UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

The Possibilities of Compute Shaders - an Analysis

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Jochen Hunz

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Anna Katharina Hebborn, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im November 2013

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Jochen Hunz
(Matr.-Nr. 210 200 176)

Thema: **Untersuchung der Möglichkeiten von Compute Shadern**

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt wie im Bereich der GPU. Während anfangs nur Vertex- und Fragment-Programme möglich waren, so haben sich inzwischen, nach einem unified Shader-Modell, andere Programmiermöglichkeiten ergeben. So sind mit OpenGL 3.2 die Geometry Shader und mit OpenGL 4.0 die Möglichkeiten zur Tesselierung hinzugekommen. Nachdem unter DirectX 11 bereits 2009 die neuen Compute Shader eingeführt wurden, gibt es seit August 2012 diese auch unter OpenGL 4.3.

Ziel dieser Arbeit ist es, die Funktionsweise und Programmiermöglichkeiten von Compute Shadern detailliert zu analysieren. Darauf aufbauend soll ein (oder mehrere) Beispiel(e) ausgesucht und praktisch umgesetzt werden. Das Ergebnis ist schließlich eine Einschätzung, wie Compute Shader eingesetzt und programmiert werden, bzw. wo deren Möglichkeiten und Grenzen liegen.

Schwerpunkte dieser Arbeit sind:

1. Erlernen von modernem OpenGL
2. Einarbeitung in Compute Shader
3. Auswahl eines oder mehrerer Beispiele
4. Implementierung
5. Bewertung
6. Dokumentation der Ergebnisse

Koblenz, den 3.06.2013

[illegible] [illegible]

- Jochen Hunz - - Prof. Dr. Stefan Müller-

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Abstract

The following thesis analyses the functionality and programming capabilities of compute shaders. For this purpose, chapter 2 gives an introduction to compute shaders by showing how they work and how they can be programmed. In addition, the interaction of compute shaders and OpenGL 4.3 is shown through two introductory examples. Chapter 3 describes an N-Body simulation that has been implemented in order to show the computational power of compute shaders and the use of shared memory. Then it is shown in chapter 4 how compute shaders can be used for physical simulations and where problems may arise. In chapter 5 a specially conceived and implemented algorithm for detecting lines in images is described and then compared with the Hough transform. Lastly, a final conclusion is drawn in chapter 6.

Zusammenfassung

Die folgende Arbeit analysiert die Funktionsweise und Programmiermöglichkeiten von Compute Shadern. Dafür wird zunächst in Kapitel 2 eine Einführung in Compute Shader gegeben, in der gezeigt wird, wie diese funktionieren und wie sie programmiert werden können. Zusätzlich wird das Zusammenspiel von Compute Shadern und OpenGL 4.3 anhand zweier einführender Beispiele gezeigt. Kapitel 3 beschreibt dann eine N-Körper Simulation, welche implementiert wurde um die Rechenleistung von Compute Shadern und den Einsatz von gemeinsamen Speicher zu zeigen. Danach wird in Kapitel 4 gezeigt, inwiefern sich Compute Shader für physikalische Simulationen eignen und wo Probleme auftauchen können. In Kapitel 5 wird ein eigens konzipierter und entwickelter Algorithmus zur Erkennung von Linien in Bildern beschrieben und anschließend mit der Hough Transformation verglichen. Zuletzt wird in Kapitel 6 ein abschließendes Fazit gezogen.

Contents

1 Introduction

Graphics processing units (GPUs) are many-core processors with a high data and computation throughput. Former GPUs were designed and optimized for computer graphics. That means it was a challenge to use a GPU for general purposes. Todays GPUs are general-purpose parallel processors with support for various programming interfaces such as Nvidia's CUDA, Khronos Group's OpenCL, Microsoft's C++ Accelerated Massive Parallelism (C++ AMP) or DirectCompute with DirectX 11. On August 6th, 2012 the Khronos Group announced and immediately released the OpenGL[1] 4.3 specification which brings, among other things, the OpenGL adaption of DirectCompute: The OpenGL compute shaders. [26]. The compute shader stage is separated from the graphics pipeline and shares many of the same data types with the graphics stages (cf. appendix A.1). Like the other programmable shaders, compute shaders are written in the OpenGL Shading Language (GLSL), allowing GLSL developers to start using the graphics hardware for general-purpose computation on graphics processing units (GPGPU) without the need to learn one of the additional APIs mentioned before. Compute shaders provide high-speed general-purpose computing due to the large number of parallel processors on modern graphics hardware [15]. Further, they provide memory sharing and thread synchronization techniques and they can be dispatched completely independently from the rest of the OpenGL pipeline. All in all, compute shaders can be used for several applications like particle physics, fluid behavior, crowd simulation, ray tracing, global illumination [33] or for image processing.

This thesis presents an analysis of the functionality of compute shaders and their possibilities and limitations. First, chapter 2 shows how compute shaders work and how they can be programmed. Second, chapter 3 provides an implementation of an *N-Body* simulation which demonstrates the performance of compute shaders. Next, chapter 4 compares the usage of compute shaders and the CPU for physical simulations, especially fabric simulations. After this, chapter 5 shows the use of compute shaders for image processing purposes by detecting lines in images. Lastly, a final conclusion is drawn in chapter 6.

[1]Abbreviation for Open Graphics Library.

2 OpenGL Compute Shaders

This chapter gives a detailed overview how compute shaders work and how they can be programmed. For this, the modern GPU architecture is outlined and the compute shader specific OpenGL and GLSL language features are described. Furthermore, the usage of supportive OpenGL features with compute shaders are shown through two introductory examples.

2.1 Modern GPU Architecture

The *Fermi*™ architecture [18] continues and improves the idea of an unified shader model of the *G80* architecture introduced with the *GeForce 8800* in 2006. The basis for all GPU computing with Fermi and following generations of GPU architectures is *CUDA*. CUDA is the hardware (and software) architecture enabling the execution of programs on the GPU written in various languages. The language of interest for this thesis is the OpenGL Shading Language (GLSL). By using GLSL, several shader types can be realized and executed on the GPU using the CUDA architecture: vertex, fragment, geometry, tessellation, and compute shaders, which are considered in more detail through the chapters of this thesis. A compute shader is compiled and linked to a compute program. Such a compute program gets executed by a set of parallel *threads* on the GPU. These threads are organized in *local work groups* and grids of local work groups (also called the *global work group*), whereby every thread is executed concurrently. These threads within a work group can communicate among themselves through *barrier synchronizations* and a *shared memory* L1 cache. Every local work group has its own shared memory with a size of 64 KB. The execution of the threads is done by the *CUDA cores*. A CUDA core features a fully pipelined integer arithmetic logic unit and a floating point unit. Furthermore, a GPU, based on the Fermi architecture, is constructed from up to 16 *streaming multiprocessors (SM)*, containing 32 CUDA cores each. See figure 1 for an illustration of an SM. Altogether, a *GeForce GTX 580* consists of 16 SMs having 32 CUDA cores each, resulting in 512 CUDA cores in total. One SM executes one or more work groups, whereby threads are executed by the CUDA cores. Thereby, the threads are arranged in groups of 32 threads, called a *warp*. Each warp is executed in *lock-step* [21]. The *warp scheduler* units of the SM allow two warps to be executed concurrently. Regarding figure 1: A SM has also access to 16 *load and store units (LD/ST)* whose task is to calculate source and destination addresses within the cache or the *dynamic random access memory*. Furthermore, a SM has access to four *special function units (SFU)*. Each SFU executes transcendental instructions such as sine, cosine, reciprocal, and square root. See appendix A.2 for an illustration of the Fermi architecture.

The next generation of CUDA GPUs is based on the *Kepler*™ [19] compute architecture introduced in 2012. The new key feature of the Kepler generation is the new streaming multiprocessor architecture called *SMX*. The newest Kepler architecture is the *Kepler GK110*. A GK110 is constructed from up to 15 SMXs. Each SMX consists of 192 CUDA cores, 64 *double-precision units*, 32 SFUs and 32 load and store units. The size of the shared memory is also 64 KB. In contrast to the Fermi architecture, the new architecture provides four warp schedulers, therefore allowing four warps to run concurrently, whereby one warp is still formed out of 32 threads. See appendix A.3 for an illustration of the Kepler architecture and A.4 for an illustration of one SMX.

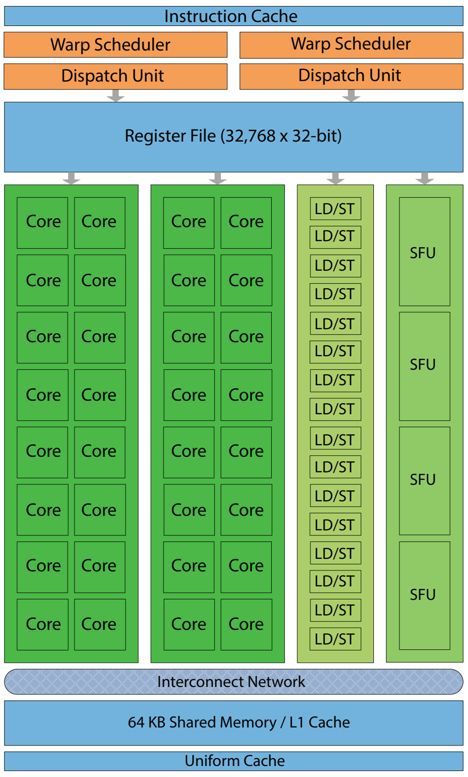


Figure 1: Fermi streaming multiprocessor (SM). A SM consists of 32 CUDA cores, 16 load and store units (LD/ST) and four special function units (SFU). Threads within a local work group ,executed by one SM, can cooperate among themselves through barrier synchronizations and a shared memory L1 64 KB cache. [18, p. 8]

2.2 Compute Shader Usage

This section provides an introduction to the use of compute shaders in OpenGL programs. It will show how to create, write and dispatch a compute shader using new language features. The dispatch of a compute shader can be fully configured to run on the GPU as effectively as possible. This means, the compute shader gets dispatched as one global work group. The global work group is a three dimensional space of local work groups. Each local work goup itself forms a three dimensional space of *threads* and gets executed on one streaming multiprocessor of the GPU as described in section 2.1. How to define the compute shader's dispatch is described in detail in section 2.2.2.

2.2.1 Creation

Creating a compute shader is similar to the established workflow of creating other shader types. Therefore

GLuint **glCreateShader**(GL_COMPUTE_SHADER)

creates a shader object, *glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)* loads the shader code and *glCompileShader(GLuint shader)* compiles the shader.
In contrast to other shader programs, a compute program can only hold compute shaders and can not be mixed with other types. To create a compute program, *glCreateProgram()* is called. Afterwards the shader object is attached by *glAttachShader(GLuint program, GLuint shader)* and linked by *glLinkProgram(GLuint program)* which makes the shader object discardable using *glDeleteShader(GLuint shader).*

Listing 1 at section 2.6 provides example source code.

2.2.2 Dispatch

To dispatch a compute shader, the compute program gets bound using *glUseProgram(GLuint program)* at first, whereby subsequent commands get applied to this program. Thereafter the shader is dispatchable at any point in the program using

void **glDispatchCompute**(GLuint num_groups_x,
GLuint num_groups_y,
GLuint num_groups_z).

The three parameters *num_groups_x*, *num_groups_y* and *num_groups_z* define the three dimensional space of the global work group as seen in figure

2. Consequently a call of *glDispatchCompute(6,4,4)* dispatches a three dimensional global work group having $6 * 4 * 4 = 96$ local work groups. Another way to define the space of the global work group is using

void **glDispatchComputeIndirect**(GLintptr indirect),

where the parameter *indirect* is the byte-offset to the buffer currently bound to the *GL_DISPATCH_INDIRECT_BUFFER* target. This buffer object must contain a set of parameters which could be passed to *glDispatchCompute()*, otherwise unintended things can happen, for example program termination.
The maximum number of dispatchable local work groups for the x,y and z dimensions can be queried calling

```
void glGetIntergeri_v(GLenum pname,
                      GLuint index,
                      GLint* params )
```

with *pname* set to *GL_MAX_COMPUTE_WORK_GROUP_COUNT*. The minimum value for each dimension is 65535. [43]

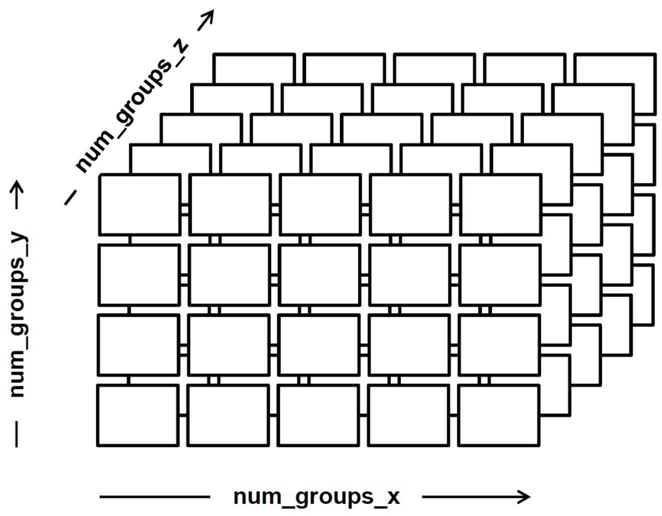


Figure 2: Global work group split into local work groups in three dimensions. [3] This global work group could be split using *glDispatchCompute(6,4,4).*

Using *glDispatchCompute()* or *glDispatchComputeIndirect()* requires the fixed definition of the local work group size in x,y and z dimension using an input layout qualifier in the shader code (see 2.2.3). OpenGL 4.4

introduced the extension *ARB_compute_variable_group_size* [10] which offers the ability to write generic compute shaders that operate on arbitrarily dimensioned local work groups. To dispatch such a compute shader, the extension implements the function

```
void glDispatchComputeGroupSizeARB(GLuint num_groups_x,
                                   GLuint num_groups_y,
                                   GLuint num_groups_z,
                                   GLuint group_size_x,
                                   GLuint group_size_y,
                                   GLuint group_size_z ),
```

where *group_size_x*, *group_size_y* and *group_size_z* are define the local work group size in x, y and z dimensions (cf. figure 4).

According to section 2.1 and to [18, 19, 38], a few heuristics for the composition of the global work group can be stated:

- The size of the local work groups should be a multiple of the warp size, which is 32 for the Fermi™and Kepler™architecture.
- The number of local work groups should be at least the number of available *streaming multiprocessors*, which depends on the GPU itself. That is because one local work group is executed on one streaming multiprocessor. A number of local work groups less than the number of streaming multiprocessors would lead to idle streaming multiprocessors.

2.2.3 Inputs

The only necessary input of a compute shader is the definition of its local work group size, using a special layout input declaration:

```
layout(local_size_x = X, local_size_y = Y, local_size_z = Z) in;
```

X, Y and Z are the local sizes for the specific dimension. Their default value is 1, thus

```
layout(local_size_x = 4, local_size_y = 3) in;
```

describes a two dimensional local work group having $4 * 3 * 1 = 12$ threads (cf. figure 3). The defined local work group size is determinable by calling

```
void glGetProgramiv(GLuint program,
                    GLenum pname,
                    GLint *params )
```

with *pname* set to *GL_COMPUTE_WORKGROUP_SIZE* and **params* is filled with three integers giving the size of the work groups. Using the OpenGL 4.4. extension *ARB_compute_variable_group_size* (see 2.2.2) allows the dispatch of a compute shader that operates on arbitrarily dimensioned local work groups. A compute shader using this extension gets its local size assigned by the program and must hold its own layout input declaration as:

```
layout (local_size_variable) in;
```

The local work group size is also limited and can be queried calling *glGetIntegeri_v()* with *pname* set to *GL_MAX_COMPUTE_WORK_GROUP_SIZE* (cf. 2.2.2). The minimum values are 1024 for the x and y dimension, and 64 for the z dimension. Furthermore, the product of the x, y and z dimensions must be less than *GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS*, whose minimum value is 1024. [43]

GLSL offers a set of built-in variables (also cf. figure 3 & 4). Thereby the *uvec3 gl_WorkGroupID* represents the three dimensional index of the local work group within the global work group and *uvec3 gl_NumWorkGroups* stores the total number of work groups. Therefore it always holds:

$$0 \leq gl\_WorkGroupID \leq gl\_NumWorkGroups - 1$$

The built-in variable *gl_NumWorkGroups* is a compute shader constant, thus its use requires a fixed local group size. With the OpenGL 4.4 extension *ARB_compute_variable_group_size* the possibility of using variable local group sizes was introduced. For this the new built-in variable *gl_LocalGroupSizeARB* must be used instead of *gl_NumWorkGroups* to prevent a compile-time error. The *uvec3 gl_LocalInvocationID* serves as the index of the shader invocation within the work group. Furthermore, *uvec3 gl_WorkGroupSize* stores the size of the local work group, and consequently the total amount of invocations or threads within one work group. The range of *uvec3 gl_LocalInvocationID* can thus be limited to:

$$0 \leq gl\_LocalInvocationID \leq gl\_WorkGroupSize - 1$$

A global index of the invocation can be determined using the built-in *uvec3 gl_GlobalInvocationID* which is computed by:

$$\begin{aligned} gl\_GlobalInvocationID = gl\_WorkGroupID * gl\_WorkGroupSize + \\ gl\_LocalInvocationID \end{aligned}$$

The last built-in type *uint gl_LocalInvocationIndex* is a one dimensional representation of the *gl_LocalInvocationID*. This unsigned integer can be used to access group shared memory (see section 2.3). It is computed as follows:

$$\begin{aligned} &gl\_LocalInvocationIndex = \\ &\quad gl\_LocalInvocationID.z * gl\_WorkGroupSize.y * gl\_WorkGroupSize.x + \\ &\quad gl\_LocalInvocationID.y * gl\_WorkGroupSize.x + \\ &\quad gl\_LocalInvocationID.x \end{aligned}$$

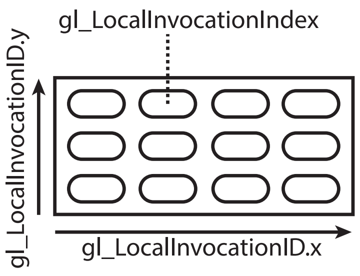


Figure 3: Two dimensional local work group, annotated with GLSL built-in variables. Each ellipse represents one single thread. This work group could be defined by using a layout input declaration as:
layout(local_size_x = 4, local_size_y = 3) in

2.2.4 Outputs

Compute shaders do not have any built-in outputs, nor any user-definable outputs which are automatically passed from one shader to another. Nevertheless a compute shader can write information back to buffers, texture images or atomic counters. For a more detailed consideration of using these features in compute shaders, consider the descriptions provided in section 2.5.

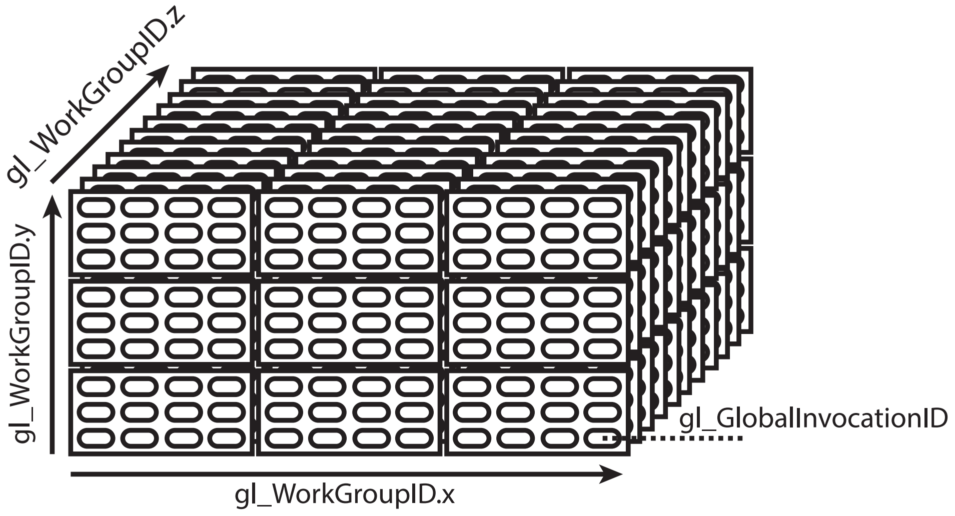


Figure 4: Three dimensional global work group having a two dimensional local work group size, annotated with GLSL built-in variables. This global work group could be dispatched using *glDispatchComputeGroupSizeARB(3,3,11,4,3,1).*

2.3 Group Shared Memory

Group shared memory is a L1 cache, which is available for every local work group. Compute shader invocations within one local work group can communicate with each other using one or more shared memory variables. To declare a shared variable the *shared* storage qualifier is used. Access to a shared variable is generally much faster than access to images or storage blocks [47, p. 445]. Accessing shared memory instead of global memory is about a hundred times faster [38]. Therefore, they provide a good platform for shader code optimization, especially when multiple invocations of the shader access the same data. Shared variables are coherent, thus writes to shared variables from one invocation will eventually be seen by other invocations within the same local work group. They may not have initializers and there is no defined order of execution with regards to reads and writes. To achieve such an ordering, memory barriers must be employed (see section 2.4). The total storage size for all shared variables in a compute shader is limited and can be queried using *glGetIntegerv()* with pname set to *GL_MAX_COMPUTE_SHARED_MEMORY_SIZE*. The value obtained is in bytes and its minimum size is 32 KB= $32 * 10^3$ B. [43]

2.3.1 Atomic Memory Functions

Atomic memory functions perform atomic operations on a signed or unsigned integer which is stored in a buffer or shared variable. Every operation reads a value from memory, computes a new value, writes it back to memory and returns the original read value. Thereby the function guarantees no access of any other shader invocation between the time the original value is read and the time the new value is written. A list of available atomic memory functions can be found in appendix A.5.

2.4 Synchronization

As described in section 2.1, one local work group is divided into a number of smaller chunks, while all invocations within one chunk are executed in lockstep. The time-sliced chunks may be assigned to the graphics processor's computational resources in any order. Therefore, a chunk of invocations could be completed before any more chunks from the same local work group begin. If, for example, invocations within a local work group want to communicate with each other using shared variables or a shader must ensure a particular order of execution, the code must perform synchronizations.
The function *barrier()* executes a flow control barrier. That means, one invocation will be blocked until all other invocations have reached this barrier. In addition, the *memoryBarrier()* function orders memory reads and writes for all kinds of variables. In contrast to this, the function *groupMemoryBarrier()* provides this barrier for the current work group only. In addition, *memoryBarrierShared()* controls the ordering of memory transactions to shared variables.
Beyond these ways of synchronization, OpenGL offers additional barriers for specific types of variables. The functions *memoryBarrierImage()*, *memoryBarrierBuffer()* and *memoryBarrierAtomicCounter()* provide barriers for their assigned features as described in section 2.5.

2.5 Supportive OpenGL Features

OpenGL offers several features to support compute shaders. This section provides an overview over the supportive features and extensions of OpenGL 4.

2.5.1 Image Load Store

This extension offers the ability to shaders to read from and write to a single level of a texture object from any shader stage. Determining the size of an image can be achieved using the GLSL built-in function ivec **imageSize**(gimage image)

where the dimensions of the returned type *ivec* are depending on the image type.

To load from a texture object, the extension provides the function

gvec4 **imageLoad**(gimage image, image_coord)

where the returning value *gvec4* is the data from the image depending on which format is specified using a *format layout qualifier.* The type of the parameter *image* depends on the declared image type. The second parameter *image_coord* is the image texel coordinate which is **n** dimensional according to the dimensions of the image type. Accessing a texel outside the boundaries of the image will return zero.

The counterpart of *imageLoad* is the built-in GLSL function

void **imageStore**(gimage image, image_coord, gvec4 data)

where *data* is the data written to the image at the given coordinates *image_coord*. Any store operation outside the boundaries of the image will be ignored. The format of *data* is based on its defined *format* given to *glBindImageTexture*.

void **glBindImageTexture**(GLuint unit, GLuint texture,
GLint level, GLboolean layered,
GLint layer, GLenum access,
GLenum format)

binds an image from *texture* using the given image *unit*. The *access* parameter restricts how the shader accesses the image. The value of *access* can be *GL_READ_ONLY*, *GL_WRITE_ONLY* and *GL_READ_WRITE*.

Please consider the OpenGL *ARB_shader_image_load_store* specification in [7] for an overview of the image types, format qualifiers or the memory qualifiers available. Section 2.6.1 provides a usage example of image load store and OpenGL compute shaders.

2.5.2 Shader Storage Buffer Object

A shader can perform random access reads, writes and atomic memory functions to data stored in a *shader storage buffer object* (SSBO). In contrast to *uniform buffer objects* (UBO), *SSBOs* can be larger than *UBOs*. Consequently, a *UBO* has to be at least 16KB in size, whereas the minimum size of a *SSBO* has to be at least 16MB. To determine the maximum size of a SSBO, *MAX_SHADER_STORAGE_BLOCK_SIZE* is queried with *GetInteger64v()*. The storage of a *SSBO* is unbounded, this means it can have an array of arbitrary length. As opposed to this, an *UBO* must have a specific and fixed storage size. The size of an array used with a *SSBO* can be queried using its *length* function. The workflow of creating, binding and mapping the buffer with data is similar to UBOs.

Furthermore, the extension offers the new packing layout qualifier *std430* for SSBOs, which provides a tighter packing of arrays and structures. With the existing *std140* qualifier, a multiple of a $16B$ of memory will be allocated for every array or structure of scalars and vectors. So every element in an array of *float*, *int* or *uint* will take up the size of a *vec4* ($16B$) instead of $4B$. In contrast to that, such elements will take up $4B$ using the *std430* qualifier. An exception is an array of *vec3*, which still requires $16B$.

Section 2.6.2 provides a usage example of SSBOs and OpenGL compute shaders.

2.5.3 Atomic Counter

Atomic counters [31] are GLSL unsigned integers which can only be manipulated using built-in atomic memory operations. The storage of an atomic counter comes from a buffer object, therefore they are created, mapped and bound, just like other buffer objects, by using *GL_ATOMIC_COUNTER_BUFFER* as the *target*. As a buffer object will store atomic counters, the buffer binding index and the offset within the buffer object are specified by the *binding* and *offset* layout qualifiers. Therefore

```
layout (binding = 2, offset = 16) uniform atomic_uint counter;
```

declares an atomic counter *counter*, bounded to binding point 2 and placed at offset 16 within the buffer object.

uint **atomicCounter**(atomic_uint c)

performs an atomic read of the specific atomic counter *c*, which must be of type *atomic_uint*.

uint **atomicCounterIncrement**(atomic_uint c) adds one to the value of the atomic counter *c*, atomicly. The functions returns the original value of *c*. The analog counterpart is

uint **atomicCounterDecrement**(atomic_uint c)

which performs an atomic subtraction of one to the value of *c*. This function also returns the original value of *c*.

2.6 Introductory Examples

This section provides two introductory examples of the use of compute shaders. For this, many code snippets are shown, which prevent a step by step guide.

Listing 1 shows the workflow to create a dispatchable compute program as decribed in section 2.2.1:

Listing 1: Creation of a compute program

```
// Create a shader object of type GL_COMPUTE_SHADER.
GLuint shader = glCreateShader(GL_COMPUTE_SHADER);

// Set the source of the shader, where shader_source of type GLchar*
// holds the source code.
glShaderSource(shader, 1, &shader_source, 0);

// Compile the shader code.
glCompileShader(shader);

// Create a shader program.
GLuint program = glCreateProgram();

// Attach the compiled shader to the shader program.
glAttachShader(program, shader);

// Link the program. A compute program can only hold compute shaders!
glLinkProgram(program);

// Delete the shader, it is no longer needed.
glDeleteShader(shader);
```

2.6.1 Inverting an Image

This example is about inverting an image, which shows the usage of *image load store* (see 2.5.1) in combination with compute shaders. The compute shader will get two images as an input. The texture *input_texture* is the original texture and *output_texture* stores the inverted values of the original texture. Listing 2 creates the empty OpenGL texture object *output_texture*, which is bound to the specified *image unit* 1:

Listing 2: Creation of an empty texture object and binding it to the *image unit* 1

```
 1  // Unsigned integer which will refer to the texture object.
 2  GLuint output_texture;
 3
 4  // Generate one texture object.
 5  glGenTextures(1,&output_texture);
 6
 7  // Bind the generated texture object to the GL_TEXTURE_2D target.
 8  glBindTexture(GL_TEXTURE_2D, output_texture);
 9
10  // Specify storage for one, two-dimensional texture level.
11  glTexStorage2D( GL_TEXTURE_2D,    // Target
12                  1,                // Amount of texture levels
13                  GL_RGBA32F,       // Internal format of data
14                  texture_width,    // Width of the texture
15                  texture_height    // Height of the texture
16                );
17
18  // Bind the texture to the image unit 1.
19  glBindImageTexture( 1,               // Image unit
20                      texture,         // Texture to bind
21                      0,               // Level of the texture
22                      GL_FALSE,        // Is layered?
23                      0,               // Layer
24                      GL_WRITE_ONLY,   // Access restrictions
25                      GL_RGBA32F       // Format of data
26                    );
```

Line 11 of listing 2 uses

void **glTexStorage2D**(GLenum target, GLsizei levels,
GLenum internalformat,
Glsizei width, GLsizei height)

which specifies storage for a two-dimensional texture. This is done simultaneously for all *levels* of the texture. The *internalformat* specifies the size of the stored data, which must be defined in the compute shader as well.

Line 19 binds the texture to the *image unit* 1, as described in section 2.5.1. The *format* is the same as used in *glTexStorage2D()* in line 11 and the image will be *GL_WRITE_ONLY*, as the compute shader will only write the inverted data to it. In contrast to this, the input texture will be loaded from a file and generated using *glTexImage2D()*. Afterwards it will be bound to the *image unit* 0 and specified as *GL_READ_ONLY*, as the compute shader will not write to it.

Listing 3: Compute shader source code to invert an image using the image load store extension (cf. 2.5.1)

```
 1  #version 440
 2
 3  // Specify the local work group size using OpenGL 4.4.
 4  layout (local_size_variable) in;
 5
 6  // Or specify the local work group size using OpenGL 4.3.
 7  // layout (local_size_x = 32, local_size_y = 32) in;
 8
 9  // The images bound to the specific image unit of
10  // type image2D stored in the rgba32f format.
11  layout (binding = 0, rgba32f) readonly  uniform image2D input_image;
12  layout (binding = 1)          writeonly uniform image2D output_image;
13
14  void main()
15  {
16    // Use the x and y index of the global invocation as the index
17    // to read from or write to the images.
18    // Every invocation works at a unique texel position of the image.
19    ivec2 index = gl_GlobalInvocationID.xy;
20
21    // Use image load store to load from the input image
22    // at the specific index.
23    // Therefore, every invocation reads a unique texel of the image.
24    vec4 texel_color = imageLoad(input_image, index);
25
26    // Invert the read texel color
27    vec4 result_color = vec4(1.0 - texel_color.rgb, texel_color.a);
28
29    // Store the inverted color in the output image using image load store.
30    imageStore(output_image, index, result_color);
31  }
```

Listing 3 provides the source code of a compute shader usable to invert an image. Line 4 specifies the local work group size of the shader using the OpenGL 4.4 feature *ARB_compute_variable_group_size* as described in section 2.2.2. The OpenGL 4.3 method is shown in line 7 in which a two dimensional local work group size is defined. Line 11 and 12 declare the input and output images, which are of type *image2D*. They are bound to the *image unit* specified by *glBindImageTexture* in line 16 of listing 2. The *readonly* and *writeonly* qualifiers are not necessary but by using them, access violations become determinable at compile-time. An access violation at run-time leads to undefined behavior, including program termination. If an image is declared as *writeonly*, the *format qualifier* is not necessary because it only specifies the format for read operations. [7]

To dispatch the compute shader of listing 3, the compute program is set to active at line 2 in listing 4 first. Afterwards the shader is dispatched at line 5 using the the OpenGL 4.4 feature *ARB_compute_variable_group_size*. The command *glDispatchComputeGroupSizeARB()* dispatches a global work group split in two-dimensions. Each of the resulting local work groups will be two-dimensional having $32 * 32 * 1 = 1024$ threads, which is the defined minimum value of *GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS* as described in section 2.2.3. The global work group consists of $(texture\_width\ /32) * (texture\_height/32) * 1$ local work groups. If the image to invert is $1920 * 1080$ pixels in size, for instance, the global work group will consist of $(1920/32) * (1080/32) * 1 = 2025$ local work groups. Since one local work group consists of 1024 threads, $2025 * 1024 = 2073600$ threads will be executed in total. This means one thread per pixel is executed as $1920 * 1080$ pixels $= 2073600$ pixels and therefore the whole image gets inverted. If OpenGL 4.4 is not available, the commented-out code line 16 shows the OpenGL 4.3 method for dispatching such an amount of local work groups. This line is in keeping with line 7 of listing 3. Finally, line 17 provides a memory barrier to ensure synchronization of the output image between the compute shader dispatch and the rendering of it.

Listing 4: Dispatching and synchronization of a compute shader which uses *image load store*

```
 1  // Activate the compute program.
 2  glUseProgram(program);
 3
 4  // Dispatch the compute shader using OpenGL 4.4.
 5  glDispatchComputeGroupSizeARB(
 6            texture_width  / 32,    // Global work group size X dimension
 7            texture_height / 32,    //                        Y dimension
 8            1,                      //                        Z dimension
 9
10            32,                     // Local work group size X dimension
11            32,                     //                       Y dimension
12            1                       //                       Z dimension
13  );
14
15  // Or dispatch the compute shader using OpenGL 4.3.
16  // glDispatchCompute(texture_width / 32,texture_height / 32,1);
17
18  // Do a memory barrier.
19  glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

2.6.2 Moving Particles

This example illustrates the usage of *SSBOs* (see section 2.5.2), in combination with compute shaders, by computing particle movements under the influence of the gravitational forces as seen in figure 5.

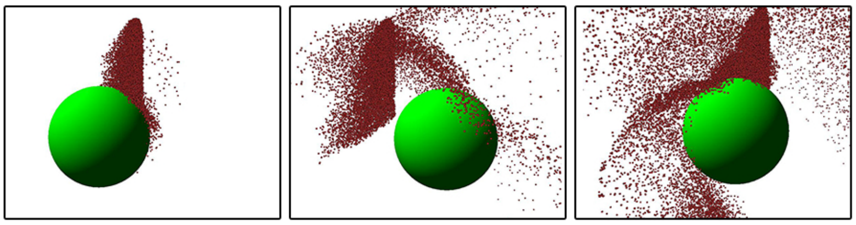


Figure 5: Particles influenced by gravitational forces, simulated at different timesteps, colliding with a sphere. Thereby, the sphere is moving from left to right and vice versa.

A particle p consists of a current position r, velocity v and mass m, whereby m is assumed to be always 1 in this example. Every r and v is stored in a *SSBO* to deliver the compute shader with data. Listing 5 illustrates the generation, initialization and binding to an *indexed buffer target* of the *SSBO* which stores the positions r. Thereby, line 12 allocates empty storage for every particle's position. One position r is stored as a *vec4* which take up $16B$ of memory storage. As m is negligible, because it is always 1 in this example, one might assume using *vec3* is more appropriate to store each r. Nevertheless it makes no difference whether *vec3* or *vec4* is used as they will always take $16B$ of storage (cf section 2.5.2). At line 19 et seq. the complete buffer is mapped to a pointer of *vec4*. The access flag *GL_MAP_WRITE_BIT* indicates that the returned pointer is used to write to the buffer data. Using the access flag *GL_MAP_INVALIDATE_BUFFER_BIT* indicates that the previous content of the entire buffer is discardable, which speeds up the mapping of the buffer. After filling the pointer with some data, line 34 *unmaps* the buffer and uploads the data to the GPU's video memory. Line 37 et seq. binds the *SSBO* to the buffer index 0.

Listing 5: Generation, initialization and binding of a *shader storage buffer object*

```
// Unsigned integer which will refer to the buffer object.
GLuint positionBuffer;

// Generate one buffer object.
glGenBuffers(1,&positionBuffer);

// Bind the generated buffer object to
// the GL_SHADER_STORAGE_BUFFER target.
glBindBuffer(GL_SHADER_STORAGE_BUFFER,positionBuffer);

// Specify empty storage for the buffer object
glBufferData( GL_SHADER_STORAGE_BUFFER,  // Target buffer type.
              maxParticles*sizeof(vec4), // Storage for all particles
              NULL,                      // Empty data
              GL_STATIC_DRAW             // Usage of the buffer
            );

// Map the Buffer to fill it with data
vec4* positions =
   (vec4*) glMapBufferRange( GL_SHADER_STORAGE_BUFFER,    // Target
                             0,                           // Offset
                             maxParticles*sizeof(vec4),   // Length
                             GL_MAP_WRITE_BIT |           // Access flags
                             GL_MAP_INVALIDATE_BUFFER_BIT
                           );

// Initialize the buffer object with data to store
for( GLint i = 0; i < maxParticles; i++ )
{
  positions[i] = vec4(0,0,0,1);
}

// Unmap the buffer and upload it to the GPU
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);

// Bind the buffer to the indexed buffer target 0
glBindBufferBase( GL_SHADER_STORAGE_BUFFER,  // Target buffer type
                  0,                         // Indexed buffer target
                  positionBuffer             // Buffer to bind
                );
```

The *SSBO* to store the velocities v for every particle p is created in a similar way. The only difference is that the velocity buffer is bound to the buffer index 1 and that the initial velocities are chosen in such a way that the particles are scattered.

Listing 6 shows the basic source code necessary to compute particle movements under the influence of G using a compute shader and *SSBOS*. Line 6 declare the buffer object for the positions using the *interface block* semantic of OpenGL 3.1. Its binding point was defined using the *glBindBufferBase()* command of listing 5.

Listing 6: Compute shader source code which interacts with *shader storage buffer objects.*

```
1  #version 440
2
3  // GLSL 3.1 interface blocks which defines the position and
4  // the velocity buffer as a shader storage block.
5  // Therefore, the buffer are interpreted as arrays of vec4.
6  layout( std140, binding=0 ) buffer Pos {
7      vec4 Positions[ ];
8  };
9
10 layout( std140, binding=1 ) buffer Vel {
11     vec4 Velocities[ ];
12 };
13
14 // Specify the local work group size using OpenGL 4.4.
15 layout (local_size_variable) in;
16
17 // Gravitational constant.
18 const vec3 G = vec3(0.0, -9.8, 0.0);
19
20 // Delta time
21 const float DT = 0.001;
22
23 void main(){
24     // Use the x index of the global invocation as the index
25     // to read from or write to the SSBOs. Every invocation works
26     // at a unique position of the buffers.
27     uint index = gl_GlobalInvocationID.x;
28
29     // Read the position and velocity from the SSBOs.
30     vec3 r = Positions[index].xyz;
31     vec3 v = Velocities[index].xyz;
32
33     // Compute acceleration
34     vec3 acceleration = computeAccleration(r,v,G);
35
36     // Integration using some numerical integration method
37     v = integrateVelocity(v,acceleration,DT);
38     r = integratePosition(r,v,DT);
39
40     // Store the integrated position and velocity.
41     Positions[index].xyz = r;
42     Velocities[index].xyz = v;
43 }
```

The data of the buffer is interpreted as an array of *vec4*, which is read- and writable through the shader. Line 10 declares the velocity buffer respectively. In line 27 the global invocations x-index of this dispatch is queried using the built-in variable *gl_GlobalInvocationID*. This index allows every single invocation to work on an unique particle inside the buffer objects. To access a buffer object, its declared array is used in line 30 or 31. Afterwards the shader calculates the acceleration acting on the particle and integrates it's position and velocity through a numerical integration method like the *Euler* or the fourth-order *Runge-Kutta* method [40, p. 490 et seq.]. Lastly, the shader writes the integrated position and velocity back to the buffers.

3 N-Body Simulation

An N-Body simulation represents an evolution of a system of bodies where each body interacts with every other body. For instance, each body could represent a single star in an astrophysical simulation which attracts each other star of the system through a gravitational force as in figure 6. Other scientific fields using N-Body simulations are molecular dynamics, plasma physics or fluid flow simulations [39]. This chapter discusses an N-Body simulation in an astrophysical context which demonstrates the computational power of OpenGL compute shaders and the effective usage of group shared memory. In order to demonstrate the capabilities of compute shaders, forces are computed using a brute-force technique, evaluating all interactions pair-wise among N bodies which leads to an $O(N^2)$ computational complexity. Other and faster approaches to solve the N-Body problem are the Barnes-Hut algorithm [4], the fast multipole method [25] or the parallel multipole tree algorithm [6]. A good overview is given by Blelloch and Narlikar. [5]

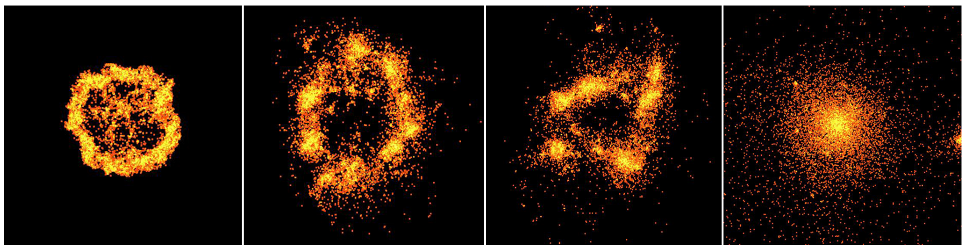


Figure 6: N-Body system with $N = 16.384, m = 1.0, \epsilon^2 = 0.01$ simulated at different timesteps running on a compute shader.

3.1 Problem Definition

Newton's law of universal gravitation calculates the force of gravity $\vec{f}$ between two masses m_1 and m_2, which are a distance r apart. This law can be written as follows: [40, p. 480]

$$\vec{f} = G * \frac{m_1 m_2}{r^2} \tag{1}$$

Based on equation 1, the force $\vec{f_{ij}}$ acting on particle i of mass m_i, caused by its gravitational attraction to body j of mass m_j, is given by the following formula (cf. [44, 39]):

$$\begin{aligned} \vec{f_{ij}} &= G\frac{m_i m_j}{|\vec{r_i} - \vec{r_j}|^2} * \frac{\vec{r_i} - \vec{r_j}}{|\vec{r_i} - \vec{r_j}|} - \nabla\phi_{ext}(\vec{r_i}) \\ &= G\frac{m_i m_j(\vec{r_i} - \vec{r_j})}{|\vec{r_i} - \vec{r_j}|^3} - \nabla\phi_{ext}(\vec{r_i}) \end{aligned} \tag{2}$$

G is the gravitational constant and ϕ_{ext} is the external potential. The left factor of the product is the magnitude of the force which is given by the product of the masses $m_i * m_j$ and reduced by the square of the distance $|\vec{r_i} - \vec{r_j}|^2$ of the body i and j (cf. equation 1). The right factor is the normalized direction of the force $\overrightarrow{f_{ij}}$.
The summation of all forces $\overrightarrow{f_{ij}}$ acting on body i, without taking external potentials into consideration results in the total force $\vec{F_i}$ on body i:

$$\begin{aligned}\vec{F_i} &= \sum_{i \neq j} \overrightarrow{f_{ij}} \\ &= \sum_{i \neq j} G \frac{m_i m_j * (\vec{r_i} - \vec{r_j})}{|\vec{r_i} - \vec{r_j}|^3} \\ &= G m_i \sum_{i \neq j} \frac{m_j * (\vec{r_i} - \vec{r_j})}{|\vec{r_i} - \vec{r_j}|^3}\end{aligned} \tag{3}$$

The mass m_i of body i and the gravitational force G are numerical constants during the equation and therefore m_i and G can be extracted from the summation. The force $\vec{F_i}$ grows without any limitations as bodies approach each other. This is especially critical when the distance of two bodies approaches zero and the fraction becomes undefined through a denominator of zero. A constant timestep numerical integration, like the *Euler* method, the *fourth-order Runge-Kutta* method [40, p. 490 et seq.] or the *Verlet* integration [45], can not guarantee enough accuracy when two bodies collides and therefore leads to physically unrealistic accelerations [44]. To solve this issue, [44] introduces a softening factor $\epsilon^2 > 0$, whereby equation 3 can be rewritten as:

$$\vec{F_i} = G m_i \sum \frac{m_j(\vec{r_i} - \vec{r_j})}{(|\vec{r_i} - \vec{r_j}|^2 + \epsilon^2)^{3/2}} \tag{4}$$

The condition $i \neq j$ of equation 3 is not necessary anymore, because the denominator is always greater than zero and thus well defined. This results in a force $f_{ii} = 0$ when $\epsilon^2 > 0$. To integrate equation 4 over time, *Newton's second law* yields the acceleration $\vec{a_i} = \vec{F_i}/m_i$ through:

$$\vec{a_i} = G * \sum \frac{m_j(\vec{r_i} - \vec{r_j})}{(|\vec{r_i} - \vec{r_j}|^2 + \epsilon^2)^{3/2}} \tag{5}$$

3.2 Implementation

The implementation, which underlies this thesis, uses an OpenGL shader storage buffer object (see 2.5.2) to store the N bodies and velocities of the system[2]. This buffer object allows the compute shader to read the positions

[2]In literature, like [24, p. 367], the Verlet integration is preferably used for particle simulations in contrast to the Euler integration method. Therefore, one could store the old body's position instead of storing its velocity.

and velocities of each body at time t and to write back the resulting positions and velocities at time $t + \Delta t$. Therefore, all positions r_i, followed by the velocity v_i, of each spherical, random ordered body i are mapped to the buffer and uploaded to the GPUs video ram. After the dispatch of the compute shader, and therefore after the computation of one simulation step, the shader storage buffer object needs to get synchronized using a *glMemoryBarrier* to ensure that every data is written. Afterwards it is used as an array buffer to fill the vertex shader with data to render the scene. Using this technique the data of the simulation does not have to be copied back to the system memory and thereby expensive copy operations between the system memory and the video memory are prevented.

3.2.1 Compute Shader

A first approach of a CPU implementation would compute the force F_i for each body i sequentially, resulting in an $O(n^2)$ computational complexity. Using a compute shader, therefore computing the forces for every body i on the GPU in parallel, enhances the performance of the program. The global work group dispatch of the shader is one dimensional and every work group has a local one dimensional size. The code ensures a dispatch of $N =$ *gl_NumWorkGroups.x* $*$ *gl_WorkGroupSize.x* invocations, where N is the total number of bodies in the simulation. As a consequence *gl_GlobalInvocationID.x* is suitable as the index of the shader storage buffer object. This guarantees that every shader invocation works on one unique body of the simulation and makes the body's position and velocity accessible. After the computation of the acceleration a_i, the shader integrates the new position and velocity of the body. Before writing back the results, the shader needs a synchronization using *barrier()* in order to prevent premature overriding of the buffers data. This avoids the interference in calculations of other shader invocations during one dispatch. Figure 6 shows the visualized bodies of a simulation with 16.384 bodies, where every body attracts each other body.

3.2.2 Using Group Shared Memory

The implementation described in the previous section 3.2.1 accesses the shader storage buffer object N^2 times for every body i. This section states an approach described by Lars Nyland et al. [39] which uses the group shared memory of the GPU in an effective way to reduce the accesses to the buffer object while reusing data. Therefore, the shader splits the position-data into N/p tiles, whereas p is the size of the dispatched work groups. These tiles are stored into the group shared memory of the GPU. Within each tile, the shader computes the acceleration a_i and proceeds with the next tile afterwards as shown in listing 7:

Listing 7: Compute shader using group shared memory to solve the n-body problem

```
 1 // Split the buffer data into $N / p$ tiles
 2 for(uint tile = 0; tile < N / p; tile++){
 3   // Load the intended position for this tile and this invocation
 4   // into the shared memory
 5   sharedMemory[invocationID] = Positions[tile * p + invocationID];
 6 
 7   // Synchronize the shared memory
 8   memoryBarrierShared();
 9 
10   // Compute the acceleration for the body i to every body j
11   // within the tile
12   for(uint j = 0; j < p; j++){
13     acceleration += interaction_i_to_j(i,sharedMemory[j]);
14   }
15 
16   // Synchronize the invocations
17   barrier();
18 }
```

Line 8 of listing 7 synchronizes the shared memory and the invocations, ensuring that the information for this tile is loaded completely. Line 13 computes the total acceleration which interacts between the body i and all bodies j within one tile using the previously filled shared memory. Afterwards *barrier()* guarantees that every invocation has done its computations to continue to the next tile. Without this last synchronization some data of the shared memory could get overridden before every invocation has used the data for its computation.

3.3 Evaluation

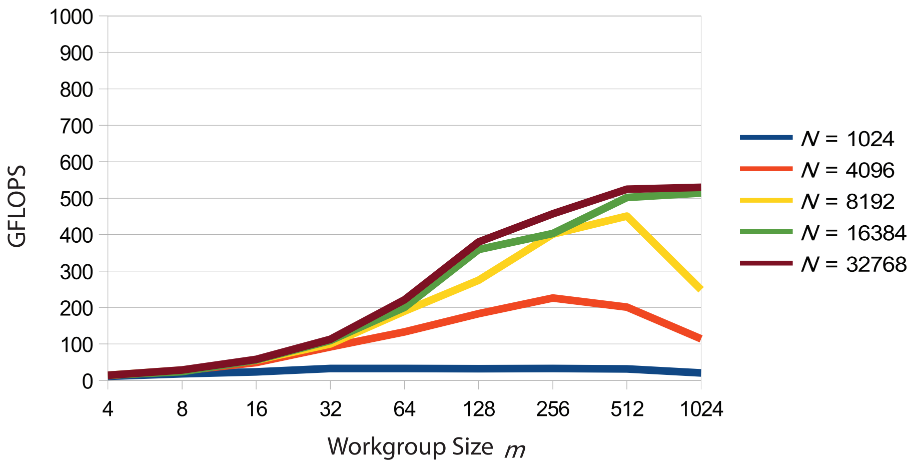
This section discusses the evaluation of the two implementations, which are described in the previous sections. For this, the computation time of the outlined compute shaders was measured using *OpenGL Timer Queries* [42, p. 45 et seq.], which gives the time a set of OpenGL commands needed, in nanoseconds. In addition, the efficiency of the compute shaders was measured by estimating the *giga floating point operations per second* (GFLOPS) the GPU performed. The approach to estimate the GFLOPS is based on the code of Nyland et al. [39], which assumes 20 FLOPS per shader invocation. Various configurations (N, m) were used. The amount of bodies N ranges from 1024 to 32768 and the size of the workgroups m ranges from 4 to 1024. This implies that a configuration $(16384, 64)$ has a total amount of $N/m = 16384/64 = 256$ workgroups. For evaluation purposes, a *GeForce GTX 580* graphics card by NVIDIA is used. The *GeForce GTX 580* is based on the *Fermi* architecture[3] running at *1544 MHz* and featuring *512 CUDA cores* [17]. These *512 CUDA cores* are arranged in *16 streaming multiprocessors*. Therefore, each *streaming multiprocessor* contains *32 CUDA cores* [16].

[3]compare section 2.1

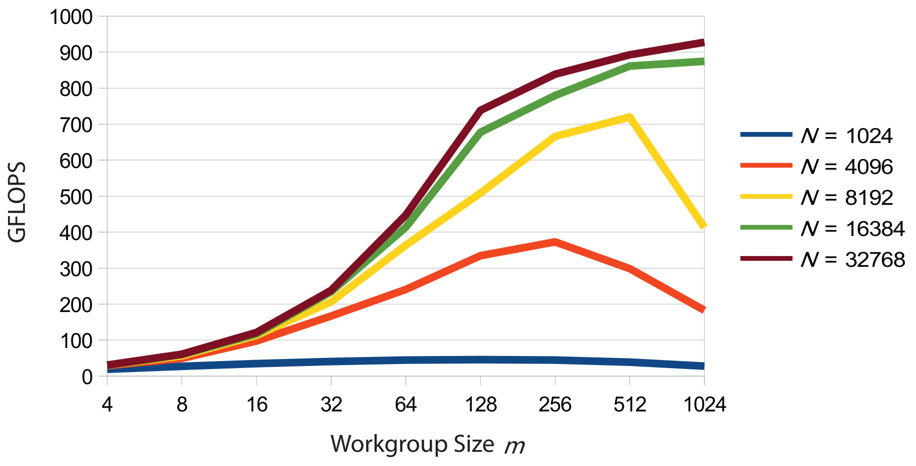
The maximum achievable floating point operations per second are 1581. [1]

Figure 7 exhibits performance graphs for the two implementations described in the previous section. Thereby the x-axis maps the size of the workgroup and the y-axis maps the GFLOPS performed by the *GeForce GTX 580*. Every curve represents a simulation with N bodies. Figure 7a presents the estimated GFLOPS for several numbers of N without the utilization of *shared memory* (*SM*) and figure 7b demonstrates the implementation which uses *SM*, respectively. In general it can be stated that the use of *SM* provides a large increase in the performed GFLOPS. Given a configuration $(N, m) = (32768, 1024)$, 529.68 GFLOPS were measured with no *SM* in use, taking an average of $40, 79$ ms per frame ($\approx$ 24 FPS[4]) . Using *SM* instead leads to 927.61 GFLOPS taking an average of 22.94 ms per frame ($\approx$ 43 FPS). This yields an increase of 175.12% GFLOPS when using *SM* in this configuration. Considering a configuration $(32768, 4)$, it can be measured that the use of *SM* increases the performed GFLOPS from 14.59 GFLOPS, taking an average of 1461.09 ms per frame, to 30.60 GFLOPS, taking an average of 701.248 ms per frame. This is a percentage increase in GFLOPS of 209.66%. In addition, the results demonstrate that the performance of a compute shader depends on the chosen compute shader dispatch and the GPU architecture. Using a workgroup size of 1024, instead of 4, is up to 36 times more efficient when simulating $N = 32768$ bodies. As said before, the *GeForce GTX 580* GPU consists of *512 CUDA cores* arranged in *16 streaming multiprocessors*. As a compute shader local workgroup runs on a single *streaming multiprocessor*, it leads to a performance decline if less than 16 workgroups are dispatched. This explains the declining number of GFLOPS performed by the GPU in the configurations $(8192, m > 512)$ and $(4096, m > 256)$. Here the dispatched workgroups are less than 16 which leads to *streaming multiprocessors* being idle and therefore a worse performance.

[4]FPS is abbreviation for *frames per second.*



(a) Without using shared memory.



(b) Using shared memory.

Figure 7: Estimated GFLOPS with (b) and without (a) the use of *shared memory* performed by a *GeForce GTX 580* by simulating the system with N bodies. The x-axis maps the size of the used workgroups, therefore the number of threads per workgroup. The y-axis maps the performed GFLOPS.

4 Fabric Simulation

This section covers the topic of fabrics behavior simulation under the assistance of compute shaders. For this, an overview of the structure of fabric and how fabric can be represented and simulated is given. With respect to the topic of this thesis, the main focus leis on the compute shaders, their dispatch, optimization and the advantages and disadvantages of their use in physical simulations, especially fabric simulations. Advanced aspects of fabric simulation, like self-collision detection or the photo-realistic rendering of fabric, are not treated.

4.1 Structure of Fabric

The behavior of fabric is based on the nature and molecular structure of the fiber material constructing the fabric and the arrangement of these fibers within the fabric [46, p. 15]. Volino and Magnenat-Thalmann differentiate between three ways how fabric fibers can be organized:

- "*Woven Fabrics:* Threads are orthogonally aligned and interlaced alternately using different patterns" [46, p. 15]. They are stiff and thin, therefore used for garments. (cf. figure 8)
- "*Knitted Fabrics:* Threads are curled along a given pattern, and the curls are interlaced on successive rows" [46, p. 15]. They are loose and elastic, thereby often used by wool and underwear. (cf. figure 9)
- "*Non-woven Fabrics:* There are no threads, and the fibers are arranged in an unstructured way, such as paper fibers." [46, p. 15]

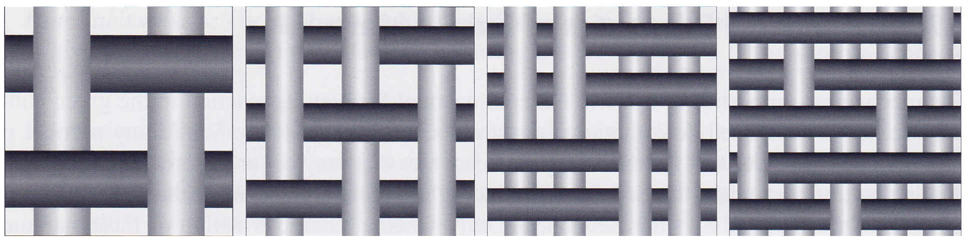


Figure 8: Woven fabric patterns: *Plain*, *Twirl*, *Basket*, *Satin*. [46, p. 16]

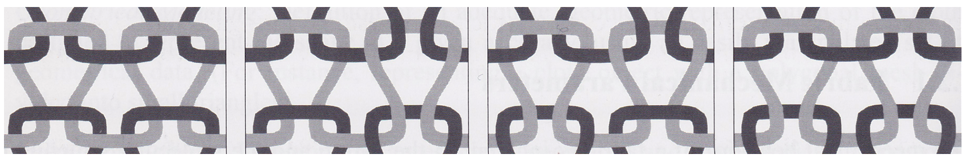


Figure 9: Knitted fabric patterns [46, p. 16]

4.2 Simulation

Woven fabrics can be simulated using a particle system which discretizes the fabric material as a set of mass points. These mass points interact with forces which are computed using a mass-spring system (see section 4.2.1). By this, the setting of the springs define the stiffness and behavior of the fabric. Figure 10 shows a simulated fabric, which collides with a sphere and a plane, at different time steps. The fabric consists out of 64×64 mass points, which are connected with 47.562 springs, and is simulated by a compute shader. Implementation specific details are described in the following sections.

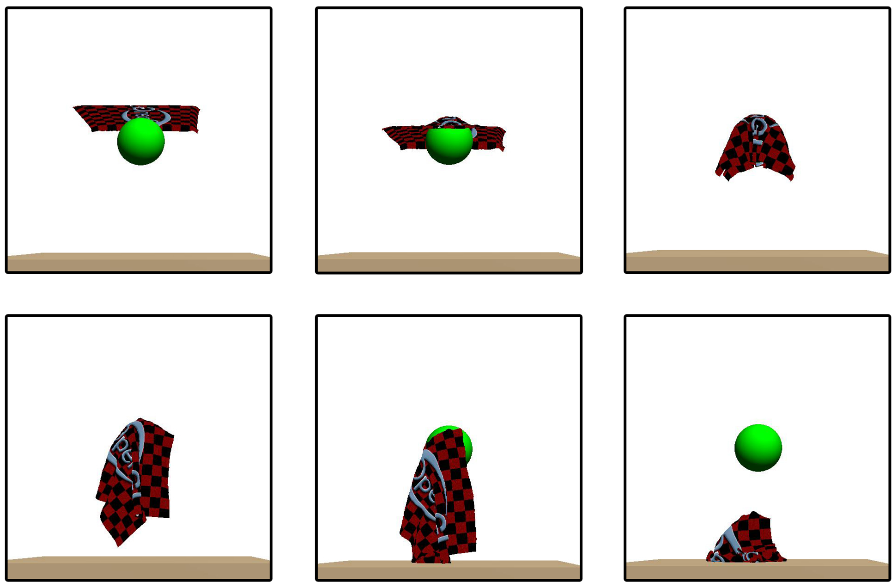


Figure 10: Fabric discretized through 64×64 mass points, which are connected with 47.562 springs. The spring constant c is 180.000 and a damping factor d of 8 is considered.

4.2.1 Mass-Spring Systems

Mass-spring systems are widely used to compute spring forces within a simulation of deformable bodies such as hair, cloth, water, or gelatin [23]. Therefore, a system of point masses is connected by springs. A spring is stretched between a fixed point and a free one. An unstretched spring, as shown in figure 11a, has a *resting length* L. If the end of the spring gets pulled away or pushed towards to the fixed one, the spring exerts a force $\vec{F}$. If the end of the spring is pulled away from the fixed point, the direction of the force $\vec{F}$ is toward its fixed point (cf. figure 11b). Should the end of the spring gets pushed towards its fixed point, the direction of the force $\vec{F}$ is toward its end (cf. figure 11c). The described law for spring forces is

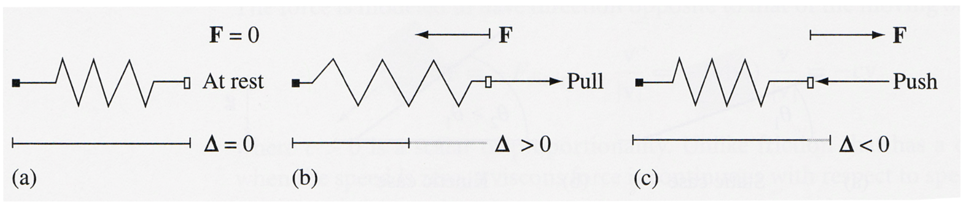


Figure 11: (a) Unstretched spring. (b) Force due to stretching the spring. (c) Force due to compressing the spring. [23, p. 35]

known as *Hooke's law*, which can be expressed as follows: (cf. [23, p. 34])

$$\vec{F} = -c\Delta\vec{U} \tag{6}$$

where the *spring constant* $c > 0$ is the constant of proportionality. The *spring constant* specifies the stiffness of a spring, this is, a large c yields a stiff spring and vice versa. $\vec{U}$ is a unit-length vector pointing in the direction of the spring and Δ is the amount by which the spring was displaced from its resting length L. This means, if the spring gets stretched, Δ is positive and any compression of the spring produces a negative value for Δ.

According to [47, p.270], all real spring systems have some loss due to friction. Adding a *damping* factor d can model this loss and also improves the simulation's stability:

$$\vec{F} = -c\Delta\vec{U} - d\vec{v} \tag{7}$$

where $\vec{v}$ is the current velocity of the free end of the spring. While $\vec{F}$ acts on the free end of the spring, the opposing force $-\vec{F}$ acts on the fixed point of the spring through *Newton's third law*. [37, p. 82]

4.2.2 Surface Representation

As described before, the fabric gets discretized through a set of mass points. The mass points are ordered in a uniform grid and are connected with springs as in figure 12. Thereby, each mass point is connected with springs to 12 mass points in its neighborhood, if possible. Three types of springs can be distinguished [29]. *Structural* springs (green in figure 12) uphold the basic structure of the set of mass points. *Shearing* springs (blue) are used to model the *shearing elasticity* [46, p. 53]. The shearing elasticity can be described using the *Kawabata Shearing Test* which involves the extension of a rectangle of cloth material at a constant speed of extension, whereby the movement is performed transversally [46, p. 20]. Similar to shearing springs, *bending* springs (red) are used to model the *bending elasticity* which can be described using the *Kawabata Bending Test*.

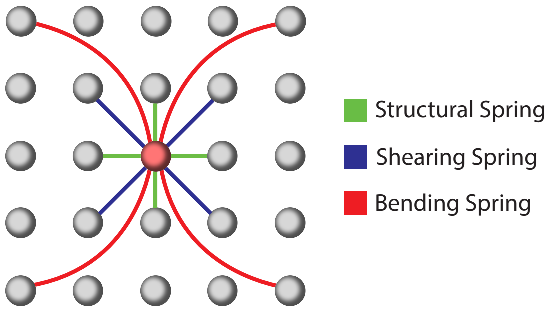


Figure 12: Set of mass points, ordered in a uniform grid to discretize a fabric. Each mass point is connected to 12 neighbors using *structural, shearing* and *bending* springs. (cf. [24, p. 366])

4.3 Compute Shader Implementation

The fabric gets discretized through a set of mass points. One mass point is represented by one four-dimensional vector by which the first three components store the position of one mass point and the fourth component its inverse mass. These four dimensional vectors are stored in an SSBO (cf. section 2.5.2). In addition, the mass points' velocities (for numerical integration purposes[5]) and the normals of the mass points' vertices (for lighting purposes) are also stored using SSBOs.

On basis of the initial positions of the mass points, the resting length for each type of spring can be computed. If the mass points are ordered in a uniform grid, a bending spring has a greater resting length than a shearing spring which again has a greater resting length than a structural spring. The three different resting lengths, the spring constant c and damping factor d are passed as uniform variables to the compute shader. The dispatch of the compute shader is chosen in a way that a compute shader invocation is running for every mass point i. The shader itself computes the total force $\overrightarrow{F}_i$ for mass point i by computing and summing up all forces $-\overrightarrow{F}_{ij}$ which are exerted by the springs the mass point i is connected to[6]. Thereby, the

[5]Instead of using the mass points' velocities for numerical integration, one could use and store the mass points' old positions to do Verlet integration.

[6]As described in section 4.2.1, $\overrightarrow{F}$ acts on the free end of the spring. The compute shader is dispatched for every mass point i, whereby i is considered as the fixed point of the spring. This means, the opposing force $-\overrightarrow{F}_{ij}$ acts on the mass point i due to Newton's third law.

force $-\vec{F}_{ij}$ is exerted by the spring which connects the mass point i and the mass point j by which i is the fixed point of the spring and j is the free one. Additionally, the gravitational constant G and external forces, like wind, can be added to $\vec{F}_i$. Afterwards, $\vec{F}_i$ is multiplied by the inverse mass of the mass point to receive an acceleration force. With this information, the new position of the considered mass point can be computed due to some numerical integration method. Consequently, if the inverse mass of the mass point is zero, no forces will act on the mass point. By this, the mass point will be fixed to its initial position. Before the shader can write back the new position, a flow control barrier, using the *barrier()* function, must be applied. This is necessary, because otherwise it comes to a *race condition*.

For lighting purposes, the compute shader can also compute one normal for one mass point's vertex by calculating the cross product of the vectors from i to two adjacent mass points. The sequence of the cross product depends on whether the coordinate system is a *left-handed* or *right-handed* system.

4.3.1 Problems

Newton's third law says that the force $\vec{F}_{ij}$ can be applied to the mass point j and the opposite force $-\vec{F}_{ij}$ can be applied to the mass point i. However, the described implementation does not apply the force $\vec{F}_{ij}$ to the mass point j, therefore equivalent forces are computed. This problem can not be trivially circumvented. A problematic approach would be as follows. The total force $\vec{F}_i$ for every mass point i is stored in a separate SSBO[7]. Each compute shader would calculate up to 12 forces $\vec{F}_{ij}$. After this, the opposite force $-\vec{F}_{ij}$ would be add to the gathered force $\vec{F}_i$ and $\vec{F}_{ij}$ would be add to the gathered force $\vec{F}_j$ for each of the up to 12 mass points j in the SSBO. For this, the gathered force must be read from the SSBO, modified, and be written back. As various shader invocations would do this simultaneously for the same mass point j, *data hazards* can occur, which leads to undefined behavior.

Nevertheless, [29] presents a technique to circumvent the computation of equivalent forces. Thus, they create independent subsets of spring-links trough *graph coloring*.

4.4 Evaluation

For evaluation purposes, a CPU implementation is compared with the compute shader implementation in terms of the computational performance. In order to keep both implementations comparable, the CPU implementation

[7]A separate compute shader integrate the mass points' positions and velocities afterwards.

also does not add the force $\vec{F}_{ij}$ to the mass point j. The computation of the CPU implementation is performed by an *Intel Core i7 920* processor and the compute shader is dispatched to a *GeForce GTX 580*. Figure 13 provides a column chart which shows the measured time in milliseconds that the compute shader and the CPU implementation need to compute different numbers of mass points. For the compute shader, the partitioning of the global work group is adjusted to the number of mass points. For instance, the global work group consists of four local work groups in the x and y dimension which again consists of two threads in each of their x and y dimension if the fabric is discretized through $8^2 = 64$ mass points. As a result, a total of $4 * 4 = 16$ local work groups are dispatched to prevent that streaming multiprocessors are idle (compare section 2.1). In contrast to this, the global work group consists of 8 local work groups in the x and y dimension which again consists of 16 threads in each of their x and y dimension if $128^2 = 16384$ mass points are used. However, free space for adjustments is there. All together, the compute shader offers great performance. A dispatch for 8^2 mass points (connected with 578 springs) needs 0.18 milliseconds in average and a dispatch for 128^2 mass points (connected with 193.418 springs) needs 0.20 milliseconds in average for the computation. This shows that a compute shader - or a modern GPU in general - is optimized for large amount of data. For smaller amounts of data like 8^2 mass points, the CPU implementation is more than three times faster than the compute shader. Nevertheless, for larger amounts of data like 128^2 mass points, the compute shader is more than 35 times faster than the CPU implementation. If the amount of data increases further, the performance difference is even greater. That means for 256^2 mass points (connected with 780.042 springs), the compute shader is more than 90 times faster than the CPU implementation.

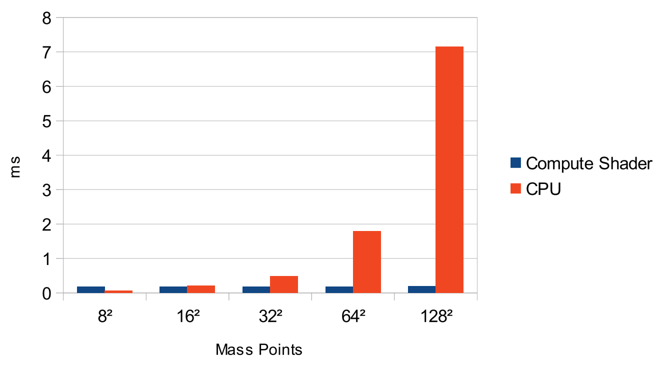


Figure 13: The measured time in milliseconds that the compute shader and the CPU implementation need to compute various numbers of mass points.

5 Line Detection

This chapter describes two different approaches to detect lines in images. Lines are typical image features which computer vision is interested in. The basis for the line detection algorithms of this chapter is some preprocessing step. Thereby an input image gets transformed to an *edge image*, where every pixel (x_i, y_j) of an edge has a pixel value $v(x_i, y_i) = 1$ and every other pixel has a pixel value $v(x_i, y_j) = 0$. One algorithm for this purpose is the *Canny edge detector*, which was developed by John F. Canny in 1986 [13]. Figure 14 shows a photograph of a building and its corresponding edge image computed by the Canny edge detector. One established method to detect lines in an edge image is the *Hough transform*, which is described in section 5.2 in detail.

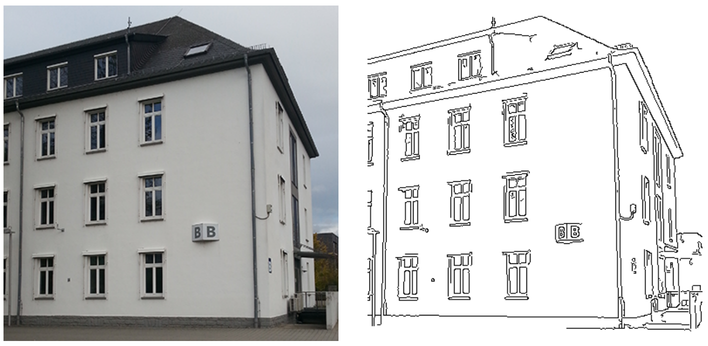


Figure 14: A building, 384×384 in size, and its corresponding edge image produced by the Canny edge detector.
Please note: the edge image is displayed in inverted colors.

During the work on this thesis, we conceived another approach to detecting lines. The goal was to develop a prototype and to compare this prototype to the results the Hough transform provides. Due to the large computing power of modern GPUs and the versatile ability of compute shaders, all possible lines in an image could be scanned. A Canny edge image serves as the input image. The outer points of the input image are numbered as in figure 15. Thus, a Canny input image of size $n_c \times n_c$ has $4n_c - 4 = k$ outer points. If some line rasterization algorithm, like the bresenham line algorithm [8], computes a line from every point k_i to all the points k_j, all possible lines in the image are considered pixel-wise. For this purpose, a compute shader was developed which considers all possible lines in the input image. So each shader invocation rasterizes one line and counts all pixels of the edge input image having a value $v(x_i, y_j) = 1$.

The line in figure 15 starts at k_{12}, ends at k_{43} and fills 16 pixels on its way. Hence, the invocation, rasterizing the line from k_{12} to k_{43}, counts 16

corresponding pixels in the image. In contrast to this the invocation, which rasterizes the line from k_{11} to k_{44}, counts less. All gathered information is stored in a separate texture, called the *line space*.

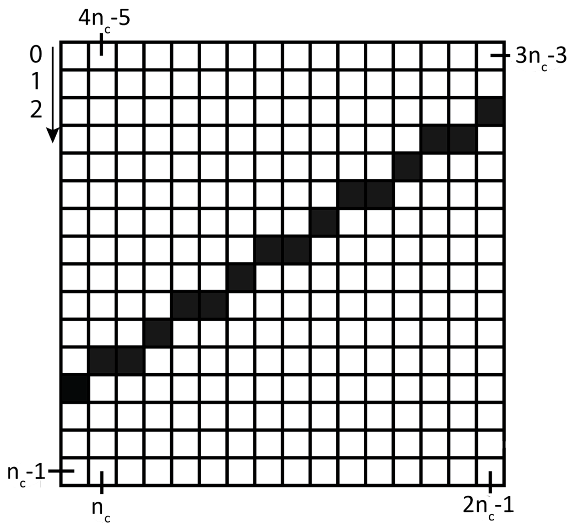


Figure 15: An input image for the line rasterization shader. The outer points k of the image are numbered from 0 to $4n_c - 5$, so that the compute shader can rasterize a line from every point k_i to all the points k_j. This image consists of one line which starts at k_{12} and ends at k_{43}.
Please note: this image is displayed in inverted colors.

5.1 Line Space

The line space is a separate texture which stores the computed information of the line rasterization shader. If a 2D coordinate system is defined in the upper left corner of the line space, then the x-axis represents the starting point k_i and the y-axis represents the ending point k_j of the k outer points of the input image. Figure 16 shows the computed line space produced by the rasterization shader running on figure 15. Each position in the line space represents one possible line of figure 15. The stored values are three dimensional vectors, whereas the first coordinate stores the number of matched line pixels and the second and third coordinates store the startpoint k_i and endpoint k_j of the considered line. Storing the startpoint and endpoint is necessary due to some postprocessing texture filtering.

The size of the line space depends on the input image and the consideration of symmetry properties. Naively, k^2 shader invocations are dispatched, whereby each invocation would rasterize a line from k_i to k_j and stores the result in a line space of size $4n_c \times 4n_c$. Doing this for figure 15, besides the line k_0 to k_1, the line from k_0 to k_2 is also computed. Both lines lie in the same first column. Furthermore this technique would compute the line k_{12} to k_{43} and the equivalent line k_{43} to k_{12}, too. Computing and storing this symmetrical information is neither efficient nor effective. Thus, it is appropriate to consider the symmetry properties. One first approach would be to only compute the lines from k_i to k_j for $0 \leq i \leq 3n_c - 3; i \leq j \leq 4n_c - 5$. Even this computes symmetrical lines. The approach used in the implementation for this thesis is illustrated in appendix A.6. Nevertheless, a line space of size $3n_c \times 3n_c$ can store the necessary information. The position $(12, 27)$ in figure 16 stores the three dimensional vector $(16, 12, 43)$ consequently. More precisely: it stores information of a 16 pixel long line starting at position k_{12} and ending at position k_{43}.

Appendix A.7 shows the line space for the edge image of figure 14.

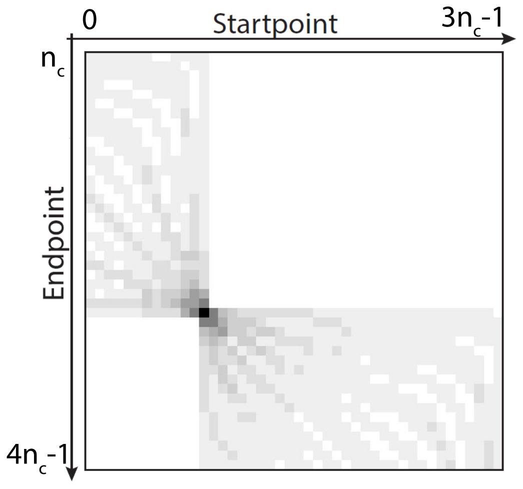


Figure 16: Line space which results from figure 15. The coordinate system's origin is in the upper left corner. The startpoint k_i of a line is plotted on the x-axis and its endpoint k_j is plotted on the y-axis. Due to optimization (see section 5.1), the x-axis goes from 0 to $3n - 1$ and the y-axis goes from n_c to $4n_c - 1$ The line of figure 15 starts in k_{12} and ends in k_{43}. Therefore, the line is encoded in the line space at position $(12, 27)$. *Please note: this image is displayed in inverted colors.*

5.2 Hough Transform

The Hough transform is a method for recognizing complex patterns such as lines, circles or any analytical and non-analytical shapes in images. The initial transform was developed and patented in 1962 by Paul V. C. Hough [30]. Hough describes a geometrical construction of a *transformed space* by mapping points to straight lines. In 1969 Azriel Rosenfeld published an "interesting alternative scheme for detecting straight lines" [41]. Therefore, Rosenfeld defines an algebraic approach based on the U.S. patent of Paul V. C. Hough: A point (x_i, y_i), which has a value $v(x_i, y_i) \neq 0$, in the input image is mapped to a line $y = y_i x + x_i$ in the transformed space, also called the *Hough accumulator.* Thereby the slope x of a line and its y-axis section serve as the axes of the Hough accumulator. Thus, for every $v(x_i, y_i) \neq 0$ in the input image, the line parameters x and y are computed at discrete intervals to cover a set of possible lines. Afterwards the Hough accumulator gets incremented at each position (x, y). This results in a drawn line in the Hough accumulator. Every point of that line represents a possible line in the input image. If a colinear set of points with value $v(x_i, y_i) \neq 0$ in the input image is transformed to lines in the Hough accumulator, all of these lines will pass through a single point. This local maximum decodes a line, given by the colinear set of points in the input image, or at least a similar one. Its slope and y-axis section are given through the specific coordinates of the Hough accumulator.

A line parallel to the y-axis can not be described using the slope-intersect form, which is a disadvantage of this approach. In 1971 Duda and Hart resolved this problem by using an angle-radius normal form rather than the slope-intersect form to describe a line [22]. The normal form of a line is given by

$$x \cos \alpha + y \sin \alpha = d, \tag{8}$$

where d is the algebraic distance from its origin and α is the angle of its normal. By restricting α to the interval $[0, \pi)$, all normal parameters are unique, therefore every line in the input image corresponds to a unique point in the Hough accumulator. If the coordinate system's origin is the center of the image, the maximum distance d to the origin is $D := \sqrt{n_c^2 + n_c^2}/2$. Thus, the resolution of the Hough accumulator depends on the number of discretizations α_{step} of α and the input image of size $n_c \times n_c$. This is, the resolution of the Hough accumulator is given by $(2D \times \alpha_{step})$.

As before, transforming a set of colinear points from the input image to sinusoidal curves in the Hough accumulator will result in one common point of intersection. Figure 17 shows a input image and its resulting Hough accumulator if the normal form of a line is used. Appendix A.8 shows the Hough accumulator for the edge image of figure 14.

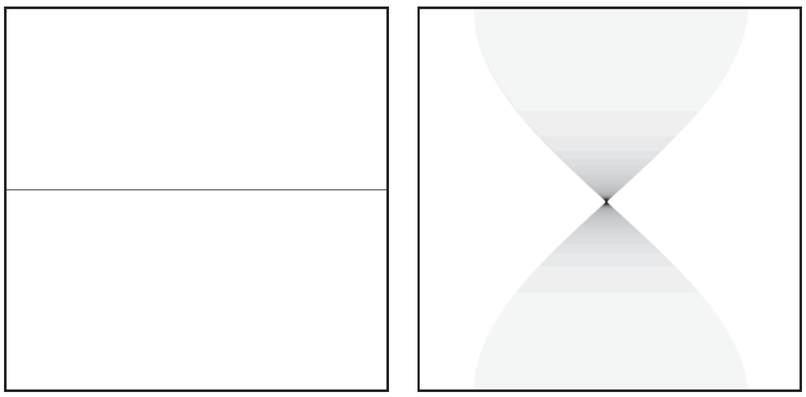


Figure 17: A line and its resulting Hough accumulator. Every point of the line gets mapped to a sinusoidal curve in the accumulator. The common point of intersection of all curves represents the line (d, α) of the input image on the left.
Please note: these images are displayed in inverted colors.

5.2.1 Problems

The Hough transform has several disadvantages. Beside a large requirement of memory, the Hough transform detects many similar lines. This behavior could lead to detected lines which do not exist in the input image. Another problem is the detection of straight lines instead of line segments. Atiquzzaman et al. [2] presented a modified Hough transform to compute line segments instead of straight lines. In addition, the input image gets only subsampled by the Hough transform. This is because the Hough transform computes the parameters in discrete interval steps. Consequently, a detected line in the Hough accumulator does not have to be the exact line of the input image but can vary in d and α. This can be too inaccurate for some applications like real-time markerless tracking for augmented reality.

5.3 Interpretation

Both results, the Hough accumulator as well as the line space (here after both called *result texture*), of size $n_r \times n_r$ must be interpreted to use their information[8]. A line in the input image produces a region in the result texture. Thus, possible lines are gathered in the neighborhood of a detected line. This means that a larger line in the input image produces a more pronounced region than a short line will do. Pixels of such regions can store greater values than pixels of regions produced by shorter lines. So, if

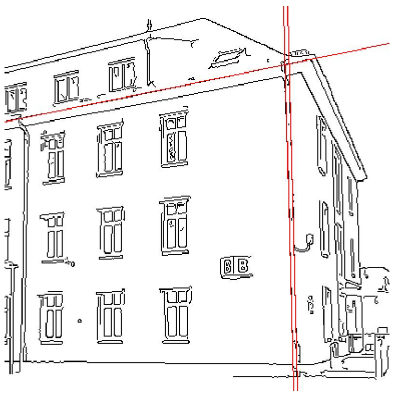
[8]In general, the Hough accumulator has no quadratic size. However, the algorithms described in the next sections consider the Hough accumulator to be quadratic.

a post-process computes the m-maxima of the result texture, non-existing lines could suppress shorter existing lines.

This section discusses three approaches to interpret the result texture. *Thresholding* simply filters the results by a threshold. The approaches described in section 5.3.2 and section 5.3.3 search for the m-maxima in the result texture, and therefore the m most distinct lines in the input image.

5.3.1 Thresholding

One approach to interpret the result texture is thresholding. For this, a threshold $t \in \mathbb{N}$ is defined. Now, every pixel (x_i, y_i) is considered. If $v(x_i, y_i) \geq t$, the pixel is regarded as a detected line. The size of the threshold t should depend on the size of the input image as well as on the size of the lines which are expected. If t is chosen to small, this approach detects too many lines. Conversely, an overlarge t would detect too few or, in the worst case, even no lines in the image. Consequently, thresholding is not a suitable technique to find the m most distinct lines in the input image. Figure 18 shows the interpretation of the result texture due to thresholding. Thereby an individual t is applied to filter out three lines from the Hough accumulator and the line space, respectively. Another drawback of thresholding is the suppression of shorter lines by longer ones.


(a) Thresholding with $t = 230$ applied to the line space.

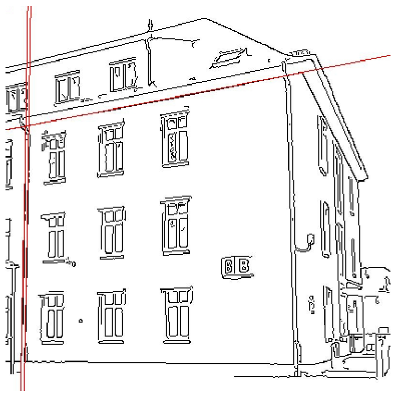

(b) Thresholding with $t = 160$ applied to the Hough accumulator.

Figure 18: Thresholding applied to the line space (a) and the Hough accumulator (b). A specific t was chosen to filter out three lines each.
Please note: these images are displayed in inverted colors.

5.3.2 Image Reduction

The number of detected lines by using thresholding depends on the used threshold t and the input image. A stable method, which detects exactly m lines, is desirable. In general, one could search for the m-maxima by going through the result texture sequentially. Nevertheless, a preferable technique would be to search for the m-maxima on the GPU. For this, the line space is divided into a grid using a compute shader. Each compute shader invocation reduces the four entries of a grid cell to one entry, so that only the largest entry remains (see figure 19). Repeating the image reduction $ld(n_r)$ times will lead to the maximum value in the result texture, and therefore to the most distinct line in the input image. As described in section 5.1, each entry in the result texture[9] additionally stores its position (x_i, y_i). With this knowledge, the detected line can be deleted in the original result texture by setting its value $v(x_i, y_i) = (0, 0, 0)$. The next maximum in the input image will be another distinct line, which is different from the first one. Therefore, doing this procedure m times will deliver the m-maxima of the result texture.

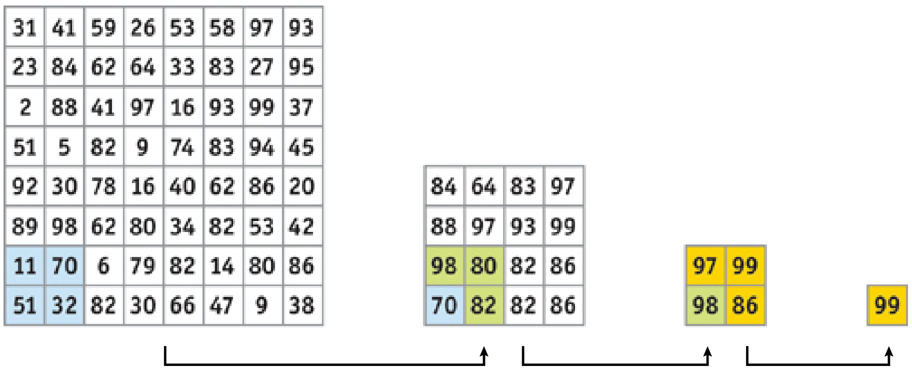
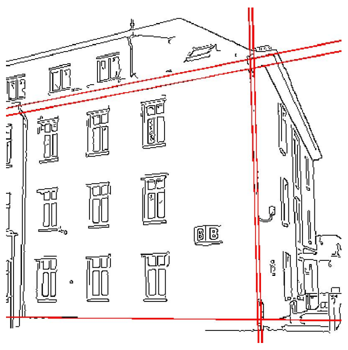


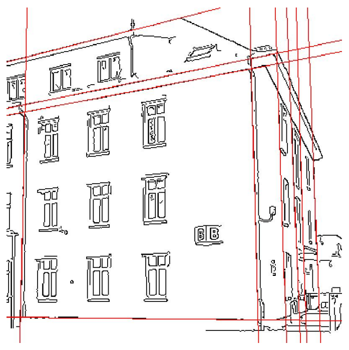
Figure 19: Image reduction of an $n_r \times n_r$ input image applied $ld(n_r) = 3$ times to find the global maximum. [12]

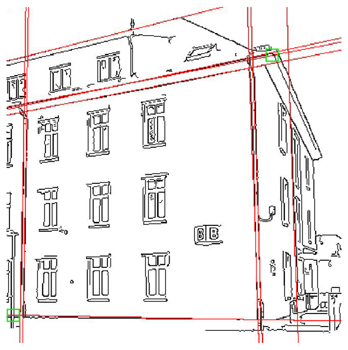
A line in the input image produces a region in the result texture, which contains similar lines. Therefore, it can improve the results, if the neighborhoods of the maxima gets deleted in the result texture. Figure 20 provides the result of image reduction applied to the line space and to the Hough accumulator. Applying image reduction without deleting the neighborhood provides better results for the Hough accumulator (figure 20c) than for the line space (figure 20a) as some similar lines have been detected in figure 20a. In contrast to this, deleting the 10×10 neighborhoods of the maxima

[9]Section 5.1 describes that each entry of the line space is a three dimensional vector, where the second and third value store the position. This technique can be transferred to the Hough accumulator too.

in the line space improves the result as shown in figure 20b. Thus, ten different and actually existing lines are detected. Figure 20c and 20d shows the slight inaccuracy of the Hough transform (marked with green rectangles). This occurs because the Hough tranform samples the input image in discrete intervals. Here, the angle α of the normal was sampled in 180 discrete steps, as described in section 5.2.1.


(a) Interpretation of the line space without deleting the neighborhood of the maxima.


(b) Interpretation of the line space with deleting the 10×10 neighborhood of the maxima.


(c) Interpretation of the Hough accumulator without deletion of the neighborhood of the maxima.

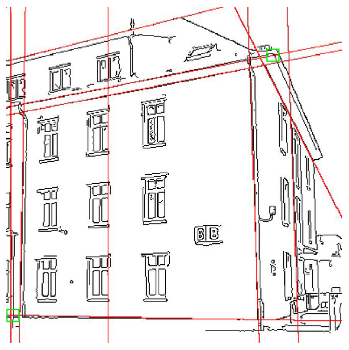

(d) Interpretation of the Hough accumulator with deletion of the 5×5 neighborhood of the maxima.

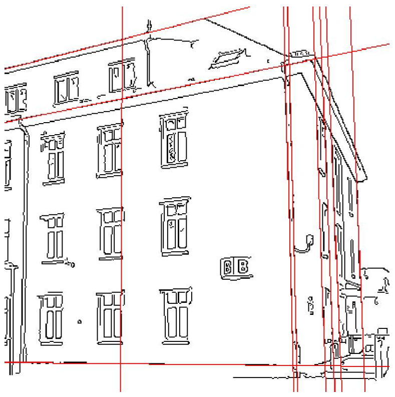
Figure 20: Ten detected lines in the input image, obtained through reducing the line space (a,b) and the Hough accumulator (c,d) $ld(n_r)$ times per line. A green rectangle indicates the inaccuracy of the Hough tranform. *Please note: all images are displayed in inverted colors.*

5.3.3 Sorting

Section 5.3.2 provides an algorithm to detect the m most distinct lines in the input image. For this, the m-maxima of the result texture are found through reducing it $m * ld(n_r)$ times. Consequently, the computational performance of the algorithm depends on m. This section provides an approach to detect any number of lines with almost constant performance. For this, the result texture gets sorted by using a sorting algorithm which is appropriate for GPUs. There are two categories of sorting algorithms: data-driven ones and data-independent ones. Thereby, data-independent sorting algorithms are well suited to be implemented for multiple processors, therefore to run on the GPU [34]. The most incisive algorithms in the literature are the *bitonic merge sort* [34] and the *radix sort* [27]. The first m texel of the sorted result texture are the m most distinct lines in the input image. Consequently, if the unfiltered result texture is sorted, the detected lines are the same as through reducing the result texture $m * ld(n_r)$ times without deleting the neighborhoods of the maxima. To improve the detected lines, the result texture can be filtered in a preprocessing step. One approach is to use the image reduction technique as in figure 19. Importantly, the result texture is not reduced $ld(n_r)$ times but less. This will reduce the regions a line in the input image produces in the result texture and will suppress false lines in such a region. Results of applying this approach to the result texture are shown in figure 21. Furthermore, appendix A.9 shows the effects of different frequent image reduction on the line space with subsequent sorting.

5.4 Evaluation

This section compares the line space, Hough accumulator and the three different interpreting techniques in terms of their computational performance. The computational performance of the line space was compared with the performance of the Hough transform. The Hough transform, used for the evaluation, is performed by OpenCV [14], hence running on CUDA. Thereby, the Hough accumulator has a resolution of $(2D \times \alpha_{step}) = (\sqrt{n_c^2 + n_c^2} \times 180)$. Figure 22 provides a column chart which shows the measured time in milliseconds. The Canny edge image of figure 14 acts as the input image and the computation is performed by the *GeForce GTX 580* GPU. The Hough transform is a lot faster than the line space computation, especially for larger images. In detail, for an input image of 256×256, the line space needs an average of 3.36 ms and the Hough accumulator needs an average of 1.39 ms to be computed. For an input image of size 768×768, the Hough transform is up to 17 times faster than the line space computation. Here, the input image consists of $768 * 768 = 589.824$ pixels, whereby 27.555 are pixels of possible lines. Thus, their value is


(a) Interpreted line space which has been reduced five times and has been sorted.

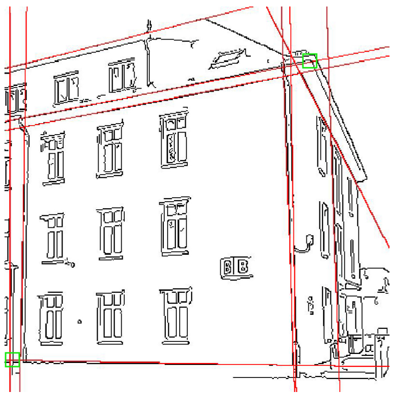

(b) Interpreted Hough accumulator which has been reduced three times and has been sorted.

Figure 21: Ten detected lines in the input image. The line space (figure a) and the Hough accumulator (figure b) have been reduced several times and have been sorted subsequently. A green rectangle indicates the inaccuracy of the Hough transform.
Please note: these images are displayed in inverted colors.

$v(x, y) = 1$. The Hough transform computes a set of possible lines for each of these pixels by calculating their distance d to the origin based on an angle $\alpha_{degree} \in [0, 180)$. As described before, the resolution of the Hough accumulator is $(2D \times \alpha_{step}) = (\sqrt{n_c^2 + n_c^2} \times 180)$, resulting in 180 sampling steps per pixel. Consequently, equation 8 is computed $27.555 * 180 = 4.959.900$ times. In contrast to this, the line space algorithm scans all possible lines in the image as described in section 5.1, therefore $3.526.670$ lines are scanned. Taking into account that a line consists of many pixels, a huge number of texture accesses are necessary. This can explain the large performance differences in favor of the Hough transform.

Figure 23 compares two interpretation techniques applied to the line space of an input image of size 384×384. The green line stands for $m * ld(n_r)$ reductions of the line space, when the interpretation technique of section 5.3.2 is applied. The cost increases linearly with the number of lines m. In contrast to this, the four horizontal lines represent the technique of section 5.3.3, running independently of the number of lines m at a constant speed. It can be concluded that applying $ld(n_r)$ image reductions m times to the line space provides good performance for small number of lines. If multiple lines must be detected, the performance is better when the line space is reduced only a few times and then gets sorted. However, one must be careful that the reduction is not applied too often in order to not lose important information about existing lines.

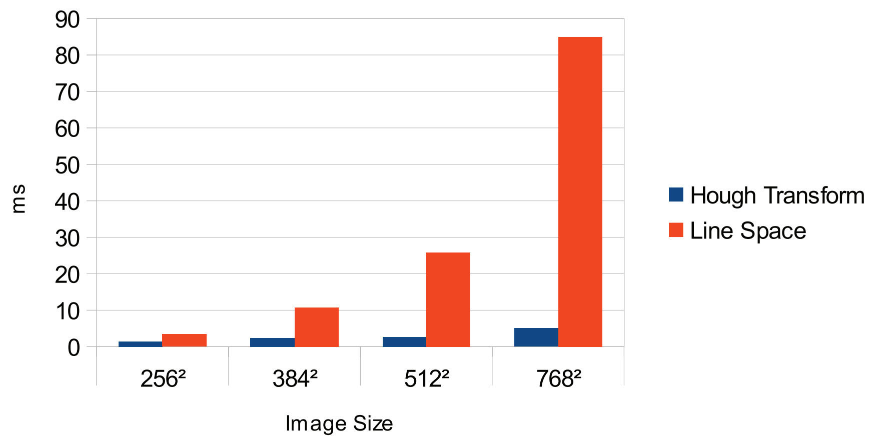


Figure 22: Computational performance in milliseconds of the Hough transform (blue) and the line space (orange) applied to the Canny edge image of figure 14 in different sizes. The Hough accumulator has a resolution of $(2D \times \alpha_{step})) = (\sqrt{n_c^2 + n_c^2} \times 180)$.

The evaluation of the quality of the detected lines is mostly a subjective assessment. Nevertheless, some statements can be made:

- Thresholding does not prevent good results since it does not deal with regions around an existing line in the result texture. Accordingly, smaller lines are not detected in the presence of larger ones. Furthermore, thresholding is not suitable when a defined number of lines m have to be detected.

- Reducing the result texture $m * ld(n_r)$ times provides the m most distinct lines in the image. Deleting the neighborhood of the maxima in the result texture can improve the quality of the detected lines.

- Reducing the result texture only a few times and sorting it subsequently can provide good results if the number of reduction steps is chosen carefully.

- The Hough transform does not only detect lines which are congruent with the input image, due to its discretization. The line space instead provides lines which are congruent with the input image.

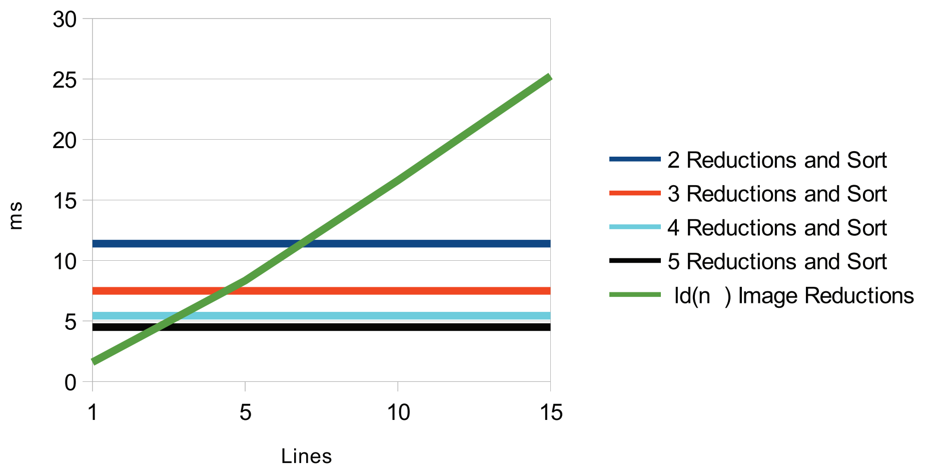


Figure 23: Comparison of two interpretation techniques applied to the line space of an input image of size 384×384.

5.5 Future Work

Beyond this thesis, the line space algorithm to detect lines in an edge image can be extended. Section 3 has shown that the use of group shared memory has a great influence on the compute shader's performance. However, the compute shader implementation of this thesis does not utilize group shared memory. The size of the line space is larger than necessary. Appendix A.7 shows a lot of unused texture space. If some optimization can be found, one third of the line space's size can be saved.

Another interesting topic is the detection of line segments instead of infinite lines. For this purpose, the bresenham line algorithm could be extended. Instead of storing the absolute number of pixels having a value $v(x, y) = 1$, the algorithm could store every set of colinear pixels, found on the way from k_i to k_j, in a separate texture layer.

6 Conclusion

Compute shaders extend the OpenGL API with the possibility to use the GPU for general-purpose computations. As compute shaders are written in GLSL, their development is similar to the conception and implementation of other established shader types. Thus, a GLSL developer does not have to learn any additional APIs to do general-purpose computation on graphics processing units. In addition to graphics-based shader types, compute shaders introduce high efficient group shared memory which allows the communication of compute shader invocations within one local work group. On the one hand, it can be difficult to find an effective way to use shared memory. On the other hand, they extend the GLSL development with great flexibility. Chapter 3 has shown that the use of shared memory has a great influence on the program's performance when data is reused. Modern GPUs are general-purpose many-core processors with a very high data and computation throughput. Consequently, one should always make sure that a compute shader works on a large data set and that enough local work groups are dispatched to prevent streaming multiprocessors from idling. Hence, it can be stated that at least 16 local work groups should be dispatched on a *GeForce GTX 580* to keep its 16 streaming multiprocessors busy. Furthermore, the local work group size should be a multiple of 32, which is the warp size. Chapter 4 has shown that compute shaders are suitable for physical simulations - especially in a direct comparison to a CPU implementation. Porting a CPU implementation to the GPU using compute shaders is not a difficult task for a GLSL developer. Nevertheless, the chapter has also shown that data hazards can occur quickly and that their prevention can be difficult and time-consuming. In chapter 5, compute shaders were used to conceive and develop a new approach to detect lines in an image. Therefore, all possible lines in an image are scanned. For evaluation purposes, the implemented prototype was compared with the results of the Hough transform. The detected lines are good and, in terms of accuracy, better than the detected lines by the Hough transform. However, the implemented prototype is much slower than the Hough transform, and more optimization is desirable.

In conclusion, it can be stated that compute shaders have not only a right to exist, they complement OpenGL in a great way.

A Additional Figures

A.1 OpenGL 4.4 Pipeline

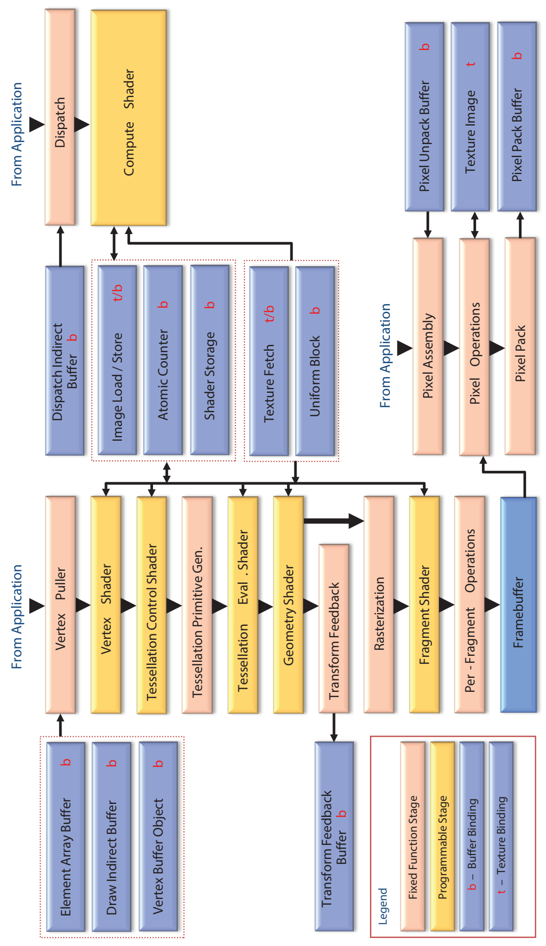


Figure 24: OpenGL 4.4 pipeline. Reddish blocks are fixed function stages, whereas yellow blocks are programmable through shader programs. The compute shader block is separated from the rest of the graphics pipeline.[42, p. 32]

A.2 Fermi Architecture

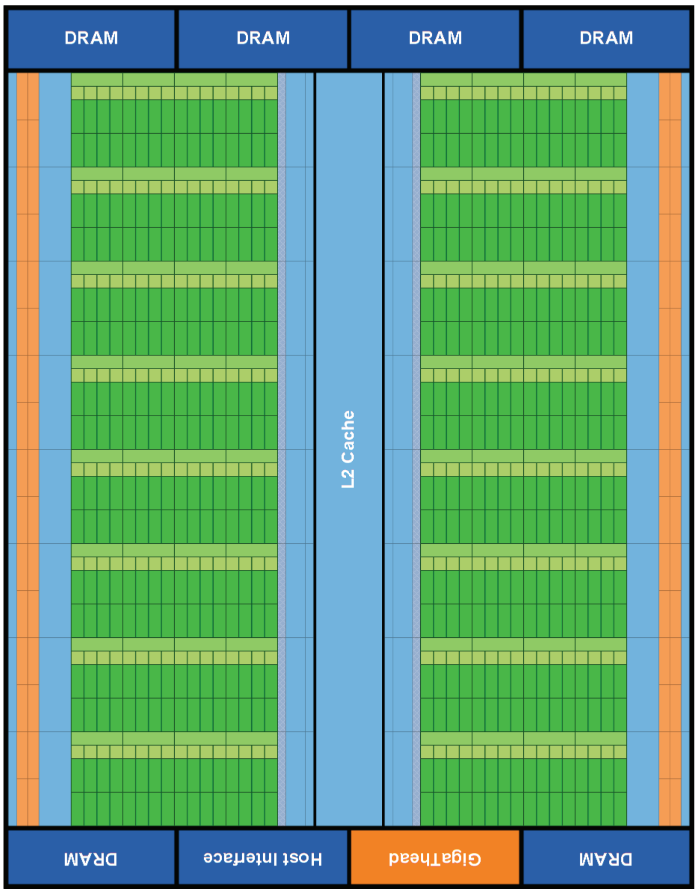


Figure 25: Fermi's 16 streaming multiprocessors (SM) positioned around a L2 cache. Each SM consists of 32 CUDA cores. [18, p. 7]

A.3 Kepler Architecture

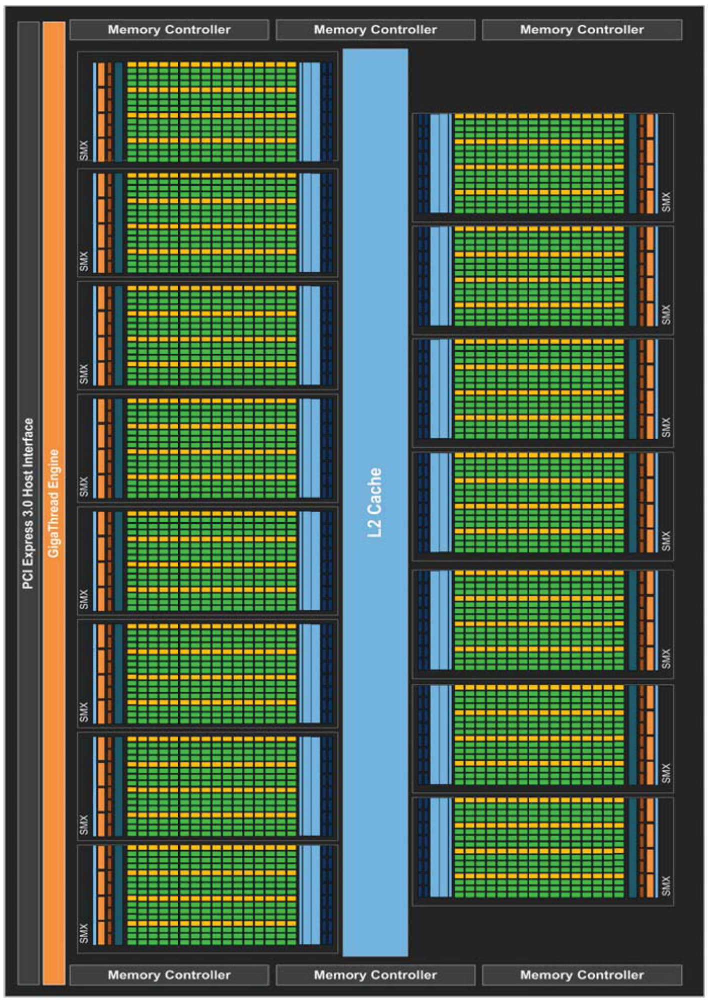


Figure 26: Kepler's 15 new streaming multiprocessors (SMX) positioned around a L2 cache. Each SMX consists of 192 CUDA cores. [19, p. 6]

A.4 Kepler Streaming Multiprocessor

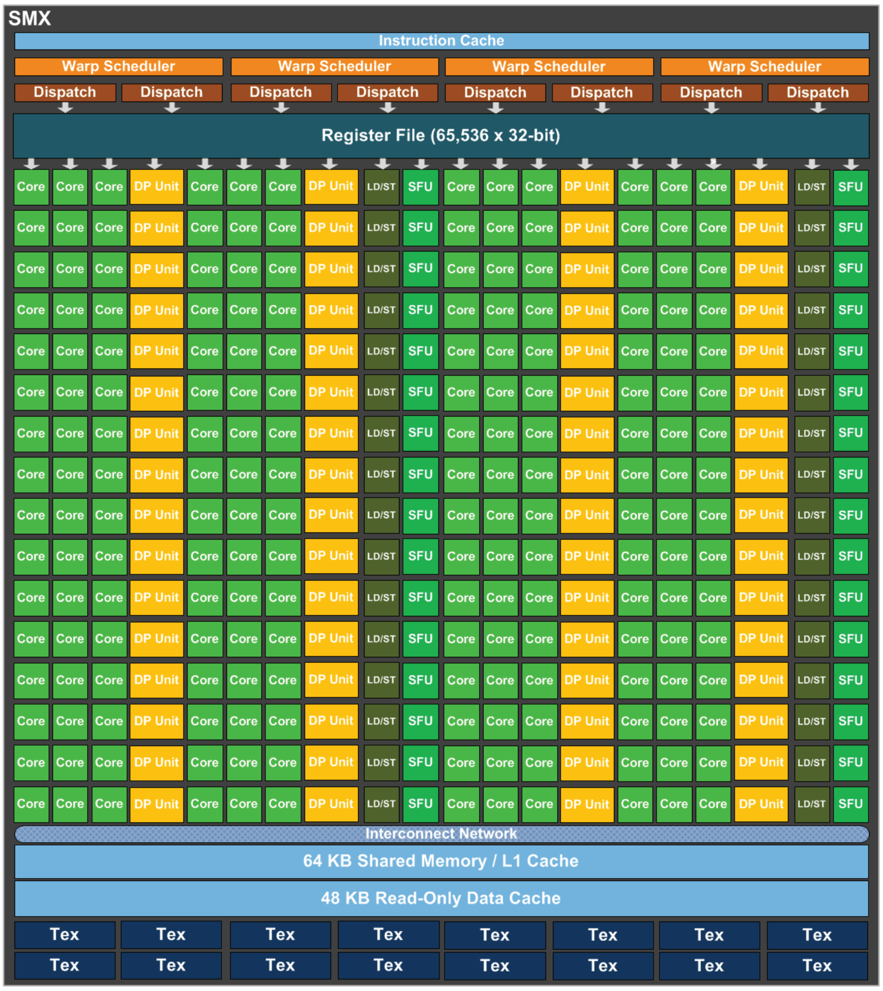
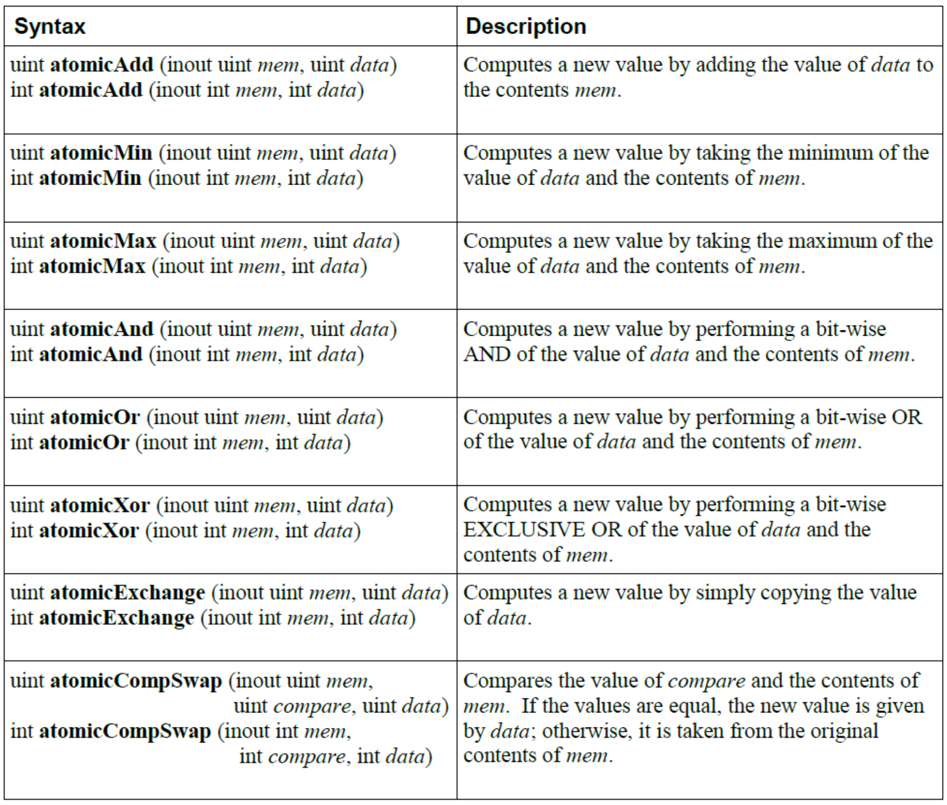


Figure 27: Kepler's Streaming Multiprocessor (SMX) consisting of 192 CUDA cores, 64 double-precision units (DP Unit), 32 special function units (SFU), and 32 load and store units (LD/ST). [19, p. 8]

A.5 Atomic Memory Functions



Syntax	Description
uint **atomicAdd** (inout uint *mem*, uint *data*) int **atomicAdd** (inout int *mem*, int *data*)	Computes a new value by adding the value of *data* to the contents *mem*.
uint **atomicMin** (inout uint *mem*, uint *data*) int **atomicMin** (inout int *mem*, int *data*)	Computes a new value by taking the minimum of the value of *data* and the contents of *mem*.
uint **atomicMax** (inout uint *mem*, uint *data*) int **atomicMax** (inout int *mem*, int *data*)	Computes a new value by taking the maximum of the value of *data* and the contents of *mem*.
uint **atomicAnd** (inout uint *mem*, uint *data*) int **atomicAnd** (inout int *mem*, int *data*)	Computes a new value by performing a bit-wise AND of the value of *data* and the contents of *mem*.
uint **atomicOr** (inout uint *mem*, uint *data*) int **atomicOr** (inout int *mem*, int *data*)	Computes a new value by performing a bit-wise OR of the value of *data* and the contents of *mem*.
uint **atomicXor** (inout uint *mem*, uint *data*) int **atomicXor** (inout int *mem*, int *data*)	Computes a new value by performing a bit-wise EXCLUSIVE OR of the value of *data* and the contents of *mem*.
uint **atomicExchange** (inout uint *mem*, uint *data*) int **atomicExchange** (inout int *mem*, int *data*)	Computes a new value by simply copying the value of *data*.
uint **atomicCompSwap** (inout uint *mem*, uint *compare*, uint *data*) int **atomicCompSwap** (inout int *mem*, int *compare*, int *data*)	Compares the value of *compare* and the contents of *mem*. If the values are equal, the new value is given by *data*; otherwise, it is taken from the original contents of *mem*.

Figure 28: Atomic memory functions [32, p. 169]

A.6 Effective Consideration of all possible Lines in an Image

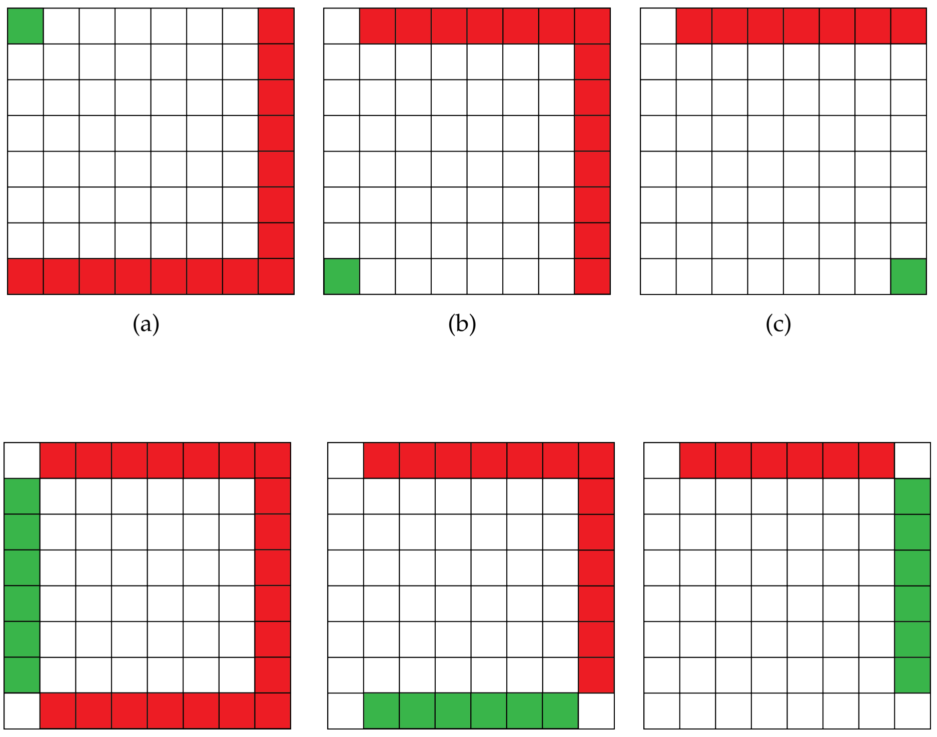


Figure 29: An effective way to consider all possible lines in an image, without considering equivalent lines. Startpoints are colored green, whereby the endpoints are colored red. From each startpoint, lines are considered to each endpoint. The corner points are considered separately (a-c) to prevent the calculation of equivalents of the outermost horizontal and vertical lines. The rest of the line space is covered by figures d-f.

A.7 Line Space of a Building

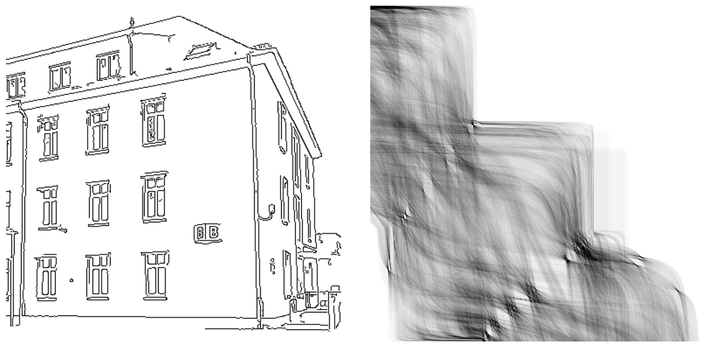


Figure 30: A building, given as a Canny edge image, and its corresponding line space. Unused texture space is conspicuous, therefore more optimization is desirable.
Please note: all images are displayed in inverted colors.

A.8 Hough Accumulator of a Building

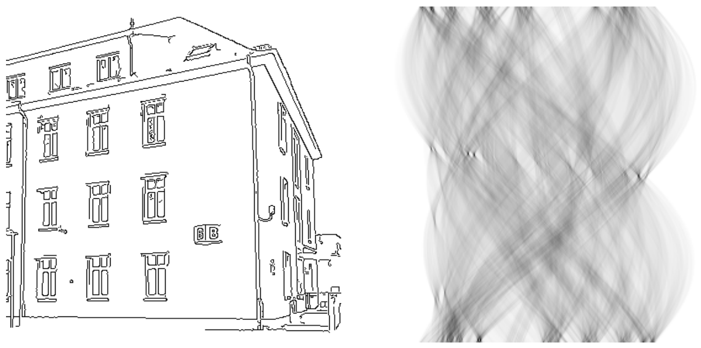
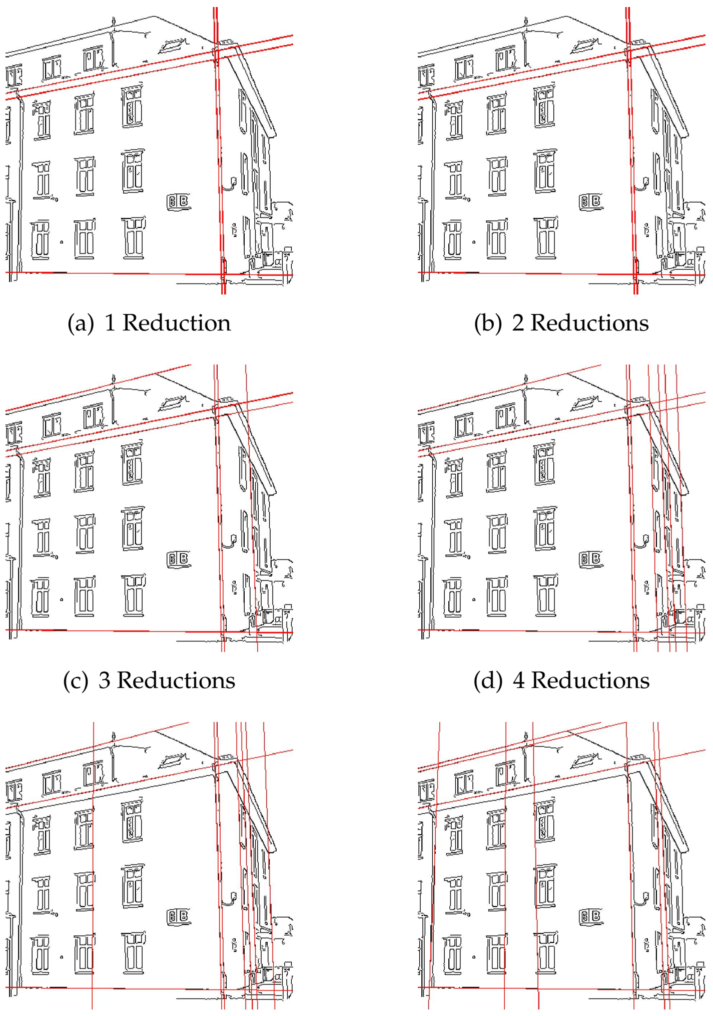


Figure 31: A building, given as a Canny edge image, and its corresponding Hough accumulator.
Please note: all images are displayed in inverted colors.

A.9 Effects of repeated Reduction on the Line Space with subsequent Sorting


(a) 1 Reduction

(b) 2 Reductions

(c) 3 Reductions

(d) 4 Reductions

(e) 5 Reductions

(f) 6 Reductions

Figure 32: Ten detected lines in the input image obtained from the line space by using the interpretation technique described in section 5.3.3

B List of Figures

C List of Listings

References

[1] Wolfgang Andermahr. Test: Nvidia GeForce GTX 580. http://www.computerbase.de/artikel/grafikkarten/2010/test-nvidia-geforce-gtx-580/2/#abschnitt_technische_daten, 2010. [Online; accessed 23-October-2013].

[2] M. Atiquzzaman and M.W. Akhtar. Complete line segment description using the hough transform. *Image and Vision Computing*, 12:267–273, 1994.

[3] Mike Bailey. OpenGL Compute Shaders. http://education.siggraph.org/media/conference/S2012_Materials/ComputeShader_1pp.pdf, 2012. [Online; accessed 03-October-2013].

[4] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. 1986.

[5] Guy Blelloch and Girija Narlikar. A practical comparison of n-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[6] John A Board Jr, Ziyad S Hakura, William D Elliott, Daniel C Gray, William J Blanke, and James F Leathrum Jr. Scalable implementations of multipole-accelerated algorithms for molecular dynamics. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 87–94. IEEE, 1994.

[7] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Eric Werness, Graham Sellers, Greg Roth, Nick Haemel, Pierre Boudier, and Piers Daniell. ARB_shader_image_load_store. http://www.opengl.org/registry/specs/ARB/shader_image_load_store.txt, 2011. [Online; accessed 05-October-2013].

[8] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[9] Pat Brown, Jeff Bolz, Piers Daniell, Christophe Riccio, Graham Sellers, Bruce Merry, and John Kennenich. ARB_shader_storage_buffer_object. http://us.download.nvidia.com/opengl/specs/GL_ARB_shader_storage_buffer_object.txt, 2012. [Online; accessed 06-October-2013].

[10] Pat Brown, Slawomir Grajewski, Jeannot Breton, and Daniel Koch. ARB_compute_variable_group_size. http://www.opengl.org/registry/specs/ARB/compute_variable_group_size.txt, 2013. [Online; accessed 03-October-2013].

[11] Pat Brown and John Kessenich. ARB_shader_group_vote. http://www.opengl.org/registry/specs/ARB/shader_group_vote.txt, 2013. [Online; accessed 03-October-2013].

[12] Ian Buck and Tim Purcell. A toolkit for computation on gpus. *GPU Gems*, pages 621–636, 2004.

[13] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[14] Intel Corporation, Willow Garage, and Itseez. Open Source Computer Vision. http://www.opencv.org, 2013. [Online; accessed 15-November-2013].

[15] Microsoft Corporation. Compute Shader Overview. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx, 2013. [Online; accessed 16-November-2013].

[16] NVIDIA Corporation. Architecture. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/architecture. [Online; accessed 24-October-2013].

[17] NVIDIA Corporation. Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications. [Online; accessed 23-October-2013].

[18] NVIDIA Corporation. *Whitepaper NVIDIA's Next Generation CUDA™Compute Architecture: Fermi™*. [Online; accessed 24-October-2013].

[19] NVIDIA Corporation. *Whitepaper NVIDIA's Next Generation CUDA™Compute Architecture: Kepler™GK110*. [Online; accessed 24-October-2013].

[20] NVIDIA Corporation. *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview*, 2006. [Online; accessed 25-October-2013].

[21] NVIDIA Corporation. *Tuning CUDA Applications for Kepler*, 2013. [Online; accessed 25-October-2013].

[22] Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[23] D Eberly. *Game Physics. Series in Interactive 3D Technology*. Morgan Kaufmann, 2003.

[24] Marco Fratarcangeli. Gpgpu cloth simulation using glsl, opencl, and cuda. *Game Engine Gems 2*, pages 365–378, 2010.

[25] Leslie Greengard. *The rapid evaluation of potential fields in particle systems*. the MIT Press, 1988.

[26] Khronos Group. Khronos Releases OpenGL 4.3 Specification with Major Enhancements. https://www.khronos.org/news/press/khronos-releases-opengl-4.3-specification-with-major-enhancements, 2012. [Online; accessed 23-September-2013].

[27] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[28] Peter E Hart. How the hough transform was invented [dsp history]. *Signal Processing Magazine, IEEE*, 26(6):18–22, 2009.

[29] Justin Hensley and Lee Howes. Bullet Cloth Simulation, 2010. SIGGRAPH Asia 2010.

[30] Paul V. C. Hough. Method and means for recognizing complex patterns, Dec. 18, 1962. U.S. Patent 3 069 654.

[31] Bill Kane-Licea, Barthold Lichtenbelt, Chris Dodd, Eric Werness, Graham Sellers, Greg Roth, Jeff Bolz, Nick Haemel, Pat Brown, Pierre Boudier, and Piers Daniell. ARB_shader_atomic_counters. http://www.opengl.org/registry/specs/ARB/shader_atomic_counters.txt, 2011. [Online; accessed 06-October-2013].

[32] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL Shading Language. http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf, 2013. [Online; accessed 04-October-2013].

[33] Mark Kilgard. OpenGL 4.3 and Beyond, 2012. SIGGRAPH Asia 2012.

[34] Peter Kipfer and Rüdiger Westermann. Improved gpu sorting. *GPU gems*, 2:733–746, 2005.

[35] Donald E Knuth. The art of computer programming, volume 3: sorting and searching, 1973.

[36] Tosiyasu L Kunii and Hironobu Gotoda. Singularity theoretical modeling and animation of garment wrinkle formation processes. *The Visual Computer*, 6(6):326–336, 1990.

[37] Ian Millington. *Game physics engine development*. Taylor & Francis US, 2007.

[38] Tianyun Ni. Direct Compute - Bring GPU Computing to the Mainstream. http://on-demand.gputechconf.com/gtc/2009/presentations/1015-Features-Advantages-DirectCompute.pdf, 2009. [Online; accessed 24-October-2013].

[39] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. *GPU gems*, 3:677–695, 2007.

[40] Rick Parent. *Computer Animation Algorithms and Techniques*. Morgan Kaufmann Publishers, 2002.

[41] Azriel Rosenfeld. Picture processing by computer. *ACM Computing Surveys (CSUR)*, 1(3):147–176, 1969.

[42] Mark Segal and Kurt Akeley. The OpenGL®Graphics System: A Specification (Version 4.4 (Core Profile) - October 18, 2013. http://www.opengl.org/registry/doc/glspec44.core.pdf, 2013. [Online; accessed 23-October-2013].

[43] Graham Sellers, Pat Brown, Daniel Koch, and John Kessenich. ARB_compute_shader. http://www.opengl.org/registry/specs/ARB/compute_shader.txt, 2012. [Online; accessed 23-September-2013].

[44] Michele Trenti and Piet Hut. N-body simulations (gravitational). *Scholarpedia*, 3(5):3930, 2008.

[45] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

[46] Pascal Volino and Nadia Magnenat-Thalmann. *Virtual Clothing Theory and Practice*. Springer-Verlag Berlin Heidelberg, 2000.

[47] Richard S. Wright, Nicholas Haemel, and Graham Sellers. *OpenGL SuperBible: comprehensive tutorial and reference*. Pearson Education, 2013.