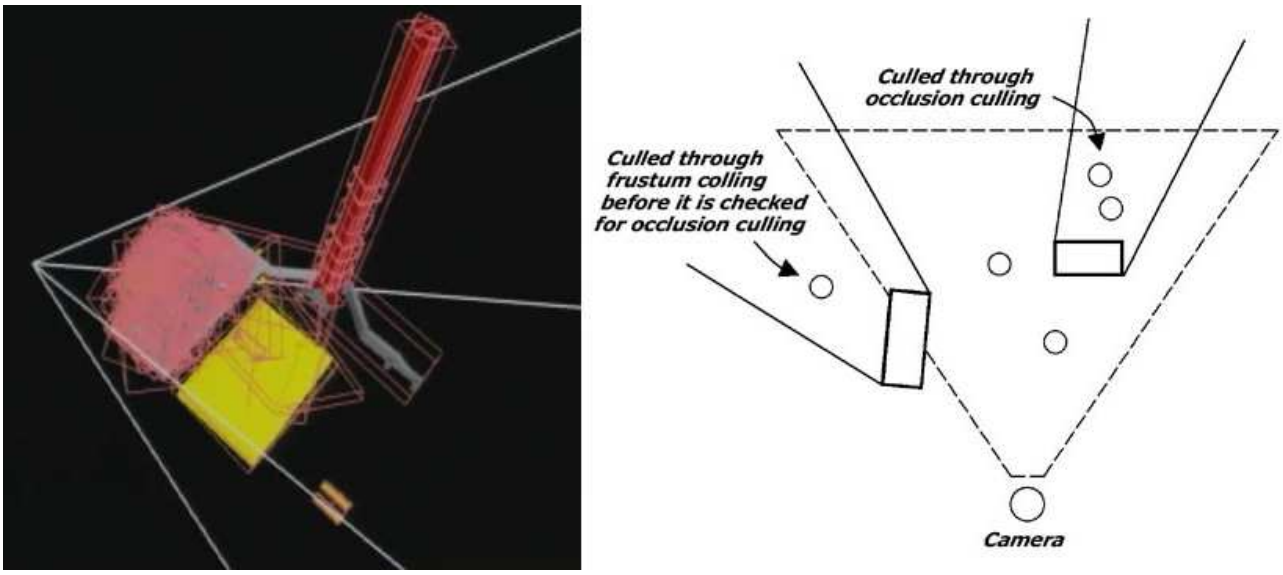

Figure 5.3: Left: An occlusion culling algorithm at work on a complex interactive 3D scene (Yoon, Salomon and Manocha, 2003). Right: Occlusion culling is concerned with objects that have passed frustum culling (and thus lie inside the view frustum) but are occluded by other objects closer to the camera (Young, 2005).

Figure 5.4: A triangle reordering study by (Sander, Nehab and Barczak, 2007). Left: The red triangles indicate areas where a vertex cache miss occurred. Right: Darker areas are portions of the frame where more overdraw occurs.

Bibliography

Reference List:

3DRT (n.d.). Dungeon Master Kit. *3DRT*. [Online]. Available from: http://3drt.com/store/environments/fantasy-environments/dungeon-master-kit.html [Accessed: 20 April 2017]

ACM SIGGRAPH (2017). *ACM SIGGRAPH*. [Online]. Available from: http://www.siggraph.org/ [Accessed: 28 March 2017]

Adams, D. (2007). Crysis Review. *IGN*. Available from: http://pc.ign.com/articles/834/834614p1.html [Accessed: 2 January 2017]

Advanced Micro Devices Inc. (2017a). *Compute Cores*. Advanced Micro Devices Inc. [Online]. Available from: http://www.amd.com/en-us/innovations/software-technologies/compute-cores [Accessed: 3 January 2017]

Advanced Micro Devices Inc. (2017b). *Capsaicin brought to you by AMD Radeon Graphics: AMD's GDC Sessions*. Advanced Micro Devices Inc. [Online]. Available from: http://www.amd.com/en-us/who-we-are/corporate-information/events/gdc [Accessed: 19 April 2017]

Allegorithmic (2017). *Substance Designer*. [Software]. Windows version. Allegorithmic.

Appel, A. (1968) Some techniques for shading machine renderings of solids. *AFIPS Conference Proc.* [Online], 32 pp.37-45. Available from: http://graphics.stanford.edu/courses/Appel.pdf [Accessed: 2 January 2017]

Bauer, S. (2015). *Graphics Processing Units - GPUs*. [Online]. Alpen-Adria-Universität Klagenfurt. Available from: https://pervasive.aau.at/BR/teaching/dsp/GPU-Sergei-Bauer.pdf [Accessed: 1 January 2017]

Bowery, J. (1974). *Spasim*. [Software]. PLATO (Programmed Logic for Automatic Teaching Operations), version 20010402. Bowery, J.

Blinn, J. and Newell, M.E. (1976). Texture and reflection in computer generated images. *Communications of the ACM* [Online], vol. 19(10), p. 542-547. Avail-

able from: http://dl.acm.org/citation.cfm?id=360349.360353 [Accessed: 1 January 2017]

Blinn, J. (1977). Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics* [Online], vol.11(2) p. 192-198. Available from: http://dl.acm.org/citation.cfm?doid=563858.563893 [Accessed: 1 January 2017]

Blinn, J. (1978). Simulation of wrinkled surfaces. *ACM SIGGRAPH Computer Graphics* [Online], vol. 12(3), p. 286-292. Available from: http://dl.acm.org/citation.cfm?id=800248.507101 [Accessed: 1 January 2017]

Blinn, J. and Kohlhase, C. (1978). *Voyager 2 encounters Jupiter: Computer Simulation*. [Video]. [Online]. NASA. Available from: https://www.youtube.com/watch?v=o4xIJlEV8Kw [Accessed: 1 January 2017]

Burgess, J. and Purkeypile, N. (2013). *Skyrim's Modular Approach to Level Design*. GDC. [Online]. Available from: http://blog.joelburgess.com/2013/04/skyrims-modular-level-design-gdc-2013.html [Accessed: 19 April 2017]

Catmull, E.E. (1974). *A subdivision algorithm for computer display of curved surfaces*. University of Utah. [Online]. Available from: http://static1.1.sqspcdn.com/static/f/552576/6419248/1270507173137/catmull_thesis.pdf [Accessed: 1 January 2017]

Core Design Ltd. (1996). *Tomb Raider*. [Software]. Windows version. Eidos Interactive.

Crystal Dynamics (2007). *Tomb Raider: Anniversary*. [Software]. Playstation 2 version. Eidos Interactive.

Crytek (2007). *Crysis*. [Software]. Windows version. Electronic Arts.

CRYTEK GmbH (2016). *Creating LODs*. CRYTEK GmbH. [Online]. Available from: http://docs.cryengine.com/display/SDKDOC2/Creating+LODs [Accessed: 20 April 2017]

Deering, M., Winner, S. , Schediwy, B., Duffy, C. and Hunt, N. (1988). The triangle processor and normal vector shader: a VLSI system for high performance graphics. *ACM SIGGRAPH Computer Graphics* [Online], vol.22(4), pp. 21-30. Available from: http://dl.acm.org/citation.cfm?doid=378456.378468 [Accessed: 3 January 2017]

Eberly, D.H. (2005). *3D Game Engine Architecture. Engineering Real-Time Applications With Wild Magic*. San Francisco: Morgan Kauffman Publishers.

Epic MegaGames, Digital Extremes and Legend Entertainment (1998). *Unreal*. [Software]. GT Interactive.

Fernando, R. ed. (2004). *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Boston: Addison-Wesley.

Fernando, R. and Pharr, M., eds. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Boston: Addison-Wesley.

GDC (2017). *GDC Vault*. GDC. [Online]. Available from: http://www.gdcvault.com/ [Accessed: 19 April 2017]

Gouraud, H. (1971). *Computer display of curved surfaces*. University of Utah. [Online]. Available from: https://collections.lib.utah.edu/details?oldid=uspace+4477 [Accessed: 1 January 2017]

Han, S., and Sander, P.V. (2016). Triangle reordering for reduced overdraw in animated scenes. *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* [Online], pp.23-27. Available from: http://dl.acm.org/citation.cfm?id=2856408&CFID=747123809&CFTOKEN=14107905 [Accessed: 20 April 2017]

Hargreaves, S. and Harris, M. (2004). *Deferred Shading*. Nvidia Corporation. [Online]. Available from: http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf [Accessed: 3 January 2017]

Harris, M. (2008). *The challenge of innovation driven curricula change. Insight into some major trends in IT and its impact on Informatics curricula*. Informatics Education Europe III. [Online]. Available from: http://www.dsi.unive.it/IEEIII/Harris.pdf [Accessed 2 January 2017]

HSA Foundation (2016). *HSA Foundation*. HSA Foundation. [Online]. Available from: http://www.hsafoundation.com/ [Accessed: 19 April 2017]

I3D (2017). *Symposium on Interactive 3D Graphics and Games*. [Online]. Available from: i3dsymposium.org [Accessed: 28 March 2017]

IEEE (2017). *IEEE Transactions on Visualization and Computer Graphics*. [Online]. Available from: http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=2945 [Accessed: 28 March 2017]

Incentive Software (1987). *Driller*. [Software]. Amiga version. Incentive Software.

Incentive Software (1991). *3D Construction Kit*. [Software]. Amiga version. Domark.

Infinity Ward and Sledgehammer Games (2011). *Call of Duty: Modern Warfare 3*. [Software]. Windows version. Activision.

Khronos Group (2017a). *Vulkan: Industry Forged*. Khronos Group. [Online]. Available from: https://www.khronos.org/vulkan/ [Accessed: 19 April 2017]

Khronos Group (2017b). *Khronos Videos and Presentations*. Khronos Group. [Online]. Available from: https://www.khronos.org/developers/library [Accessed: 19 April 2017]

Kojima Productions (2014). *Metal Gear Solid V: Ground Zeroes*. [Software]. Windows version. Konami Digital Entertainment.

Konami Computer Entertainment Japan (1998). *Metal Gear Solid*. [Software]. Playstation version. Konami.

Lengyel, E. (2004). *Mathematics for 3D Game Programming and Computer Graphics*. 2nd edn. Hingham: Charles River Media, Inc.

Liu, H.H. (2009). *Software Performance and Scalability: A Quantitative Approach*. Hoboken, New Jersey: John Wiley & Sons, Inc.

Luebke, D., Reddy, M., Cohen, J.D., Varshney, A., Watson, B. and Huebner, R. (2003). *Level of Detail for 3D Graphics*. San Francisco: Morgan Kaufmann Publishers.

Luna, F.D. (2012). *Introduction to 3D Game Programming with DirectX 11*. Dulles: Mercury Learning And Information, LLC.

Kinney, J. (2014). *Game Asset Production Pipeline in Unreal Engine*. [DVD]. Digital Tutors.

McDonald, J. (2012). Don't Throw It All Away: Efficient Buffer Management. [On-

line]. In: *Game Developers Conference (GDC) 2012*: March 5-9. San Francisco, CA. Available from: https://developer.nvidia.com/gdc-2012 [Accessed: 19 April 2017]

Microsoft Corporation (2015). *Visual Studio Graphics Analyzer*. [Software]. Windows version. Microsoft Corporation.

Microsoft Developer Network (MSDN) (2016). *Direct3D Documentation*. [Online]. Available from:
https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466%28v=vs.85%29.aspx [Accessed: 20 April 2017]

Microsoft Developer Network (MSDN) (n.d. a). *Resource Limits* (Direct3D 11). [Online]. Available from:
https://msdn.microsoft.com/en-us/library/windows/desktop/ff819065(v=vs.85).aspx#resource_limits_for_feature_level_11_hardware [Accessed: 19 April 2017]

Microsoft Developer Network (MSDN) (n.d. b). *Performance Optimizations (Direct3D 9)*. [Online]. Available from:
https://msdn.microsoft.com/en-us/library/windows/desktop/bb147263%28v=vs.85%29.aspx [Accessed: 19 April 2017]

Möller, T. A. , Haines, E. and Hoffman, N. (2008). *Real-Time Rendering*. 3rd edn. Wellesley : A K Peters, Ltd.

Naughty Dog (2016). *Uncharted 4*. [Software]. PlayStation 4 version. Sony Computer Entertainment.

Newell, M. E., Newell, R. G. and Sancha, T. L. (1972). A new approach to the shaded picture problem. *Proc. ACM National Conference*, pp. 443-450.

Nguyen, H., ed. (2008). *GPU Gems 3*. Boston: Addison-Wesley.

Nvidia Corporation (2016a). *GeForce 256*. Nvidia Corporation. [Online]. Available from: http://www.nvidia.com/page/geforce256.html [Accessed: 3 January 2017]

Nvidia Corporation (2016b). *Threaded rendering Vulkan sample*. Nvidia Corporation. [Software]. [Online]. Available from:
http://nvidiagameworks.github.io/GraphicsSamples/ThreadedRenderingVulkanSample.htm [Accessed: 20 April 2017]

Nvidia Corporation (2017a). *Nvidia GeForce 8 Series*. Nvidia Corporation. [Online]. Available from: http://www.nvidia.com/page/geforce8.html [Accessed: 3 January 2017]

Nvidia Corporation (2017b). *Nvidia Tegra X1*. Nvidia Corporation. [Online]. Available from: http://www.nvidia.com/object/tegra-x1-processor.html [Accessed: 3 January 2017]

Nvidia Corporation (2017c). *Nvidia GDC 2017*. Nvidia Corporation. [Online]. Available from: http://www.nvidia.com/object/gdc-2017.html [Accessed: 19 April 2017]

Nvidia Corporation (n.d.). *Modern GPU Architecture*. Nvidia Corporation. [Online]. Available from: ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf [Accessed: 19 April 2017]

Perry, L. (2002). Modular Level and Component Design. *Game Developer* [Online], November issue, pp.30-35. Available from: https://docs.unrealengine.com/udk/Three/rsrc/Three/ModularLevelDesign/ModularLevelDesign.pdf [Accessed: 19 April 2017]

Pharr, M. and Humphreys, G. (2010). *Physically Based Rendering: From Theory To Implementation*. 2nd edn. San Francisco: Morgan Kauffman Publishers.

Phong, B.T. (1975). Illumination for computer generated pictures. *Communications of the ACM* [Online], 18, no. 6, pp. 311-317 Available from: http://www.cs.northwestern.edu/ ago820/cs395/Papers/Phong_1975.pdf [Accessed: 1 January 2017]

Pluralsight LLC (2012). *Asset Workflows for Modular Level Design*. Pluralsight LLC. [Online]. Available from: http://www.digitaltutors.com/tutorial/857-Asset-Workflows-for-Modular-Level-Design [Accessed: 19 April 2017]

Poynton, C. (1996). *A Technical Introduction to Digital Video*. New York: John Wiley & Sons.

Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-D shapes. *ACM SIGGRAPH Computer Graphics* [Online], vol.24(4), pp. 197-206. Available from: http://dl.acm.org/citation.cfm?doid=97880.97901 [Accessed: 3 January 2017]

Sander, P.V., Nehab, D. and Barczak, J. (2007). Fast Triangle Reordering for Ver-

tex Locality and Reduced Overdraw. *ACM SIGGRAPH Papers* [Online], article no. 89. Available from: http://gfx.cs.princeton.edu/pubs/Sander_2007_%3ETR/tipsy.pdf [Accessed: 20 April 2017]

Schauerte, B. (2014). *Conference ranks*. [Online]. Available from: http://www.conferenceranks.com/visualization/msar2014.html?field=Graphics&visualization=Bars [Accessed: 19 April 2017]

Schott M. and Bishop L.M. (2016). Low-Overhead Rendering with OpenGL and Vulkan. [Online]. In: *Game Developers Conference (GDC) 2016*: March 16-18. San Francisco, CA. Available from: https://developer.nvidia.com/gdc-2016 [Accessed: 2 January 2017]

Sellers, G. and Kessenich, J. (2016). *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Boston: Addison-Wesley.

Shimpi, A.L. (2013). *AMD's Jaguar Architecture: The CPU Powering Xbox One, PlayStation 4, Kabini & Temash* [Online]. AnandTech. Archived from the original on December 11, 2013, available from: https://web.archive.org/web/20131211030440/http://www.anandtech.com/show/6976/amds-jaguar-architecture-the-cpu-powering-xbox-one-playstation-4-kabini-temash/4 [Accessed: 1 January 2017]

Shreiner, D., Sellers, G., Kessenich, J. and Kane, B.L. (2013). *The OpenGL Programming Guide*. 8th edn. Boston: Addison-Wesley.

Smith, R. (2013). *AMD Kaveri APU Launch Details: Desktop, January 14th*. AnandTech. [Online]. Available from: http://www.anandtech.com/show/7507/amd-kaveri-apu-launch-details-desktop-january-14th [Accessed: 3 January 2017]

Stanford Computer Graphics Laboratory (2017). *The Stanford 3D Scanning Repository*. [Online]. Available from: http://graphics.stanford.edu/data/3Dscanrep/ [Accessed: 25 March 2017]

Stengel, S. (2016). *NEC APC Advanced Personal Computer*. [Online]. Obsolete Technology Website. Available from: http://oldcomputers.net/nec-apc.html [Accessed: 3 January 2017]

Stephens, N. (2011). *Environment Modeling for Games*. [DVD]. The Gnomon Workshop.

Stephens, N. (2014). *Environment Modeling and Sculpting for Game Production*. [DVD]. The Gnomon Workshop.

Storsjö, M. (2008). *Efficient Triangle Reordering for Improved Vertex Cache Utilisation in Realtime Rendering*. Åbo Akademi University. MSc Thesis. [Online]. Available from: http://www.martin.st/thesis/efficient_triangle_reordering.pdf [Accessed: 20 April 2017]

Taito (1975). *Gun Fight*. [Software]. Arcade version. Taito.

Taito (1978). *Space Invaders*. [Software]. Arcade version. Taito.

Taylor, J. (2013). *AMD and The Sony PS4. Allow Me To Elaborate*. [Online]. AMD. Archived from the original on December 11, 2013, available from: https://web.archive.org/web/20130526191443/http://community.amd.com/community/amd-blogs/amd-unprocessed/blog/2013/02/21/amd-and-the-sony-ps4-allow-me-to-elaborate [Accessed: 1 January 2017]

Teque Software Development (1990). *Castle Master*. [Software]. Amiga version. Incentive software.

Van Oosten, J. (2014). *Texturing and Lighting in DirectX 11*. 3D Game Engine Programming. [Online]. Available from: http://www.3dgep.com/texturing-lighting-directx-11/ [Accessed: 1 January 2017]

Walker J. (2014). *Physically Based Shading In UE4*. Epic Games Inc. [Online]. Available from: https://www.unrealengine.com/blog/physically-based-shading-in-ue4 [Accessed: 19 April 2017]

Watkins, G.S. (1970). *A real time visible surface algorithm*. University of Utah. [Online]. Available from: http://dl.acm.org/citation.cfm?id=905548 [Accessed: 1 January 2017]

Watson, B., Walker, N. and Hodges L.F. (2004). Supra-Threshold Control of Peripheral LOD. *ACM Transactions on Graphics (TOG)* [Online], vol. 23(3), pp.750-759. Available from: http://dl.acm.org/citation.cfm?doid=1186562.1015796 [Accessed: 20 April 2017]

Whitted T. (1980) An improved illumination model for shaded display. *Communications of the ACM* [Online], vol.23(6), pp. 343-349. Available from: http://dl.acm.org/citation.cfm?id=358882 [Accessed: 2 January 2017]

Williams, L. (1978). Casting Curved Shadows on Curved Surfaces. *ACM SIGGRAPH Computer Graphics* [Online], vol. 12(3), p. 270-274. Available from: http://dl.acm.org/citation.cfm?id=807402 [Accessed: 19 April 2017]

Wilson, J. (2015). *Physically-Based Rendering, And You Can Too!*. Marmoset LLC. [Online]. Available from: https://www.marmoset.co/posts/physically-based-rendering-and-you-can-too/ [Accessed: 3 January 2017]

Xatrix Entertainment Inc . (1998). *Quake II: The Reckoning*. [Software]. Windows version. Activision.

Yoon, S.E., Salomon, B. and Manocha, D. (2003). Interactive View-Dependent Rendering with Conservative Occlusion Culling in Complex Environments. *Proceedings of the 14th IEEE Visualization* [Online], p.22. Available from: http://gamma.cs.unc.edu/VDR/final.pdf [Accessed: 20 April 2017]

Young, V. (2005). Book Excerpt: Programming a Multiplayer FPS in DirectX: Culling. *Gamasutra*. [Online]. Available from: http://www.gamasutra.com/view/feature/130689/book_excerpt_programming_a_.php [Accessed: 20 April 2017]

Suggested:

Bakhoda, A., Yuan G.L., Fung, W.W.L., Wong, H. and Aamodt, T.M. (2009). *Analyzing CUDA workloads using a detailed GPU simulator*. IEEE International Symposium on Performance Analysis of Systems and Software. [Online]. Available from: http://ieeexplore.ieee.org/document/4919648/ [Accessed: 13 May 2017]

Eberly, D.H. (2000). *3D Game Engine Design. A Practical Approach to Real-Time Computer Graphics*. San Francisco: Morgan Kauffman.

Granberg, C. (2009). *Character Animation With Direct3D*. Hingham: Charles River Media, Inc.

Gruen, H. (2015). *Constant Buffers without Constant Pain*. Nvidia GameWorks. [Online]. Available from: https://developer.nvidia.com/content/constant-buffers-without-constant-pain-0 [Accessed: 26 March 2017]

Gutierrez, D., Seron, G.J., Munoz, A., Anson, O. (2006). Simulation of atmospheric phenomena. *Elsevier Computers and Graphics*. [Online]. Available from: http://web.mit.edu/ytc/www/HLMA/Ref/1-s2.0-S0097849306001026-main.pdf [Ac-

cessed: 7 January 2017]

Hardy, A.C. (1920). A Study of the Persistence of Vision. *Proc. Natl. Acad. Sci. USA*, 6(4), pp. 221-224. [Online]. Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1084467/pdf/pnas01913-0065.pdf [Accessed: 2 January 2017]

Hollasch, S.R. (1991). *Four-Space Visualization of 4D Objects*. [Online]. Tempe: Arizona State University. MSc Thesis. Available from: http://steve.hollasch.net/thesis/ [Accessed: 7 January 2017]

Kessenich, J., Baldwin, D. and Rost, R. (2014). *The OpenGL Shading Language*. [Online]. Available from: https://www.opengl.org/documentation/glsl/ [Accessed: 7 January 2017]

Sommerville, I. (2011). *Software Engineering*. 9th edn. Boston: Addison-Wesley.

Tang, T., Yang, X., and Lin, Y. (2011). *Cache Miss Analysis for GPU Programs Based on Stack Distance Profile*. 31st IEEE International Conference on Distributed Computing Systems. [Online]. Available from: http://ieeexplore.ieee.org/document/5961739/ [Accessed: 13 May 2017]

Zhang, Y. and Owens, J.D. (2011). *A quantitative performance analysis model for GPU architectures*. IEEE 17th International Symposium on High Performance Computer Architecture. [Online]. Available from: http://ieeexplore.ieee.org/document/5749745/ [Accessed: 13 May 2017]

A. Table of measurements

Key Notes / Abbreviations:
"CB" – Collective vertex/index buffers
"FL" – Frame limiter
"FOU" – Frequency-of-update constant buffers
"FTB" – Front-to-back rendering order
"BTF" – Back-to-front rendering order
"PRE-WS" – Pre-transformation of immobile objects to World Space
"INST" – Instancing

Test case	Description	Draw calls	FPS	Avg. frame time (ms)	Iter./sec
TC-01	–	97	3441	0.290503	3441
TC-02	CB	97	3676	0.271727	3676
TC-03	CB, FL	97	60	0.303235	32792276
TC-04	CB, FOU	97	3797	0.262851	3797
TC-05	CB, FTB	97	3772	0.26461	3772
TC-06	CB, BTF	97	3671	0.271991	3671
TC-07	CB, PRE-WS	97	3775	0.264411	3775
TC-08	CB, INST	9	4193	0.238032	4193
TC-09	CB, INST, FL	9	60	0.159883	32633789
TC-10	CB, INST, FOU	9	4228	0.235997	4228
TC-11	CB, INST, FTB	9	4163	0.239366	4163
TC-12	CB, INST, BTF	9	3975	0.251118	3975
TC-13	CB, INST, PRE-WS	9	4221	0.23638	4221
TC-14	CB, INST, FL, FOU, FTB, PRE-WS	9	60	0.130589	33162978

Table A.1: Collective table of measurements for all examined test cases.

B. Real-time rendering: Fundamental technical terms and concepts

B.1 Renderer

A renderer is a computer program that generates images based on the input data it receives (Möller, Haines and Hoffman, 2008). The process it performs is called "*rendering*" in computer graphics (ibid.). Based on the inherent dimensions of the render data involved, the process of rendering can be distinguished primarily into 2D (2-dimensional) and 3D (3-dimensional). In the case of 2D rendering, the data manipulated often directly corresponds to on-screen objects or pixels. In 3D rendering, the data is first manipulated in 3-dimensional space, and is eventually projected on 2-dimensional space, in a region ultimately corresponding to a portion of the computer screen. In every case, prior to seeing the result on-screen, *rasterization* must occur, which is the process during which the computation of the individual colours of the pixels involved for displaying the image is performed (ibid.; Luna, 2012).

B.2 GPU and real-time rendering

GPU is an acronym for "Graphics Processing Unit" (Bauer, 2015). Due to the high demand for performance in graphics-related computations, contemporary computer systems usually utilize a specialized hardware unit (either discrete, or integrated to the CPU or motherboard) specifically for performing graphics-related computations, rather than using the computer's more generically designed CPU (Central Processing Unit). This is especially true when *real-time rendering* techniques are involved, which expect real-time response times and have particularly high performance requirements (ibid.; Möller, Haines and Hoffman, 2008; Luna, 2012).

Note that there is an emerging tendency towards the use of *integrated* GPUs (as opposed to dedicated ones, being the norm for several decades), to alleviate some of the synchronization difficulties and added complexity that exclusive GPU memory models cause (see section 1.3.1). Even so and despite the differences noted here, the following are invariant for real-time rendering:

1. Specialized hardware (GPUs) is always used for real-time graphics computation, and

2. The interfacing with such hardware increases overall system complexity, due to its heterogeneous processing components

GPU-rendered graphics are also often referred to as "hardware accelerated" graphics, and the techniques that produce them as employing "hardware rendering". In contrast, rendering techniques that primarily utilize the CPU are often referred to as "software rendering", or CPU-based rendering. Software rendering, although considerably slower compared to hardware rendering (and in general non real-time), is a technique allowing considerably more flexibility in the development phase of a rendering system, because it does not need to directly interface with the GPU, effectively reducing the amount of system complexity that requires to be addressed by the implementation.

Software rendering is also capable of producing much more accurate results, at the cost of longer rendering time. Due to this reason, it is often used in conjunction with higher-accuracy algorithms (e.g. for physically accurate simulation of lighting and shadows, kinematics, etc.), and is still widely used for various types of scientific visual computation applications, as well as in the film and visual effects industry.

GPU-rendered graphics are used where real-time visual performance is critical – often because of the requirement for the rendering system to be *interactive*. It is a somewhat less flexible system imposing additional requirements and inhibitions during the development phase (such as passing data to the GPU using specific structures and formats, having a limit of maximum instructions per compiled GPU shader program, etc.) but makes it up in terms of highly increased rendering speed. As GPUs become more powerful and sophisticated, the quality and complexity of real-time graphics that they are capable of increases exponentially, coming nearer to that of CPU-rendered graphics.

B.3 Polygons and polygon count

A 3D renderer typically takes its main input in the form of 3-dimensional coordinates, representing positions in 3D space. These positions (or 3D points, technically called *vertices*) are then combined by the renderer to produce renderable *primitives*.

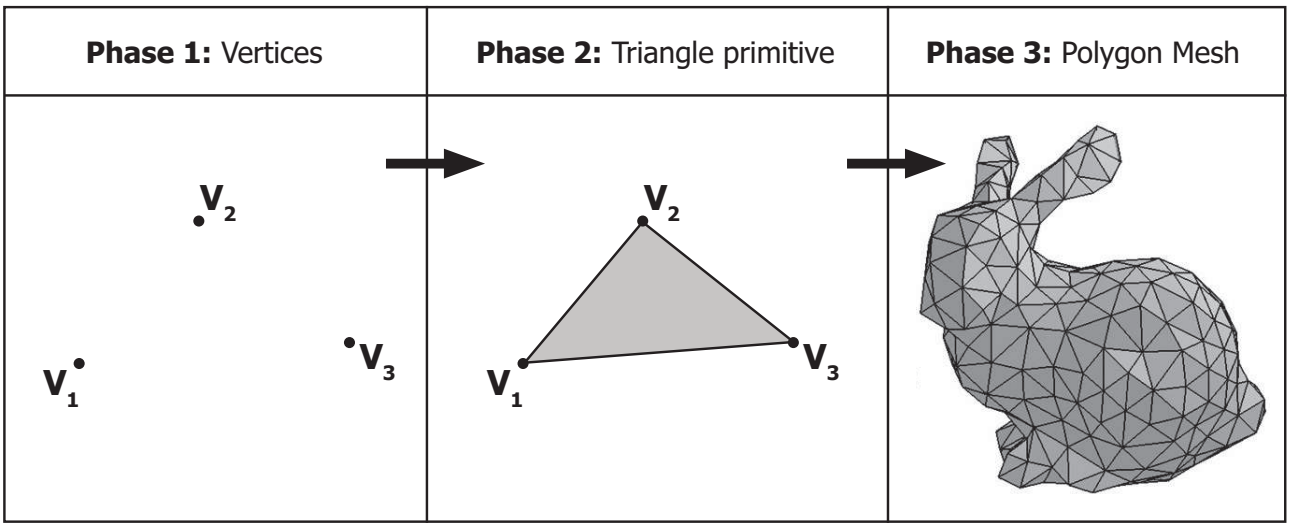

Figure B.1: The constituent phases of assembling a polygon mesh to be displayed by the GPU. Vertices are used to assemble triangle primitives, which in turn collectively formulate a polygon mesh.

Such primitives are later combined themselves in order to create more complex 3-dimensional shapes (Fig. B.1) (Möller, Haines and Hoffman 2008; MSDN, 2016; Luna, 2012).

Typical primitives supported by contemporary graphics APIs like DirectX, OpenGL and Vulkan are *points*, *lines* (which technically are line segments connecting two points), and *polygons*. These polygons, when actual rendering is concerned, are expected by the GPU as a set of *triangles*, each of which is defined as a set of 3 points representing its vertices (ibid.).

The term "*polygon count*" refers to the number of polygons that collectively compose a 3D object (i.e. a polygon mesh). It can also be used to refer to the total number of polygons contained by the entire 3D scene that is being rendered. An object's polygon count is one of the measures commonly used for determining the average impact the object being rendered will have on a rendering system's resources and overall performance (ibid.).

Also, note that in this document there is frequent reference to the *triangle* count of an object. In the context of this study this term is intended as interchangeable with "polygon count", since this study is concerned with GPU rendering and the GPU expects polygonal meshes to be strictly composed of triangles. However, in contexts that allow the use of other polygon types, a given polygon count may be a distinct value from the corresponding triangle count, because the polygons referred-to are

not necessarily triangles (but perhaps are quadrilaterals, etc.).

B.4 Frame rate and Frames per second (FPS)

The amount of rendered frames output by a rendering system in a given period of time represent the *frame rate* of that system, often measured in the number of output *frames per second*, abbreviated as "FPS" (Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005). Depending on the collective complexity of the calculations performed by the system per loop, the maximum frame rate that it can output at any given time can vary greatly, and depends on a number of factors. These can include:

- The size of the data composing the 3D scene: larger scenes that are composed of more polygons and higher resolution textures will require more time to be processed by the GPU, resulting in a comparatively lower frame rate.

- The complexity of computations performed during the "Update" phase: The more complex the calculations performed during the "Update" phase of the loop (treated in detail in section B.5), the lower the frame rate will be. Indicative computations performed here can be: Computing the current positions of the scene's objects, performing collision detection, physics simulation and animation computations, AI behaviour and/or path planning computations etc. . Applications whose main loop is mostly spent in performing this phase are often called "CPU-bound", because the majority of the computational weight falls on the CPU.

- The complexity of computations performed during the "Render" phase (see B.5): Similarly, more complex rendering computations will also lower the frame rate. Typical rendering computations can include computing lighting and shadows and post-processing visual effects applied to the frame. In contrast to the "CPU-bound" applications mentioned above, when an application is spending the majority of time comprising the main loop in rendering calculations (typically performed on the GPU), it is often referred to as a "GPU-bound" application.

B.5 Main loop structure of an interactive real-time rendering application

An interactive real-time rendering application is expected to render frames continuously and output them on screen. In consequence of this, it is required to maintain a minimum frame rate of 24 frames per second, since a frame rate lower than that would no longer allow real-time responsiveness or satisfy the persistence of vision limit (Möller, Haines and Hoffman, 2008; Poynton, 1996). In fact, for applications requiring smoother interaction with the user (such as video games, where oftentimes fast user reflexes are expected), the suggested minimum frame rate to avoid user exertion is 60 frames per second (ibid.).

Fundamentally, the "main loop" of an interactive real-time rendering application consists of a looping control structure which performs three main steps: process user input, update data based on input, and render a frame using the updated data (Fig. B.2). This process repeats for the duration of the application session, and allows the user input to directly control the rendered output displayed on screen (Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005). As mentioned, "CPU-bound" real-time applications spend more time in the "Update" phase, while "GPU-bound" ones spend more time in the "Render" phase.

B.6 Textures

Textures are data structures that contain an array of raster data (i.e., *pixels*). Usually, textures are used to store image data that constitutes an input and/or output of the rendering process. The dimensions of the texture's array can vary – textures of 1, 2 and 3 dimensions are commonly used (Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005).

The *resolution* of a texture corresponds to the size of its contained array of raster data. The resolution has an impact on the amount of GPU memory a texture occupies when loaded, and also on the amount of computations required to process or update that data when rendering is performed. Consequently, the efficient management of textures is a matter that can affect the overall performance of a rendering system (ibid.).

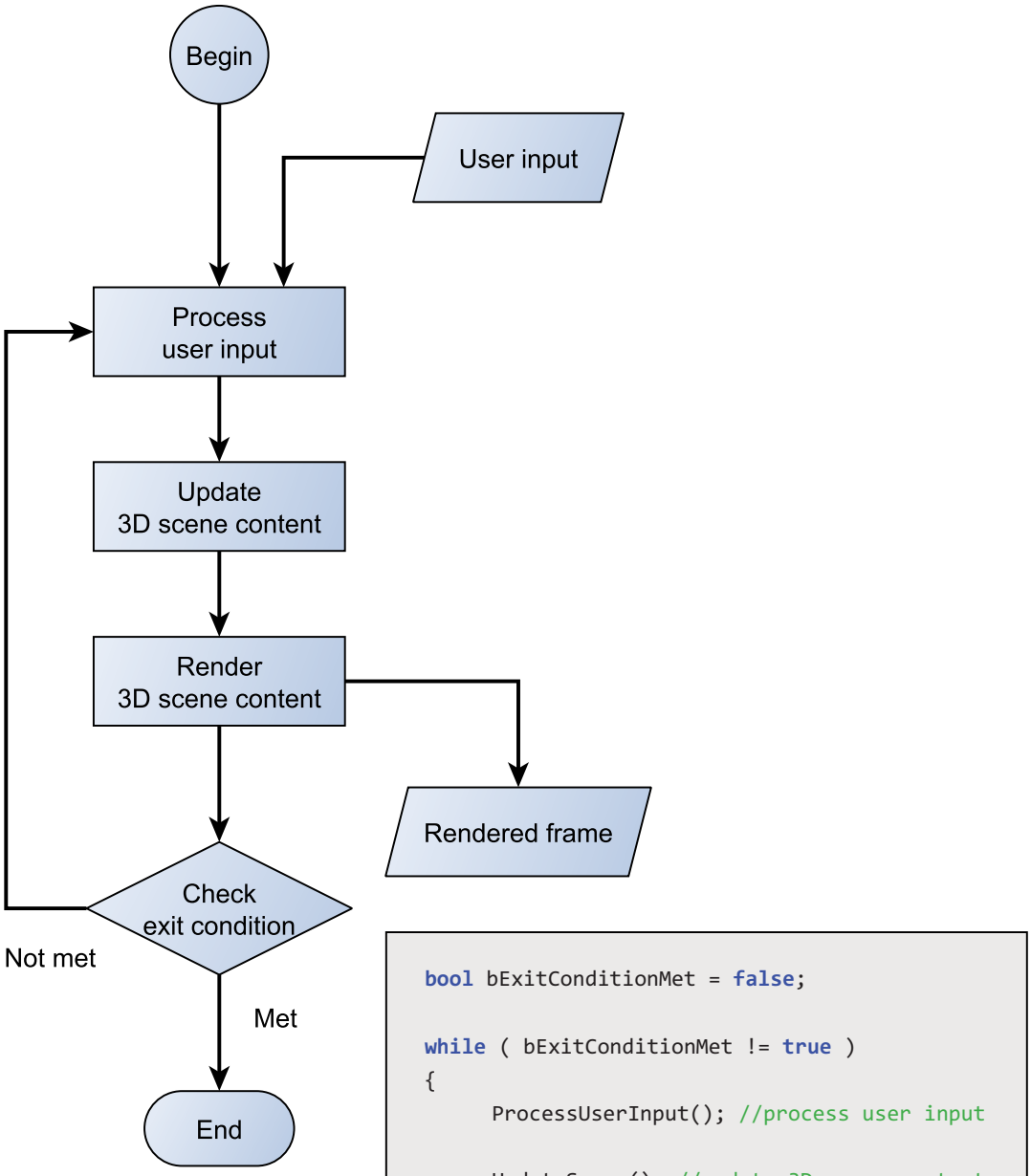

Figure B.2: The structure of the "main loop" of an interactive real-time rendering application consists of 3 main phases: Process user input, update the scene, and render a frame of the scene. This sequence is repeated continuously throughout the application session.

B.7 GPU Buffers and draw calls

In GPU-based rendering, data is required to be made accessible to the GPU prior to its computations being performed – in turn requiring CPU-GPU *synchronization* techniques for managing these system resources. As discussed in section 1.3.1, this increased complexity and heterogeneity of the system requires to be adequately addressed in order to have satisfactory performance results. Until now, *exclusive GPU memory* models are the ones more often in use, and are also the focus for this study. It is noted however that recently, there is an emerging tendency for shifting to a *unified* CPU-GPU memory model (as discussed in section 1.3.1).

Assuming the much more common exclusive GPU memory model is employed, the CPU and GPU utilize mutually exclusive memory address spaces. For GPU-based rendering to occur, data is sent from system memory (corresponding to RAM) to GPU memory (sometimes referred to as "VRAM" or "video RAM"). Depending on the GPU, this can be either a portion of the system memory allocated specifically for GPU use (i.e. partitioned), or can be physically located on the GPU itself (i.e. physically discrete). In some cases, both techniques are simultaneously employed.

In any case, in order for the data to become accessible by the GPU, they must be loaded through the graphics API used into *GPU buffers*. These are allocated portions of GPU memory that contain render data, which is typically copied there from system memory (Fig. B.3).

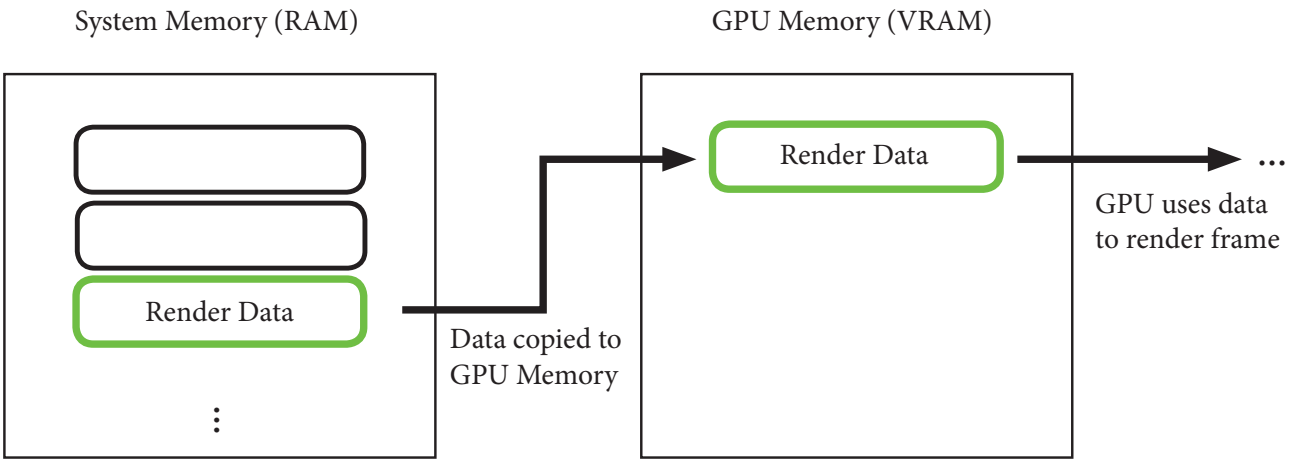

Figure B.3: The sequence of steps for render data to reach the GPU: In the exclusive GPU memory model, data needs to be copied from system memory (RAM) to GPU memory (VRAM) before the GPU can utilize the data for rendering.

Once that is done, the GPU can be directed to render the data corresponding

to one or more objects, by performing a "*draw call*". A draw call is a graphics API function call performed by the application, that directs the GPU to "draw" (that is, to render) the specified data (Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005).

As we will see, interaction between the application and the GPU is expensive, and comes with a computational overhead that reduces system responsiveness. Due to this, GPU buffer data updates, draw calls and similar operations should be kept to a minimum so that the related overhead is correspondingly minimized (ibid.).

B.8 Shader programs

Contemporary GPUs allow the composition of relatively short programs that are subsequently loaded on, and executed by, the GPU. These programs, called *shader programs*, allow programmable control over parts of the rendering pipeline adhered to by the GPU. There are several rendering stages that shader programs can be composed for, although the most fundamental operations that could be briefly mentioned here are those of the *vertex shader* and the *fragment shader*. These are of main concern to us, as they are directly connected to optimizations proposed in this study.

- A vertex shader is a program that is executed for every vertex of the input data to be rendered.
- A fragment shader is a program that is executed for every pixel fragment generated by processing the input data to be rendered.

B.9 Pixel fragments

A pixel fragment is a set of data generated during the rasterization process (see section B.1). It is capable of representing the potential final colour of a pixel, and also stores a *depth* value corresponding to that fragment. After being generated, pixel fragments are evaluated. During their individual evaluation, a generated fragment may be discarded, combined with, or may overwrite another fragment (Möller,

Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005; MSDN, 2016). This process continues until all the fragments corresponding (and potentially contributing) to a pixel's value are considered, and the final colour value of that pixel becomes computed (ibid.).

B.10 Depth culling

One of the ways the evaluation of pixel fragments is performed is by the process of *depth culling*. During depth culling, the distance of the pixel fragment from the renderer's "virtual camera" position is considered. This is done by evaluating the fragment's *depth* value. If subsequent fragments are found whose distance from the camera position is greater than that of the last fragment, they are discarded. On the other hand, if subsequent fragments have shorter distances from the camera position, they overwrite the previous fragment's value (Fig. B.4). This behaviour occurs to mimic the effect that opaque objects that are closer to the viewer occlude objects that are farther away. Hence, through depth culling, the computation of the colour value of pixel fragments that are not going to be visible is skipped, expediting computations (Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005; MSDN, 2016).

B.11 Overdraw

When subsequent pixel fragments do not get discarded, but either replace or combine with previous ones, *overdraw* occurs. Essentially, overdraw represents the overwriting of a pixel's prospective value by a new one (Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005; MSDN, 2016). Computationally speaking, the following brief but important observations can be made regarding overdraw's influence:

1) The greater the degree of overdrawing, the higher the overhead in computing the resultant frame. That is because the presence of overdraw increases the total number of pixel fragments that are fully computed (and not skipped). So it is desirable to have minimal overdraw when rendering.

2) The order of rendering 3D objects matters. Rendering the same data in a

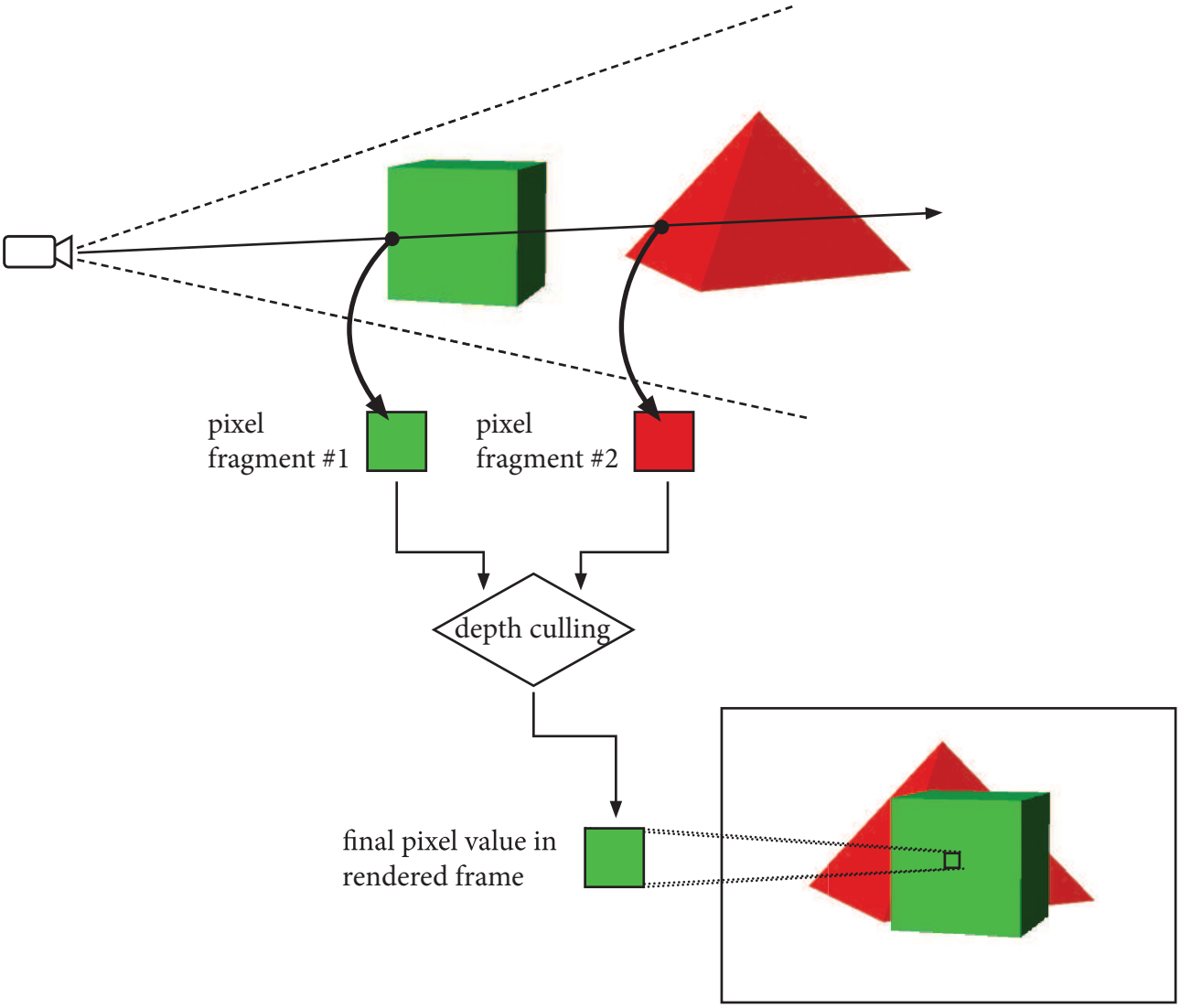

Figure B.4: Overview of the depth culling process. Since the green pixel fragment (#1) has a lower depth value, it is retained, and the red fragment (#2) is discarded.

different order can greatly influence the extensiveness of the required computations (see Fig. B.5 for an example). As will be seen, render data can be effectively sorted prior to beginning rendering.

B.12 Shader complexity

Since shader programs are required to execute per frame with very high frequencies – such as for every scene vertex or even for every pixel fragment within the frame – the inherent computational *complexity* of these shader programs is an important factor for system performance. This term corresponds to the amount and complex-

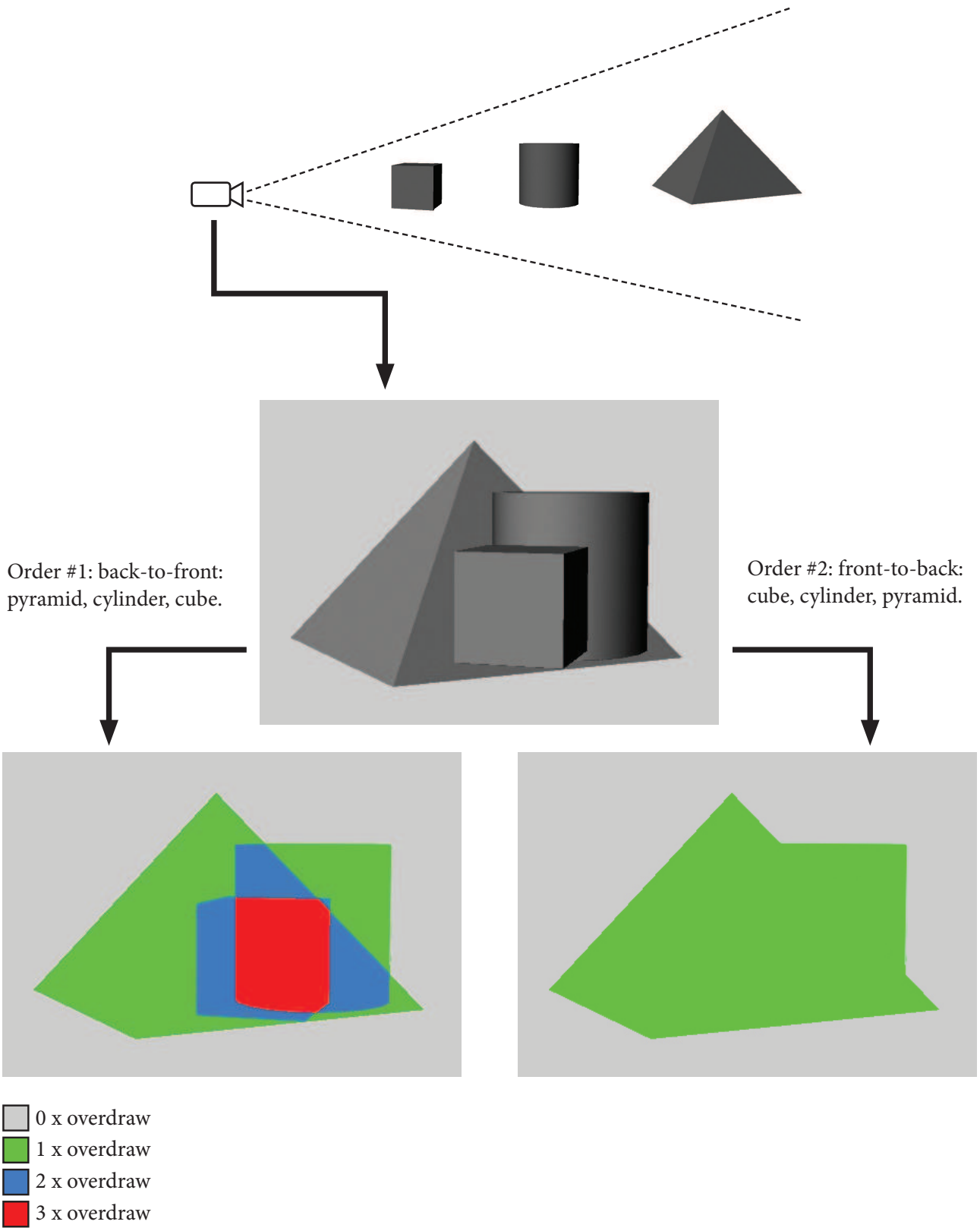

Figure B.5: An example illustrating the overdraw process, and how the rendering order can affect the amount of overdraw for a given frame.

ity of compiled GPU instructions that the shader program in question amounts to (MSDN, 2016; Luna, 2012; Eberly, 2005). Shader complexity can vary a lot across different portions of a 3D scene, and even different portions of a single frame (Fig. B.6). This is particularly true for fragment shaders, which may involve very elaborate computations to accurately determine a pixel fragment's colour value. Minimizing such complexity wherever applicable is always desired for all types of shader programs (ibid.).

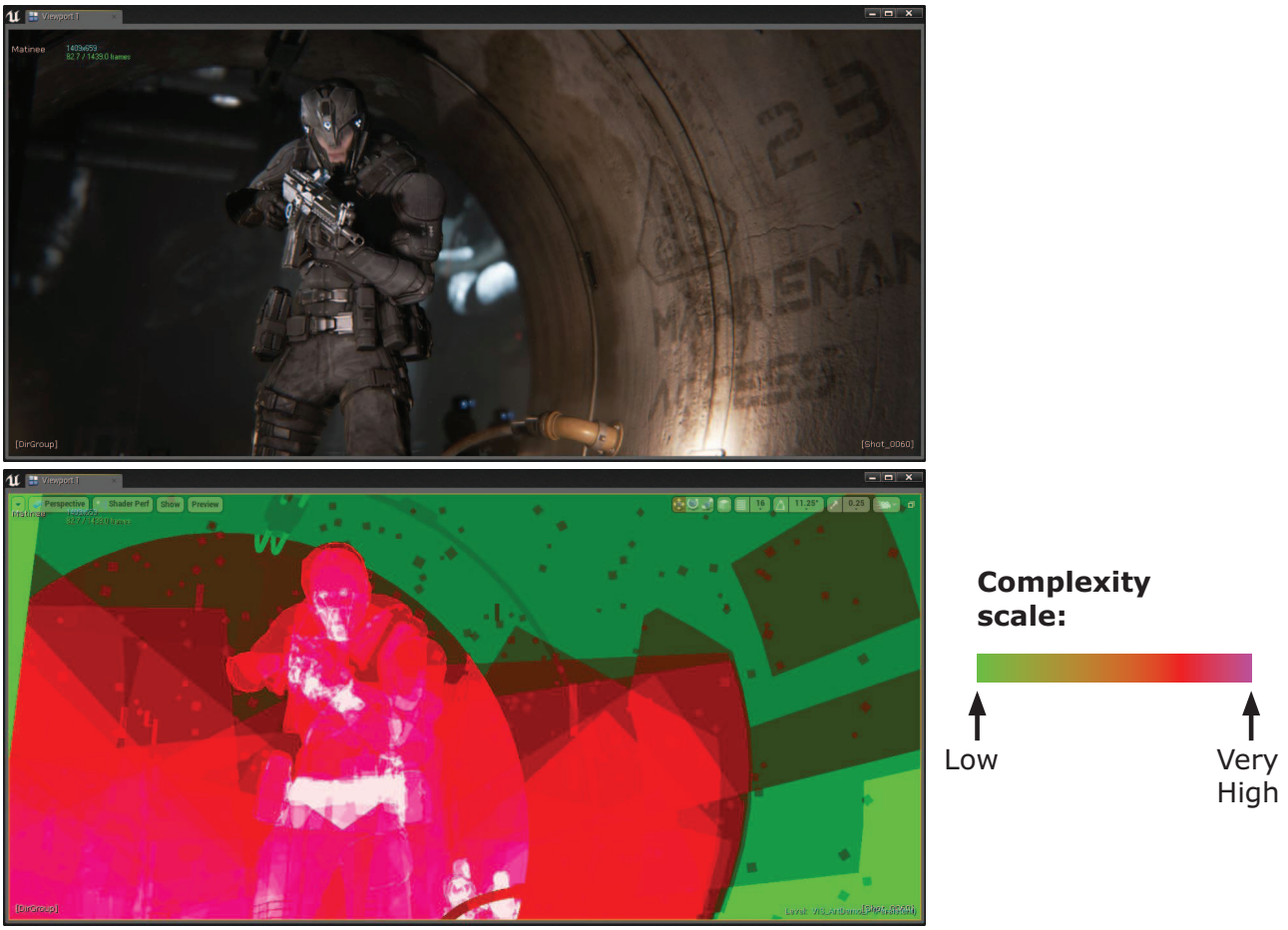

Figure B.6: A visualization of per-pixel shader complexity for a given frame (Epic Games Inc., 2016).

B.13 Coordinate systems and spaces

For more effective and flexible results, 3D data is manipulated across several coordinate spaces in the process of rendering a frame (Lengyel, 2004; MSDN, 2016; Möller, Haines and Hoffman, 2008; Luna, 2012; Eberly, 2005; Shreiner et al., 2013). The data is manipulated, then transformed to another space, where manipulation

continues until the process results in rasterization and pixel fragment computation. Since this study involves optimizations pertaining to such coordinate spaces, a short overview for those that are discussed is given here (also see Fig. B.7):

- Local (or Object) Space: This is a 3-dimensional space where coordinates are specified with the assumption that the origin (0,0,0) of the space is local – i.e. represents the immediate 3D object being specified. This is particularly helpful during the creation of 3D assets, since the origin (0,0,0) is made to lie close to or exactly at the centre of the 3D object being designed.
- World Space: This is the 3D space that all Local-Space objects must be eventually converted into, each assuming its respective position, orientation and scale in the 3-dimensional "world" (or "scene") that contains them.
- View Space: A 3D space where the origin is made to coincide with the position of the renderer's current viewpoint, or "virtual camera". All other objects are specified relative to that position. The direct conversion of an object's coordinates from Local Space to View Space is also often referred to as the "Model-View" transformation.

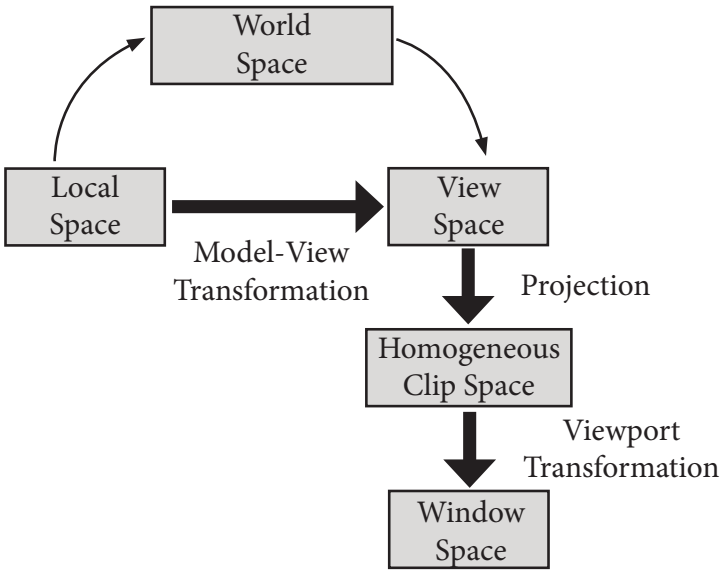

Figure B.7: Common coordinate systems and spaces for 3D transformations (Lengyel, 2004).

List of Figures

1.1 The Phong shading model. 8

1.2 Environment mapping. 9

1.3 Bump mapping and a screenshot from a 3D animation by Blinn and Kohlhase. 9

1.4 First usage of ray tracing. 10

1.5 "Spasim", an example of early "nearly-real-time" 3D rendering. 11

1.6 Solid 3D surfaces in the Freescape 3D engine. 11

1.7 The 3D CAD software and game engine 3D Construction Kit. 12

1.8 "Quake II: The Reckoning" (Xatrix Entertainment Inc., 1998) and "Unreal" (Epic MegaGames, Digital Extremes and Legend Entertainment, 1998), competitors at the time. 12

1.9 Deferred shading example. 13

1.10 Wireframe and solid views of a character from two installments of the video game series "Tomb Raider" (Core Design Ltd., 1996; Crystal Dynamics, 2007). 14

1.11 Comparison of a character from two installments of the video game series "Metal Gear" (Konami Computer Entertainment Japan, 1998; Kojima Productions, 2014). 14

1.12 Ray tracing benchmarks between 2004 and 2007. 15

1.13 Crysis (Crytek, 2007) and Call of Duty: Modern Warfare 3 (Infinity Ward and Sledgehammer Games, 2011). 15

1.14 A CPU-rendered frame by the open-source PBR renderer "pbrt" (Pharr and Humphreys, 2010). 16

1.15 A real-time rendered PBR frame from the video game "Uncharted 4" (Naughty Dog, 2016). 17

1.16 A screenshot from a 3D scene in Unreal Engine 4's real-time PBR-shaded environment. 18

1.17 Real-time PBR material examples. 19

1.18 A timeline of 3D rendering milestones. 20

1.19 Fujitsu's MB14241, perhaps the first dedicated "GPU". 21

1.20 NEC's uPD7220, the first graphics hardware to qualify as a LSI. . . . 21

1.21 Nvidia's GeForce 256. 22

1.22 Comparison of ALUs in a contemporary CPU and GPU. 23

1.23 CPU and GPU evolution, moving towards eventual integration. 23

1.24 A diagrammatic illustration of UMA, NUMA, and what AMD refers to as a Heterogeneous Unified Memory Access (hUMA) architecture. . . . 24

1.25 Utilization of hUMA by AMD's "Kaveri" APU. 25

1.26 Nvidia Tegra X1 specifications and size. 25

1.27 Indicative screenshots of the author's Wavefront .OBJ loader/viewer application. 26

1.28 Screenshot of the author's real-time 3D engine, 2011. 27

1.29 A diagrammatic overview of the developed 3D engine's main system components. 27

2.1 Modularity and reusability in contemporary 3D content creation. . . . 33

2.2 Screenshots of the test data set used in the study. 35

2.3 The data set after import into the testing environment. 38

2.4 Diagrammatic overview of the generic algorithm A1. 41

2.5 Overview of the "initialization" section of the optimized algorithm A2. 45

2.6 Overview of the "per-loop" section of the optimized algorithm A2. . . . 46

3.1 Diagrammatic overview of using collective GPU buffers. 50

3.2 Comparison of the constant buffers employed by the generic algorithm A1 and the optimized algorithm A2. 54

3.3 Illustration of x10 overdraw for TC-06, while TC-05 achieves x1 for computing the same pixel. 57

3.4 Bar chart comparing TC-02 and TC-08. 59

3.5 Bar chart comparing TC-08, TC-11 and TC-12. 60

3.6 An example illustrating the effect of instancing-induced overdraw. . . 61

3.7 Flowchart of main loop with frame limiter added. 64

3.8 Main loop with frame limiter added (code segment). 65

3.9 Bar chart comparing framerate and response time for TC-02 and TC-03. 66

3.10 Bar chart comparing TC-02, TC-03, TC-08 and TC-09, and the apparently anomalous speed decrease of TC-03 compared to TC-02. 67

3.11 Bar chart demonstrating the effect adjusting the frame limit has on rendering speed. 68

3.12 Bar chart demonstrating the effect of pre-WS objects usage on rendering speed by comparing TC-02, TC-07, TC-08 and TC-13. 73

4.1 Collective bar chart comparing the rendering speed of all test cases examined. 77

4.2 Collective bar chart summarizing the study's results by comparing the initial configuration TC-01 (A1) with the optimized TC-14 (A2). 78

5.1 A Nvidia sample showcasing threaded rendering using Vulkan. 80

5.2 Examples of LOD usage. 81

5.3 Examples of the occlusion culling process. 81

5.4 Examples illustrating the triangle reordering process. 82

B.1 The constituent phases of assembling a polygon mesh to be displayed by the GPU. 97

B.2 Structure of the "main loop" of an interactive real-time rendering application. 100

B.3 The sequence of steps for render data to reach the GPU. 101

B.4 Overview of the depth culling process. 104

B.5 The overdraw process. 105

B.6 A visualization of per-pixel shader complexity for a given frame. 106

B.7 Common coordinate systems and spaces for 3D transformations. . . . 107

List of Tables

2.1 The test data set's 3D objects 34

3.1 TC-01 and TC-02 comparison 51

3.2 TC-02 and TC-04 comparison 55

3.3 TC-05 and TC-06 comparison 56

3.4 TC-02 and TC-08 comparison 58

3.5 TC-08, TC-11 and TC-12 comparison 59

3.6 TC-02 and TC-03 comparison 65

3.7 TC-08 and TC-09 comparison 66

3.8 Additional tests demonstrating the effect of modifying the frame limit. 67

3.9 Comparison of TC-02, TC07, TC-08 and TC-13. 72

A.1 Collective table of measurements for all examined test cases. 94