# Profiling und Optimierung Großer Biomolekularer Szenen

BACHELORARBEIT

zur Erlangung des akademischen Grades

**Bachelor of Science**

im Rahmen des Studiums

**Medieninformatik und Visual Computing**

eingereicht von

**Felix Kugler**
Matrikelnummer 01526144

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.techn. Ivan Viola
Mitwirkung: Dipl.Ing Tobias Klein

Wien, 23. August 2018

Felix Kugler    Ivan Viola

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Profiling and Optimization of Large Biomolecular Scenes

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Media Informatics and Visual Computing**

by

**Felix Kugler**
Registration Number 01526144

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.techn. Ivan Viola
Assistance: Dipl.Ing Tobias Klein

Vienna, 23rd August, 2018

| Felix Kugler | Ivan Viola |
|---|---|

# Erklärung zur Verfassung der Arbeit

Felix Kugler
Getreideweg 4, 7062 St. Margarethen im Burgenland

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. August 2018

Felix Kugler

# Kurzfassung

Informationsvisualisierungen und Unterhaltungsmedien verlangen nach Möglichkeiten zum effizienten Rendern immer größerer Szenen. Dank ihrer hoch-parallelen Architektur sind GPUs in der Lage diesen Bedarf zu decken. Mit ihrem Potential steigt jedoch auch ihre Komplexität. APIs wie OpenGL stellen mit jeder weiteren Version mehr Schnittstellen zur Verfügung, um dieses Potential zu nutzen. Diese Schnittstellen effizient zu benutzen wird jedoch immer schwieriger für Programmierer und Anwendungsdesigner. Diese Arbeit versucht eine Hilfestellung im Design solcher Anwendungen darzustellen indem mehrere existierende Methoden und Variationen beschrieben und implementiert und ihre Auswirkungen auf eine praktische Anwendung gemessen werden. Auch wenn einige Methoden zum Rendern von statischer Geometrie im Allgemeinen verwendet werden können, liegt der Fokus auf dem Rendern von biomolekularen Daten als Kugeln auf Billboards.

# Abstract

Scientific visualizations and entertainment purposes demand ways to quickly render larger and larger virtual scenes. Through their highly parallel architecture, GPUs are capable of providing for that demand. But with their processing capabilities, their complexity increases too. With each new version, APIs like OpenGL provide an increasing amount of interfaces to harness the available capabilities. However, using them efficiently can be difficult for programmers and application designers. This work attempts to guide the design of such applications by describing and implementing different existing methods and variations and measuring their impact on the performance of a real-world application. While some techniques are applicable to rendering of static geometry in general, the focus lies on rendering biomolecular data as spheres using billboards.

# Contents

# CHAPTER 1

# Introduction

Rendering large and complex scenes in real-time is the task of many applications, whether it is to visualize large amounts of scientific data or to display detailed virtual worlds for entertainment in video games. As the available processing power increases, so do the demands for increasingly larger scenes. However, GPUs also become increasingly more complex and fully utilizing them can be challenging.

## 1.1 Motivation

GPU architectures provide an increasing amount of functionality which can be used to process and display data on the screen. This process is called rendering. Some steps of this process are programmable and others are built into hardware. Application developers have to decide how to use, and sometimes abuse, the available functionality. Interactive applications need to be able to generate a new image within a few milliseconds (commonly less than  16ms). Therefore, reducing the time it takes to generate a new image is the main goal and the sole measure of performance in the context of this work.

Efforts have been made to design useful models to estimate the performance of algorithms analytically for limited cases[ZO11]. However, this work focuses on empirical measurements in a real-world application.

## 1.2 GPU Rendering

Rendering is the task of generating an image of what a camera in a virtual scene would see. When using APIs like OpenGL or DirectX, this involves one or several *draw calls* and sometimes additional processing steps. The OpenGL nomenclature used here differs from that of DirectX but the functionality is similar. A draw call is the transformation of a numeric description of the scene in terms of points, called vertices, with arbitrary per-vertex information, into the pixel of an image, with arbitrary per-pixel information, usually color. This transformation is split up into multiple steps. Each step processes a stream of input elements into a stream of output elements. The first step's input and last step's output are the per-vertex and per-pixel information respectively. Some steps are programmable, others are fixed, implemented in hardware. Efficient design of rendering applications requires making good use of the available resources and functionality of the GPU. A short overview of the stages of the pipeline follows.

**Vertex Shader** A programmable transformation of the input vertices. Given the attributes of a vertex it produces a new set of attributes to send to the next stage.

**Tessellation Control Shader** An optional programmable stage that allows splitting up input primitives using tessellation by determining the number of primitives to generate over the surface of the input primitive.

**Tessellator** An optional fixed step to subdivide input primitives into smaller ones. Each output vertex has an additional attribute storing its position on the input primitive.

**Tessellation Evaluation Shader** An optional programmable stage to determine arbitrary attributes for each vertex of the primitives the input was split into.

**Geometry Shader** An optional programmable stage to transform each input primitive into an arbitrary, but limited number of primitives.

**Vertex Post-Processing** A fixed step that clips primitives to the screen and transforms them into screen space.

**Rasterization** A fixed step to transform all primitives into the pixel they cover on screen. The per-vertex attributes of the previous step are interpolated over the surface of the primitive. These points on the surface are called fragments.

**Fragment Shader** A programmable step to transform the interpolated attributes into the attributes to be written to the image.

**Depth** An optional fixed step that tests a fragments depth against the depth of the output image to reject fragments occluded by others.

An important aspect of how GPUs work is the fact that they highly parallelize work. Each step is intended to be performed on thousands of elements. For this, GPUs implement wide single instruction multiple data (SIMD), meaning that most instructions for the GPU operate on vectors of data. For example, where processors without SIMD might have an instruction to calculate the sum of a number $a$ and a number $b$, GPUs have an instruction to calculate the sums of multiple pairs of numbers at once. For this purpose the arithmetic units are present multiple times, each working in parallel. To allow this parallelism, the calculations must not depend on each other. This can be ensured by having shaders process elements independently of each other.

In contrast to CPUs, optimizing GPU programs is about finding bottlenecks. Modern GPUs consist of many parallelized components and as such, dependencies between them can lead to underutilization, and as a result, performance loss. As the latency of some processes may be completely hidden by the GPU's pipelining, while others may cause severe stalls, it is important to identify the latter. Although the shading stages depend on each other, this dependency only exists per element. In other words, it is entirely possible, and even likely, that some elements are already being processed by the fragment shader while others are still being processed by the vertex shader. To hide latency, resources underutilized by one stage can be used by other stages. Therefore, optimizing one shading stage may produce no measurable improvement when another stage causes pipeline stalls. Profiling tools are able to measure the utilization of each component and the communication between them, and thereby help to find these bottlenecks.

## 1.3 Biomolecular Scenes

A scene can be described as a collection of objects. Each object has a position, orientation, and arbitrary, application specific, properties, as well as a mesh. The mesh defines the shape of the object as a list of primitives. Most frequently, these primitives are triangles, but lines or points are common as well. They are stored in terms of positions, relative to the objects center and arbitrary, application specific, properties, like color for example. Multiple objects may use the same mesh but draw them at different positions and orientations. The goal is to render the scene, that is, to simulate what a camera placed at a specific position and orientation in the scene would see. The largest scene examined here contains tens of thousands of unique meshes and hundreds of thousands of objects, resulting in tens of millions of primitives.

Some objects like fluids can change their shape and, more importantly, the topologies of their mesh between frames. These present additional challenges. Dynamic shapes were not considered in this work. The topology of the meshes is assumed to be static.

Correct rendering of semi-transparent surfaces may require sorting at the level of primitives or fragments. This is outside of the scope of this work. The scenes examined here only contain opaque objects, as semi-transparent surfaces present additional challenges when determining visibility and occlusion of objects.

In this work various methods for rendering such scenes are described and compared, as well as a number of ways to improve the performance of these methods. The results are demonstrated on the *Marion* framework [MKS$^{+}$17], a C++ implementation of *CellView* [MAPV15], which itself is an application to visualize large biomolecular datasets. In the context of this work, the protein instances of Marion are the objects and the protein types are the meshes. The primitives are spheres, rendered as triangle billboards with correct per-fragment depth for accurate occlusion when zoomed at atomic level. The unaltered version of this program will here be referred to as the *baseline* version. All results are compared to that version.

While many tasks can be performed much faster on GPUs than on CPUs, the additional overhead of transferring input data to the GPU and results back to the CPU over the bus can outweigh the advantage in processing speed[GH11]. This work focuses on applications that are able to transfer the entire scene to the GPU memory once, at start-up, and then render it many times from different views and with different settings. Displaying the result on screen does not require a transfer back to the CPU either. Therefore communication overhead is negligable in this application.

Level of Detail (LOD) is used to adjust the amount of detail in each object based on its distance to the camera. Objects closer to the camera are rendered with more detail than objects further away. If the geometry can be described procedurally it can be generated for each frame in the desired amount of detail. It is also possible to pre-compute or create multiple versions of each mesh with varying amounts of detail and switch between them. The latter has been implemented in Marion. Each protein type consists of multiple point-clouds with varying number of points.

The methods outlined here are applicable to scenes of comparable size and, although only billboards are rendered, are applicable to arbitrary geometry unless noted otherwise. Additionally, while only OpenGL was used here, DirectX offers all relevant functionality.

## 1.4 Contribution

This work aims to assist the decision processes during the design of rendering applications by taking a real-world application and comparing the render time between different variations in the process. The work explors different configurations of the depth test for billboard rendering, the cost of per-fragment attributes, the cost of redundant calculations and different ways to implement a level of detail system.

CHAPTER 2

# Profiling GPU Programs

Zhang and Owens [ZO11] have presented a model of the GPU architecture in a way that allows assessing performance bottlenecks through code analysis, simplifying the process of optimizing code. Stephenson et al. [SSHL$^{+}$15] present a profiling tool based on code instrumentation. While useful, both approaches are only concerned with individual computations (as CUDA or OpenCL kernels) and ignore the interactions with the fixed hardware functions provided for graphic rendering.

By comparing the time of rendering a test scene through different APIs, Dobersberger [Dob15] performs a similar role in the design of rendering applications as this work. By performing measurements on a real-world application this work can be seen as an extension of some of the work performed by Dobersberger. In contrast to Dobersberger, this work does not focus on CPU time, as it has been found to be negligible compared to the GPU time for the Marion application. Neither are texture look-ups examined here.

Modern GPUs provide a large number of built-in measurements that can be used to find performance issues. Some of these can be read using tools like Nvidia Nsight [nsi] and RenderDoc [ren] or through the OpenGL implementation [RSR13]. This work only focuses on these measurements.

RenderDoc provides information about the duration of calls on the GPU, the number of invocations of each shader, and the number of primitives and fragments generated. Optimizing the operations with the largest GPU time and the shaders with the most invocations leads to the highest amount of expected performance gain. It should be noted that although the documentation does not explicitly say so, experiments show that the number of pixel shader invocations includes fragments discarded by the depth test.

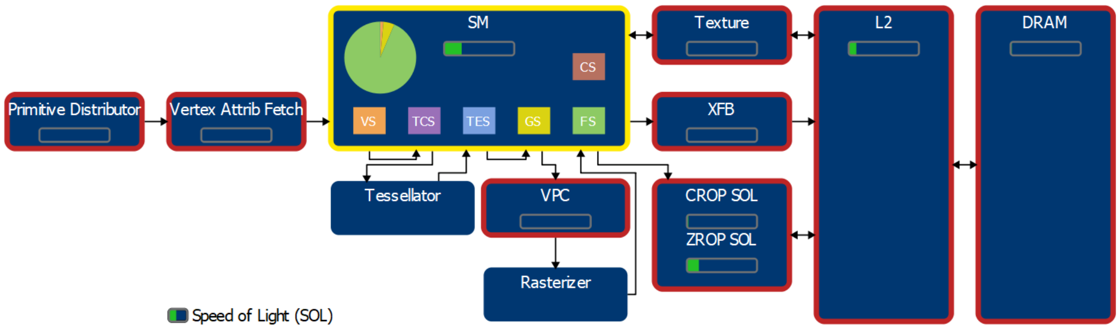

Figure 2.1: Screenshot of the *pipeline overview* in *Nvidia Nsight*. The different components of the GPU are shown as rectangles. The green bars represent the ratio between the amount of work processed by a unit compared to the theoretical amount of work it can process, called the *speed of light*. In this example, most units are underutilized suggesting that they spend a lot of time waiting on each other. Additionally, a pie chart shows the percentage of time the streaming multiprocessors spend on each program stage. Here the majority of the time is spent on the fragment shader.

Additionally to the number of invocations, Nsight also provides information about the load of each hardware component. Among others, these components are the streaming multiprocessors (SM) performing the calculations within a shader, the render output units (ROP) performing per-sample tests, and writing out the results of the fragment shader, and the components of the memory-subsystem consisting of the texture unit (TEX), cache (L2) and DRAM. A diagram of these components, as it is visualized in Nsight, can be seen in Figure 2.1. They display the ratio between the amount of work a component has to process compared to the theoretical amount it could process, called the speed of light, as a percentage (SOL%). Generally, components with a low SOL% are under-utilized. By moving work from a component with high load to one with small load the performance can be improved.

Load can be moved from the streaming multiprocessors to the memory-subsystem by storing results of calculations where possible and reading them from memory rather than re-calculating them. Vice versa, it is possible to move load from the memory-subsystem to the streaming multiprocessors by re-calculating values instead of storing them. Note that the render output units too can be memory bound, therefore reducing the number and size of values written by the fragment shaders can have the same benefits. This approach is explained further in Section 3.3.

Another way to reduce the load on the render output units, as well as memory, is to avoid overdraw by rendering fragments front to back and thereby allowing more fragments to be discarded. Where an early depth test is possible, this can also reduce the load from the fragment shader. More information on this solution can be found in Section 3.1.

All of the available values can aid in the discovery of bottlenecks. When comparing the performance of different versions, only GPU durations are presented here. These are generally different from the duration of calls as seen by the CPU as modern GPUs perform pipelining and batching to increase throughput. That is, calling API functions only puts tasks into a queue. The GPU duration is measured by the GPU and only includes the time actually spent processing.

CHAPTER 3

# Rendering Optimization

The methods and variations in the rendering process that were tested are described in the following sections. Their effect on the GPU duration of the molecule rendering in Marion were measured. A summary of all improvements is provided in Chapter 4. Throughout this work, a small test scene as well as a large scene of a biomolecular dataset were used to analyze the effect of different aspects of the scene.

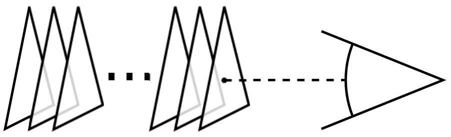

Figure 3.1: The "stack" scene has the virtual camera look onto a stack of 10,000 primitives in front of each other, leading to a large amount of overdraw.

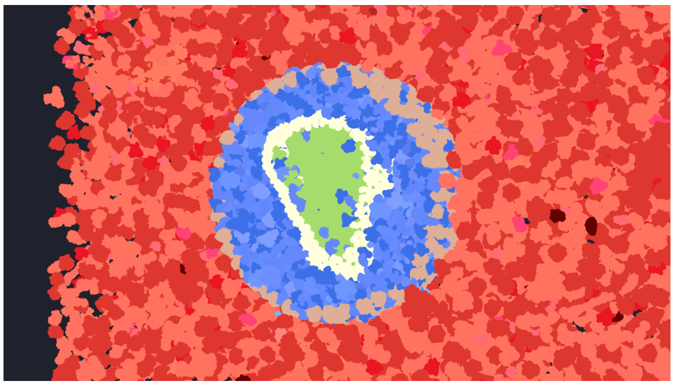

Figure 3.2: The "HIV" scene has the virtual camera look at a cutaway view of an HIV cell in blood plasma.

The first scene, the "HIV" scene shows an HIV cell in blood plasma. It consists of 35 thousand objects and a total of 2.8 million primitives. A screenshot of this scene can be seen in Figure 3.2. This scene was used as a starting point for further investigation, as it represents a real-world use case of the Marion application.

Early profiling showed a large amount of overdraw, that means the same pixels are being set multiple times by different invocations of the fragment shader. To measure the impact of overdraw, as well as methods to avoid it, the "stack" scene was created. It shows a stack of 10,000 spheres orthogonal to the view plane such that they occlude each other and only one sphere is visible (see Figure 3.1). This arrangement causes a large amount of overdraw.

All measurements were performed using a *Nvidia GeForce GTX 970* GPU on a machine running *Windows 10*. All renderings were done in a resolution of 1920 by 1080 pixels in full-screen mode.

## 3.1 Reducing Overdraw

When multiple opaque triangles occupy the same pixel, the fragment shader may need to be evaluated once for each triangle even though only the one closest to the camera will be visible in the final result. This is called *overdrawing*. Performance gains are possible if it can be determined beforehand whether a fragment will be occluded by others.

### 3.1.1 Depth Sorting

Rendering geometry such that primitives closer to the camera are rendered before primitives further away from the camera (front to back) can allow the GPU to discard fragments as they can be tested against fragments already written. This is called *early depth test*. Ideally, only a single fragment needs to be evaluated for each pixel. Unfortunately, the order at which primitives would have to be rendered depends on the camera position and may change between frames. Sorting the primitives every time may be unfeasible for large scenes.

| | back-to-front | front-to-back |
|---|---|---|
| default depth | 9.9 ms | 5.1 ms |
| per-fragment depth | 9.9 ms | 5.5 ms |
| + conservative depth | 9.6 ms | 5.5 ms |

Table 3.1: Effect of the order of elements on the GPU duration for the protein program in Marion when rendering the stack scene.

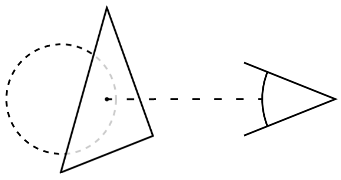

Figure 3.3: A triangle, completely covering the area of the implicit object for the virtual viewer, is drawn. For each fragment of the triangle, the visibility and depth of the implicit object behind it are calculated.

To evaluate the potential of this approach, the "stack" scene, a simple test scene with 10,000 primitives at the center of the screen and varying depth, was rendered once front-to-back and once back-to-front. The GPU durations are compared in Table 3.1.

When rendering objects front-to-back the GPU duration is reduced by 43%. Although this is an extreme example, it suggests that sorting the objects can provide a performance gain. In combination with the approach outlined in Section 3.4.2, the order can be trivially communicated to the draw call, as commands are rendered in the order they are placed in the buffer storing the draw commands.

### 3.1.2 Early Depth Test for Implicit Objects

Spheres and other round objects are difficult or expensive to render as triangle meshes but can be defined implicitly. For these objects it is possible to render them precisely and relatively cheaply using the fragment shader. The Marion application depicts proteins as clouds of spheres. These can be drawn by drawing a triangle that fully contains the sphere on screen (called the impostor) and calculating a ray-sphere-intersection, or approximation thereof, to discard fragments outside of the sphere and get the depth for fragments inside of it.

When the depth values are only known after the fragment shader it is not possible to perform the depth test before the fragment shader. However, it is possible to specify a hint to the GPU, specifying that the fragment shader will only ever increase the depth of the fragment relative to its default depth, making it safe to discard fragments whose default depth already fails the depth test. This is called *conservative depth test.* Care must be taken to ensure that the actual object is completely behind the impostor, that

| | GPU duration |
|---|---|
| default depth | 162 ms |
| per-fragment depth | 170 ms |
| per-fragment depth, conservative depth | 170 ms |
| default depth, no discard | 23 ms |
| per-fragment depth, no discard | 190 ms |

Table 3.2: GPU duration for the protein program in Marion for the "HIV" scene with different variations of the fragment shader.

is, every fragment of the object has a greater depth than the corresponding fragment of the impostor. To achieve this when rendering spheres, radius of the sphere is applied as an offset to the z position of the triangles. A visualization of the implicit sphere and the corresponding triangle can be seen in Figure 3.3.

The depth values stored by the GPU are understood to be in normalized device coordinates, that is, they are in the range $[-1, 1]$ and not linear. The default depth is calculated from clip space coordinates $(x, y, z, w)$ and clipping planes $near$ and $far$ as

$$\frac{z}{w} \cdot (far - near) \cdot 0.5 + (near + far) \cdot 0.5$$

Rendering the "stack" scene both front-to-back and back-to-front shows that the latter takes almost twice as long. When rendering back-to-front all fragments always pass the depth test, independent of whether the default depth was used or not. However, when rendering front-to-back, using non-default depth does incur a performance penalty as fragments can only be discarded after their depth has been calculated. In this scene the penalty amounts to a 8% increase in render time. Using conservative depth does not lead to a large performance gain. All timings can be seen in Table 3.2

The "HIV" scene shows similar results. Rendering with non-default depth is slightly slower than rendering with default depth. The rendering time is increased by 5%. Using conservative depth does not significantly change the rendering time.

Discarding fragments in the fragment shader seems to be very costly. Avoiding it reduces render time by 86%. However, without it, the triangle shape of the primitives becomes visible. Instead of discarding fragments, the same effect can be achieved by setting the depth of all fragments outside of the implicit object to infinity. This, however, is slower than every other attempt. 12% slower than the baseline version.

## 3.2 Loop-Invariant Code Motion

A simple way to reduce the amount of processing required is to remove unnecessary computation. A simple way of achieving this is to move code out of loops. These calculations, that do not depend on any calculation inside the loop, are called loop-invariant.

OpenGL programs consist of a sequence of shaders that perform some kind of transformation on elements. Generally, each consecutive shader operates on a larger set of elements than the one before. The vertex shader operates on vertices or patches, the tessellation evaluation shader on vertices on the patch, the geometry shader transforms each vertex into an arbitrary number of vertices and the fragment shader operates on the fragments making up the primitives. This structure can be seen as a series of nested loops where each shader is one nesting level. As such, loop-invariant code motion can be applied between the shaders by moving calculations from later shaders to earlier ones.

Many operations used in 3D rendering are linear, such as matrix multiplication and linear interpolation. Shaders perform matrix multiplications to transform values. These values are then linearly interpolated by the tessellation evaluation shader and before the fragment shader. Therefore, linear transformations after the interpolation can be moved before the interpolation to reduce the number of calculations necessary.

To calculate the correct depth, Marion passes the view space coordinates to the fragment shader, adds an offset to the position and transforms the result into clip space. As the addition of the offset is a linear operation it is possible to move the transformation into clip space to the previous shader stage, the geometry shader. This is possible because when given view space position $v$, view space offset $o$ and projection matrix $P$ the following formular holds true:

$$P \cdot (v + o) = P \cdot v + P \cdot o$$

The result of $P \cdot v$ can be calculated in the previous shader stage and be linearly interpolated. As the offset $o$ is 0 for all components but $z$, the matrix multiplication $P \cdot o$ can be reduced to $p_{*,3} \cdot o$ where $p_{*,3}$ is the third column of the matrix $P$. Thereby the amount of computation is reduced.

In the geometry shader, Marion generates triangle billboards for each input vertex. Here the same logic can be applied. Instead of calculating the positions of the three vertices making up the triangles in view space and then transforming them individually into clip space, it is possible to transform the center into clip space and then apply offsets to it.

| clip space transformation | GPU time |
|---|---|
| per fragment | 170 ms |
| per billboard vertex | 47 ms |
| per billboard center | 40 ms |

Table 3.3: GPU time of the protein program in Marion when rendering the "HIV" scene. The projection of point coordinates into clip space was performed at different steps in the pipeline.

Performing the transformation for each fragment is very slow. By moving it out of the fragment shader and into the geometry shader the render time is reduced by 72%, a speed up by a factor of more than 3. Calculating the positions of the billboard vertices in clip space instead of view space reduces the render time by another 15%. All timings can be seen in Table 3.3.

## 3.3 Memory Bandwidth Utilization

GPUs allow rendering an arbitrary number of per-pixel information. Additionally to color this may include object ids or surface normals. Different data types are available for these per-pixel values. They differ in the number of components, value range, precission and their size in memory. The speed at which data can be read from memory is limited. By reducing the size of this per fragment information, the memory bandwidth can be better utilized. This has shown to yield a large improvement in performance in the Marion application. Suggestions on how to store commonly required information follow.

### 3.3.1 Trade-Off between Storage and Accuracy

The size of values in memory can be reduced by storing them with less precision. However, doing so can lead to visual artifacts in the final result. Different types can be tested to find the lowest precision that still yields acceptable results.

When the range of the values is known beforehand, fixed point formats make better use of the available precision than floating point formats. The components of normal vectors for example only have values in the range of -1 to 1 which makes them a perfect candidate for values that should be stored as normalized integers. These are integer values that are implicitly scaled to the range of $[-1, 1]$ by the GPU. Depending on the required precision one can use 8, 16 or 32 bit integers.

### 3.3.2 Redundancy

If the memory access is the bottleneck, it can be beneficial to trade memory access for computation. If a value can be calculated from other values, it is not necessary to store it. For example, when storing unit vectors on a hemisphere it is enough to store two

| atom id | instance id | position | normal | size | GPU time |
|---|---|---|---|---|---|
| 4x 32 bit float | 4x 32 bit float | 4x 32 bit float | 4x 32 bit float | 64 byte | 170 ms |
| 16 bit integer | 16 bit integer | 4x 32 bit float | 4x 32 bit float | 36 byte | 105 ms |
| 16 bit integer | 16 bit integer | 3x 32 bit float | 2x 32 bit float | 24 byte | 105 ms |
| 16 bit integer | 16 bit integer | 3x 16 bit float | 2x 8 bit integer | 12 byte | 56 ms |
| 16 bit integer | 16 bit integer | not stored | not stored | 4 byte | 45 ms |

Table 3.4: GPU time in Marion for different output types when rendering the HIV scene. The output size is per fragment.

components as the third one can be calculated using the following equation:

$$z = \sqrt{1 - x^2 - y^2}.$$

A comparison of the encoding error and performance impact of more advanced methods to store unit vectors has been conducted by Zina et al. [CDE+14]. Additionally, storing fragment normals can be avoided completely as they can be approximated using the derivatives of the depth buffer if the accuracy is sufficient for the task.

Another example for redundant information are world positions of fragments. Instead of storing them explicitly they can be calculated from a pixel's position in screen space, its depth and the view properties.

It is not always necessary to store positions as 4-component homogeneous coordinates. 3-component Cartesian coordinates suffice if the fourth component is always 1. A similar reasoning can be applied to color, where it may not be necessary to store an alpha channel. Although this does reduce the memory requirements, it did not measurably increase the performance of Marion.

The baseline implementation of Marion exclusively uses 4-component 32 bit float vectors. Atom and instance ids are each stored in the first component of a vector. The first step is to replace them with 16bit integers. This already yields an improvement of 38%. Reducing the number of components in the positions and normals does not yield an improvement. However, by reducing their accuracy from 32 bit float to 16 bit float an additional reduction in render time by 47% is measured. Removing the position and normal outputs completely, as they can be calculated from fragment positions, fragment depth and the parameters of the camera, reduces the render time by another 20%. Compared to the existing implementation of Marion, an improvement of 74% is achieved. Exact timings can be seen in Table 3.4. These measurements are taken on the HIV scene.

## 3.4 Level of Detail

It can become infeasible and also unnecessary to always render all objects at full detail. In the case of visualization it is sometimes even unwanted to keep too much detail, as

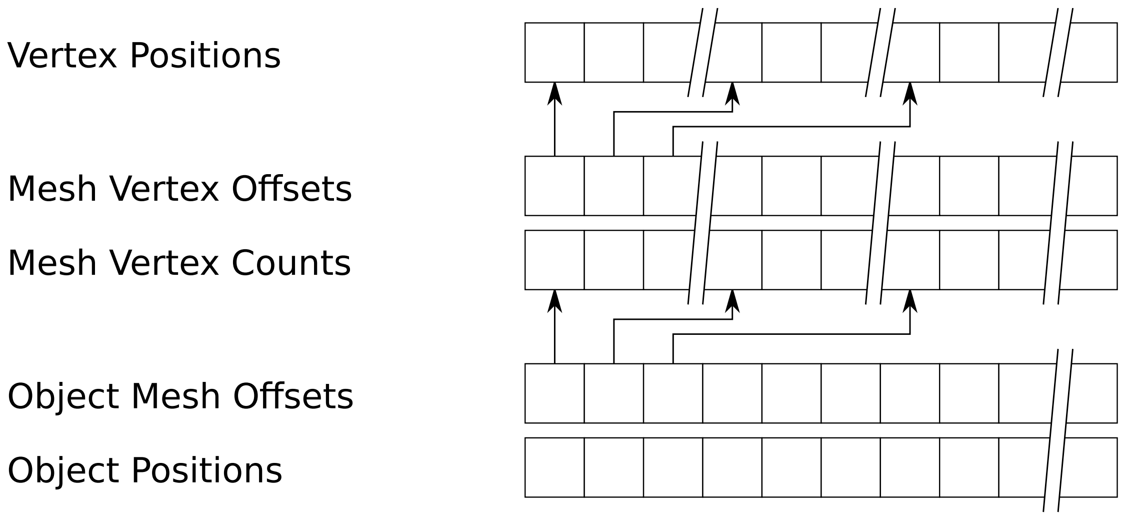

Figure 3.4: Visualization of the memory layout of a scene in Marion. For each mesh, the index of the first vertex and the number of vertices is stored. Similarly, as each object can have several meshes, depending on the LOD, the index of the first mesh for each object is stored. Both the vertices for each mesh and the mesh for each object need to be stored continuously to allow efficient addressing.

it increases the amount of visual clutter. Level of Detail (LOD) methods are used to change the amount of detail in an object depending on its distance to the camera. As the camera changes, the desired amount of detail in each object changes.

For geometry that can be described as a collection of mathematical surfaces or curves the GPUs tesselator can be used to generate a view dependent number of triangles or lines. It is also possible to store different versions of each mesh and select the appropriate one per object. Selecting an appropriate mesh for each object is an additional step that needs to be taken during rendering. This can be performed by one or several compute shaders before the draw call.

Before going into how the scene is rendered, an overview of how the scene is stored is given. As the scene in Marion is static, it is uploaded to GPU memory once and resides there over the runtime of the application. All information about the scene is stored in several buffers. The vertex positions of each mesh are stored consecutively in a single buffer. An additional buffer stores the index of the first vertex and the number of vertices for each mesh. A third buffer stores for each object the index of its mesh. Two more buffers store object positions and orientation. See Figure 3.4.

The naive approach is to select the mesh for each object on the CPU and issue an individual draw call for each. In this approach, a call to the OpenGL function `glDrawArrays` with the appropriate parameters is issued once for each object. By using fewer (usually more complex) calls instead of many small ones, the performance can be improved as each call has a certain amount of overhead[Dob15].

### 3.4.1 Tessellation

One method that can be used to change the geometry within a draw call based on the current view is tessellation. Although this technique is designed to render procedural surfaces that can be arbitrarily interpolated, it can also be used to implement mesh slection[LMPSV14]. This approach is implemented in the baseline version of Marion with some variations.

When using tessellation, the primitives submitted to the GPU are patches. A patch consists of an arbitrary but constant number of vertices. For each patch vertex, the tessellation control shader is executed to perform transformations on the vertex attributes, and determine the tessellation level, the number of primitives generated for the patch. Then, for each primitive generated, the tessellation evaluation shader is executed to determine the attributes of the primitives to be passed to the next shader. Generally, patches define 2-dimensional surfaces. The tessellation control shader determines the number of vertices to generate along the two surface dimensions, and the tessellation evaluation shader determines the properties for each point on the surface given its two dimensional position on the patch.

The type of the patch defines how primitives are generated from that surface. It is not possible to directly control the topology of the mesh generated for the patch. The topology is irrelevant when rendering points. In isoline mode the tessellator generates $u \cdot (v + 1)$ points for a patch with tesselation levels $u$ and $v$ along the two dimensions of the patch.

To use tessellation to switch between meshes, a patch for each object is created. The tessellation control shader selects the mesh and sets the tessellation level. The number of tessellation steps across the two dimensions of the patch is set to $\lceil \sqrt{n} \rceil$ with n being the number of vertices in the mesh. This ensures the generation of a sufficient number of vertices. The vertices are spread evenly over the surface. Therefore, it is possible to determine a unique index for each vertex based on its surface position. Using that position, the tessellation evaluation shader looks up the appropriate vertex attributes from a buffer.

An advantage of this method is that all objects can be drawn with a single draw call, keeping the driver overhead small. However, the number of primitives that can be generated for a single patch may be insufficient for some applications. The specification only requires support for a tessellation level of up to 64 for both dimensions, resulting in a maximum of $64 \cdot 65 = 4,160$ vertices. This limitation can be bypassed by generating more than one sphere in the geometry shader[LMPSV14]. Another solution is to split up objects that exceed that limit before the draw call. The latter is implemented Marion.

The splitting of objects can be solved using a compute shader as a preprocessing step. This shader is called for each object, performs the mesh selection, and groups the resulting stream of vertices into batches, small enough for the tessellation shader to handle. Each batch represents a range of vertices by storing the index of the first and the number of vertices. Additionally, an object id is stored for each batch. Each shader invocation determines the number of batches needed for its object and uses an atomic add operation on a shared variable to determine a starting index to write its batches to. The following GLSL function performs this step:

```
void main() {
    int objectId = int(gl_GlobalInvocationID.x);
    int lod = determineLod();

    int meshOffset = objectMeshOffsets[objectId];
    int meshId = meshOffset + lod;

    int vertexOffset = meshVertexOffsets[meshId];
    int vertexCount = meshVertexCounts[meshId];

    int batchCount = 1 + (vertexCount - 1) / MAX_BATCH_SIZE;
    int batchOffset = atomicCounterIncrement(batchCount);

    int batchStart = 0;
    for (int i = 0; i < batchCount; i++) {
        batchSize = min(
            MAX_BATCH_SIZE,
            vertexCount - batchStart
        );
        batchVertexOffsets[batchOffset + i] =
            vertexOffset + batchStart;
        batchVertexCounts[batchOffset + i] = batchSize;
        batchObjectIds[batchOffset + i] = objectId;
        batchStart += MAX_BATCH_SIZE;
    }
}
```

After the batching process, the scene can be rendered. The following tessellation control shader sets the tessellation level according to the batches:

```
layout(vertices = 1) out;

in int batchId[];

patch out int vertexOffset;
patch out int vertexCount;
patch out int objectId;

void main(void) {
    vertexOffset = batchVertexOffsets[batchId[0]];
    vertexCount = batchVertexCounts[batchId[0]];
    objectId = batchVertexObjectIds[batchId[0]];

    gl_TessLevelOuter[0] = ceil(sqrt(count));
    gl_TessLevelOuter[1] = ceil(sqrt(count)) - 1;
}
```

As this tessellation control shader will only produce a square number of vertices, it generally generates too many. The superfluous vertices can be discarded by the geometry shader later. How to calculate the actual vertex id from the patch coordinates can be seen in the following tessellation evaluation shader:

```
layout(isolines, point_mode) in;

patch in int vertexOffset;
patch in int vertexCount;
patch in int objectId;

void main() {
    vec2 tessCoord = gl_TessCoord.xy;
    int x = int(round(tessCoord.x * gl_TessLevelOuter[1]));
    int y = int(round(tessCoord.y * gl_TessLevelOuter[0]));
    int vertexId =
        vertexOffset + x + y * int(gl_TessLevelOuter[1] + 1);

    // write shader outputs using vertexId and objectId
}
```

### 3.4.2 One Draw Command per Object

OpenGL offers an API to specify a list of draw commands residing as a buffer on the GPU, allowing an arbitrary number of meshes to be drawn with a single CPU call. This is called *multi draw* and can be achieved using the function `glMultiDrawArraysIndirect`. Further, this list can be generated using a compute shader before the draw call. This is very similar to the approach of using multiple calls to `glDrawArrays` but makes it possible to select the mesh appropriate for the object's distance on the GPU, and thereby avoid the driver overhead. It also does not have any of the limitations of tessellation.

The commands are structs of four values of type uint. The first and second value store the number of primitives and the number of instances to be drawn, respectively. The third and fourth represent the index of the first primitive and the index of the first instance, respectively. Essentially, a draw command stores two ranges of indices. The Vertex shader will be executed once for each combination of indices in these two ranges. The number of instances can be set to zero to omit rendering of a command. This can be used to apply culling. The buffer of draw commands is defined in GLSL as follows:

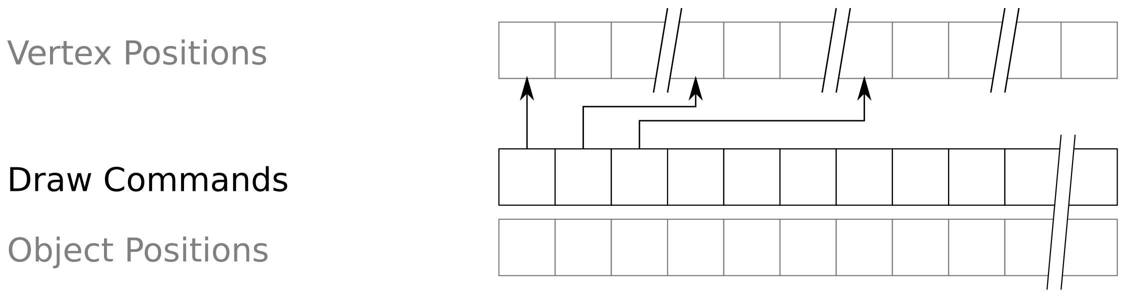

Figure 3.5: The draw command buffer has one entry for each object and stores the index of the first and number of vertices in the vertex buffer, according to the level of detail selected during preprocessing.

```
struct DrawCommand {
    uint count;
    uint instanceCount;
    uint first;
    uint baseInstance;
};

buffer DrawCommands {
    DrawCommand drawCommands[];
};
```

The `DrawCommand` struct can be replaced with a `uvec4`. Although it should not make a difference, it seems as if the GLSL compiler is not able to reliably optimize memory writes to structs, causing the shaders with such writes to sometimes, but not always, be slower by a factor of 20. In all tests, a `uvec4` was used.

A simple way to generate the buffer of draw commands is to execute a compute shader that writes a draw command for each object. The draw command contains the index of the first and the number of primitives of the mesh in the vertex array. The result of this step is visualized in Figure 3.5. These values can be set to point to the mesh appropriate for the distance between the object and the camera by running the following GLSL function for each object:

```
void main() {
    int objectId = int(gl_GlobalInvocationID.x);
    int lod = determineLod();
    int meshOffset = objectMeshOffsets[objectId];
    int meshId = meshOffset + lod;

    drawCommands[objectId].count =
        meshVertexCounts[meshId];
    drawCommands[objectId].first =
        meshVertexStarts[meshId];
    drawCommands[objectId].instanceCount = 1;
    drawCommands[objectId].baseInstance = objectId;
}
```

This approach faces none of the limitations caused by the tessellation shader and also achieves good performance.

### 3.4.3 One Draw Command per Mesh

A single draw command may specify multiple instances of the same mesh. Therefore, instead of having a draw command for each object, it is possible to have one for each mesh with one instance for each object using that mesh. As the actual mesh of each object depends on its distance to the camera, this requires additional compute shaders before the draw call.

Simiarly to per-vertex information, per-instance information for each mesh is written consecutively into a buffer. This allows addressing the instances of a mesh using the index of the first instance and the number of instances of that mesh. This is implemented in three stages. At first, a compute shader runs once per object to determine the appropriate mesh depending on the objects distance to the camera. Atomic increment operations are used to count the number of objects per mesh. The second compute shader is executed per mesh and uses atomic increment operations to calculate an offset at which the instances of the mesh can be written to. A third compute shader runs per object and writes its index in the aforementioned buffer. The last two steps essentially perform a stream compaction. Afterwards, it is possible to look up an object's position and orientation from a buffer using the index.

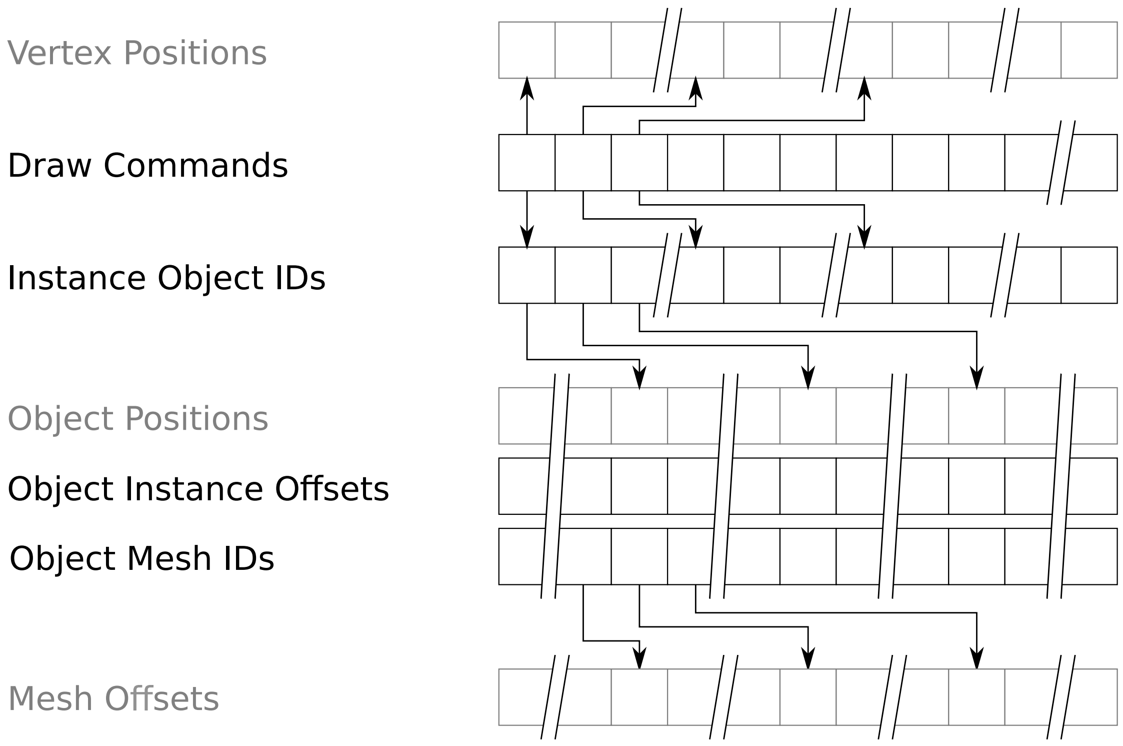

Figure 3.6: Visualization of the memory layout of the auxiliary buffers used for rendering. Draw commands refer to ranges of vertices and instances. For each instance, the corresponding object id is stored. For each object the id of the appropriate mesh for the current frame is stored. Buffers already shown in Figure 3.4 are shown in gray for context.

This method requires some additional buffers. First of which stores the id of the lod-appropriate mesh for each object as an integer. A second buffer maps instance ids to object ids to allow the vertex shader to look up object information using the instance id. Lastly, a single integer value to determine the instance offset for each mesh is stored. A visualization of the additional buffers can be seen in Figure 3.6.

Before preprocessing, the draw command buffer as well as the instance offsets are cleared to zero. Afterwards, stage one, determining the mesh for each object and counting the occurrences of each, can be achieved by running the following GLSL function for each object:

```
void main() {
    int objectId = int(gl_GlobalInvocationID.x);
    int lod = determineLod();
    int meshOffset = objectMeshOffsets[objectId];
    int meshId = meshOffset + lod;

    objectMeshIds[objectId] = meshId;
    objectInstanceOffset[objectId] =
        atomicAdd(drawCommands[meshId].instanceCount, 1);
}
```

The instance count member of the draw commands is used for counting. Whether the improved locality from using an additional buffer for it improves performance has not been tested.

Secondly, to determine the instance offset for each mesh, this GLSL function is run for each mesh:

```
void main() {
    int meshId = int(gl_GlobalInvocationID.x);

    int offset = atomicAdd(
        instanceOffset, drawCommands[meshId].instanceCount
    );

    drawCommands[meshId].baseInstance = offset;
    drawCommands[meshId].first = meshVertexOffsets[meshId];
    drawCommands[meshId].count = meshVertexCounts[meshId];
}
```

Lastly, the object ids are written into the instance object id buffer by running the following GLSL function for each object:

```
void main() {
    int objectId = int(gl_GlobalInvocationID.x);
```

| version | preprocessing | rendering |
|---|---|---|
| tessellation | 0.03 ms | 170 ms |
| one draw command per object | 0.09 ms | 152 ms |
| one draw command per mesh | 3.2 ms | 148 ms |

Table 3.5: GPU duration of the preprocessing and the protein program in Marion for different rendering methods.

```
    int meshId = objectMeshId[objectId];
    int instanceId =
        drawCommands[meshId].baseInstance +
        objectInstanceOffset[objectId];

    InstanceObjectId[instanceId] = objectId;
}
```

Using instancing, Dobersberger is able to reduce the render time of a test scene from 13 ms to less than 1 in their setup. [Dob15] However, it should be noted that they used a much simpler fragment shader than Marion. This amount of speed-up is not expected in the scenario where the fragment shader dominates the render time.

When rendering the HIV scene using the multi-draw API, it was possible to reduce the render time by 11% compared to the version using tessellation. The improvement is slightly diminished by the increased time required to preprocess the scene. The atomic operations in the preprocessing steps can harm parallelism. Algorithms for stream compaction that make better use of parallel hardware exist. However, this has been left for future work. In many cases, having one draw call per object is probably the best solution as it is almost as fast as the fastest method but also very simple to implement and maintain. Timings for each method can be found in Table 3.5

CHAPTER 4

# Results

The methods described in Chapter 3 were implemented and tested both individually and in every combination. In total the render time in the Marion application for the HIV scene was reduced from 170 ms to 35 ms, that is, by 79%. The timings for each version is shown in Table 4.1.

Each optimization shows a large improvement on its own, but the improvements are not cumulative. After optimizing the fragment shader, either by moving the projection out of it or by reducing the size of the per-fragment information, the other optimizations only produce minor gains. This is to be expected, as most of the load is caused by the fragment shader.

| version | preprocessing | protein program |
|---|---|---|
| baseline | <0.1 ms | 170 ms |
| reduced output (Section 3.3) | <0.1 ms | 45 ms |
| multi draw (Section 3.4.2) | <0.1 ms | 152 ms |
| instanced rendering (Section 3.4.3) | 3 ms | 148 ms |
| early projection (Section 3.2) | <0.1 ms | 40 ms |
| reduced output + instanced rendering | 3 ms | 41 ms |
| instanced rendering + early projection | 3 ms | 74 ms |
| reduced output + early projection | <0.1 ms | 40 ms |
| reduced output + multi draw + early projection | <0.1 ms | 35 ms |
| reduced output + instanced rendering + early projection | 3 ms | 40 ms |

Table 4.1: GPU duration for the preprocessing step(s) and the protein program, using the methods described in Section 3. These measurements are taken from the HIV scene.

Interestingly, when combined with the optimized protein program, having one draw command per object is actually faster than having one per mesh. This is still true when ignoring the larger time required for preprocessing which could be avoided through the use of a parallel stream compaction algorithm. This may be caused by the additional indirection in the vertex shader, as object ids need to be looked up in the instance object id buffer. Therefore, this method is not only straight forward to implement but also very efficient. This also supports what has been found by Dobersberger[Dob15]. For best performance, applications should render all their objects with as few multi-draw commands as possible.

CHAPTER 5

# Future Work

These are additional methods and ideas that can improve the performance but are not implemented.

## 5.1 Occlusion Culling

A method to remove the number of primitives being rendered that is left unexplored is to determine whether they are occluded by others. Whereas the depth test rejects occluded fragments, occlusion culling removes entire primitives or objects based on whether they are occluded by others. One way of achieving this is by partitioning the scene into subspaces and issuing occlusion queries for the bounding boxes of each partition against an estimated depth map. Occlusion queries are an API provided by OpenGL to test the visibility of objects. The estimated depth map can be generated from the previous frame combined with information about the camera movement[GKM93]. If the bounding box of a subspace is completely invisible, every object within is completely invisible as well. As the reverse does not hold true, this only allows culling of some objects. Grottel et al. [GRDE10] use this approach to render large numbers of particles. Another way to generate such an estimated depth map is to render all objects in a pre-pass with a simplified fragment shader.

The estimated depth map can also be used to discard more fragments during the early depth test in the final render pass. As the fragment shader is the most costly operation in Marion, allowing more fragments to be discarded early could increase performance.

### 5.1.1 Frustum Culling

It can be determined that an object is completely outside of the view frustum and, therefore, has no effect on the final image. This can be done by storing simple bounding volumes, one for each object, that can be tested against a frustum. Popular choices are cuboids and spheres as they can be tested with very few calculations. Assarsson and Möller [AM99] explore various algorithms for achieving this. When using a LOD method, the frustum test can be combined with the mesh selection for little additional cost.

The performance gain achieved with this method depends on the number of objects that can be culled. When exploring a scene from inside, as it is common in video games, a large portion of the scene can be culled, as the camera frustum is only a small subspace. If the objects are almost all in front of the camera, few objects can be culled.

As the GPU already does not generate fragments for primitives outside the view frustum, this method only helps to reduce the number of vertex shader invocations. Little gains are expected in Marion, since the fragment shader takes up the majority of the render time.

## 5.2 Simplifying Primitives at a Distance

The number of primitives for objects at a distance can be reduced using LOD. Additionally, it is possible to simplify the primitives themselves when viewed from far away, as their precise shape and appearance will not be as important. The per-fragment depth allows correct occlusion between intersecting primitives. This effect is barely noticeable at a distance and could therefore be turned off to reduce the amount of necessary processing in the fragment shader. For the same effect, primitives can be replaced with simpler ones. Impostors can be replaced with simple triangles, triangles can be replaced with points.

## 5.3 Pre-Ordering Objects

Rendering can be sped up by avoiding overdraw. If it can be estimated beforehand how users gonna interact with the scene, it is possible to order the objects such that ones that are more likely to be in front of others appear first in the list of objects. For example, the objects of the HIV scene could be ordered such that objects further away from the center are rendered first, as they tend to occlude objects closer to the center.

# Bibliography

[AM99] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms. *Technical Report 99-3, Chalmers University of Technology, Sweden*, 1999.

[CDE$^+$14] Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT)*, 3(2):1–30, 2014.

[Dob15] Simon Dobersberger. Reducing driver overhead in opengl, direct3d and mantle. *University of Applied Sciences Technikum Wien*, pages 2–32, 2015.

[GH11] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144, 2011.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 231–238. ACM, 1993.

[GRDE10] Sebastian Grottel, Guido Reina, Carsten Dachsbacher, and Thomas Ertl. Coherent culling and shading for large molecular dynamics visualization. *Computer Graphics Forum*, 29(3):953–962, 2010.

[LMPSV14] Mathieu Le Muzic, Julius Parulek, Anne-Kristin Stavrum, and Ivan Viola. Illustrative visualization of molecular reactions using omniscient intelligence and passive agents. *Computer Graphics Forum*, 33(3):141–150, 2014.

[MAPV15] Mathieu Le Muzic, Ludovic Autin, Julius Parulek, and Ivan Viola. cellview: a tool for illustrative and multi-scale rendering of large biomolecular datasets. In Katja B editor, *Eurographics Workshop on Visual Computing for Biology and Medicine*, 2015.

[MKS$^+$17] Peter Mindek, David Kouřil, Johannes Sorger, David Toloudis, Blair Lyons, Graham Johnson, Meister Eduard Gröller, and Ivan Viola. Visualization multi-pipeline for communicating biology. *IEEE Transactions on Visualization and Computer Graphics*, 24(1), 2017.

[nsi] NVIDIA Nsight Graphics | NVIDIA Developer. `https://developer.nvidia.com/nsight-graphics`. [accessed 10-September-2018].

[ren] RenderDoc. `https://renderdoc.org/`. [accessed 10-September-2018].

[RSR13] Daniel Rakos, Graham Sellers, and Christophe Riccio. ARB_query_buffer_object. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_query_buffer_object.txt`, 2013. [accessed 16-September-2018].

[SSHL$^+$15] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. Flexible software profiling of gpu architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 185–197. ACM, 2015.

[ZO11] Yao Zhang and John D. Owens. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 382–393, 2011.