Adam Mickiewicz University in Poznań
Faculty of Mathematics and Computer Science

Miłosz Kosobucki
index no. 329519

Modelling System of Karst Caves for Computer Graphics

(Modelowanie systemu jaskiń krasowych dla grafiki komputerowej)

Master's thesis
in Computer Science
written under supervision of
Wojciech Kowalewski PhD

Poznań, 2013

Oświadczenie

Ja, niżej podpisany **Miłosz Kosobucki** student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt.:

Modelling system of karst caves for computer graphics

(Modelowanie systemu jakiń krasowych dla grafiki komputerowej)

napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w formie wydruku komputerowego jest zgodny z egzemplarzem pracy dyplomowej w formie elektronicznej.

Jednocześnie przyjmuję do wiadomości, że gdyby powyższe oświadczenie okazało się nieprawdziwe, decyzja o wydaniu mi dyplomu zostanie cofnięta.

.....	.....
data	podpis

Contents

Chapter 1

Introduction

Karst formations are ubiquitous in every continent of Earth. It's estimated that about 25% of Earth population depends on drinking water obtained from karst aquifers (Ford and Williams 2007). With such profound influence on human race, it is essential to know how these geological structures evolve and how they may react to human activity.

Various simulation models were developed that try to predict how karst aquifers evolve in time, and how they react to changes in environment. These models are implemented in computer software and represent simulated karst structure as net of fractures.

These tools, being aimed at speleogenesis experts, present results of calculations with simple plots. Programming project of this thesis called *karstgen* provides solution for richer presentation of geometric structure of modelled karst formation. It can take input data in format that is similar to formats of files produced by simulation software and generate triangle mesh in two file formats, one of which is simple and popular textual file format supported by most three–dimensional modelling software. Presentation in such program may be beneficial for better understanding of data or for later usage in e.g. video games.

Since karst evolution models usually simulate large datasets, karstgen uses GPU acceleration to speed–up mesh generation process.

1.1 Structure of the thesis

Below is overview of each chapter along with description on how it contributes to the thesis.

Chapter 2 – Karst and karstification process
This chapter introduces basic definitions related to karst landscape forms used later in the work. Karstification process is presented with basic overview of chemical reactions that drive it. Information contained in this chapter provides rationale for the shape of karst evolution simulation models presented in subsequent chapters.

Chapter 3 – Related work
References to various works relevant to the subject are presented here. Brief overview of karst evolution models is presented what explains some of the design decisions taken in the programming project.

Several works related to rendering, rather than simulating, caved terrains are also described.

Chapter 4 – OpenCL heterogeneous programming platform
Here, OpenCL is described. It's a programming library, maintained by Khronos Group Inc., that lets programmers leverage diverse computational resources offered by modern computers in standardized manner. OpenCL is extensively used by programming project of this thesis so this introduction gives reader a background for understanding implementation details.

Chapter 5 – Isosurface extraction with Marching Cubes
In this chapter Marching Cubes algorithm that is used in programming project is described. This algorithm extracts polygonal meshes of isosurfaces from three–dimensional scalar functions. GPU–accelerated variant used in project is also described.

Chapter 6 – Programming project description
This chapter describes features and architecture of programming project, as well as some more in–depth technical details of the implementation.

Chapter 7 – Conclusions and further work
Final chapter of the thesis summarizes results of the programming

project. Possible use cases are presented. Potential new areas of improvement and further development are discussed as well.

Chapter 2

Karst and karstification process

2.1 Introduction

This chapter will briefly describe karst and processes that govern the development of karst caves. First, basic definitions are introduced followed by description of elements of karst landscape and formations. Next, a high level overview of the karstification process is presented along with chemical reactions in limestone aquifers that are essential in formation of caves. Process and chemistry of speleothems formation is described at the end of the chapter.

2.2 Basics

Definitions

Karstification is not a strictly defined term. Depending on context it may mean all forms of corrosion of soluble rocks or it may encompass whole range of processes that lead to devolopment of karst formations.

Usually karstification means a landscape forming process that consists of dissolution of various kinds of bedrock. The most common kinds of solutes are limestone, dolomite, and gypsum (Field 2002). However, given right conditions even some weathering–resistant rocks like quartzite may be subject to karstification (Migoń 2010).

Although chemical dissolution is the main driving force behind karstification, mechanical forces may also play a role in the fi-

nal looks of the karst landscape. That's why sometimes, all these forces together are put under the umbrella term of karstification.

Karst terrain formation developed through the means of *karstification*. The origin of the term is a German form of Slavic word kras or krš meaning bleak, waterless place (Field 2002).

Karst cave hollow space in a karst structure that is large enough for human to enter (Hill and Forti 1997)

Aquifer geological formation that is capable of holding large amounts of water through porosity, and other empty spaces inside.

Recharge process of addition of water to an aquifer.

Elements of karst landscape

Karstification process may produce very interesting and varied landscape. Some of the prominent elements of karst landscapes seen in figure 2.1 are:

Sinkholes general term for closed depressions of various shapes.

Resurgences places where water that entered the aquifer is re–emerging to the surface

Tunnels or corridors medium–sized passages connecting larger voids.

Shafts vertical or steeply inclined passages of varying sizes. Deepest known shafts are up to half kilometre deep (Field 2002, p. 167).

Speleothems limestone formations built through calcium carbonate precipitation. Speleothems that hang down from the ceiling are called *stalactites* and ones that are emerging from the floor are called *stalagmites*. When stalactites and stalagmites merge, they form *limestone columns* – not to be confused with *pillars*.

Pillars vertical columns of rock that are left when surrounding rock material dissolves.

Ponors openings in bottom or sides of depressions through which water emerges or into which it disappears. Partially or completely.

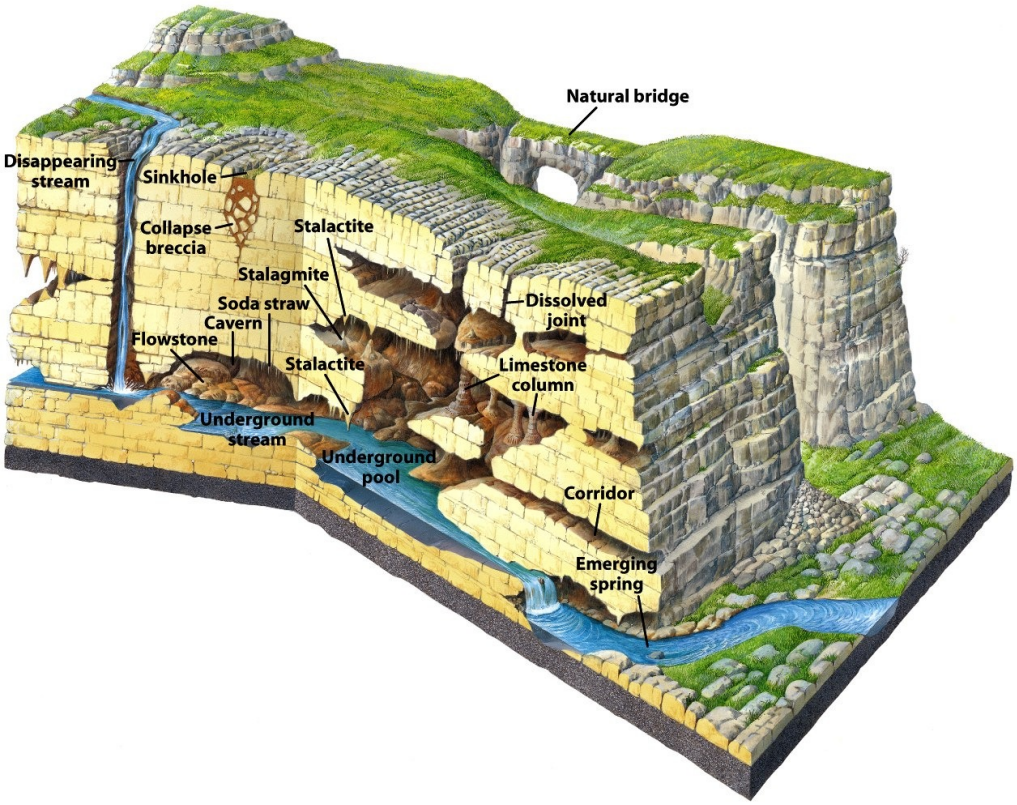

Figure 2.1: Karst landscape showing various features of karst aquifers. Figure from book by Marshak 2007

2.3 Overview of the karstification process

The most important factor in karstification process is flow of solvent through an underground aquifer. Since bedrock is subject to geological processes, a net (sometimes called a matrix) of fractures of varying diameter and shape is present in it. This solvent, usually water, flows through this kind of net and reacts with rock in ways described later.

Such karst aquifer may be surrounded on its sides by an aquitard, a substance that is impenetrable to water.

Water that enters the system may come from precipitation, rivers or lakes. Inflow of water may happen through diffuse infiltration or point infiltration. Diffuse infiltration happens on larger areas, covered by small fractures whereas point infiltration requires a prominent frac-

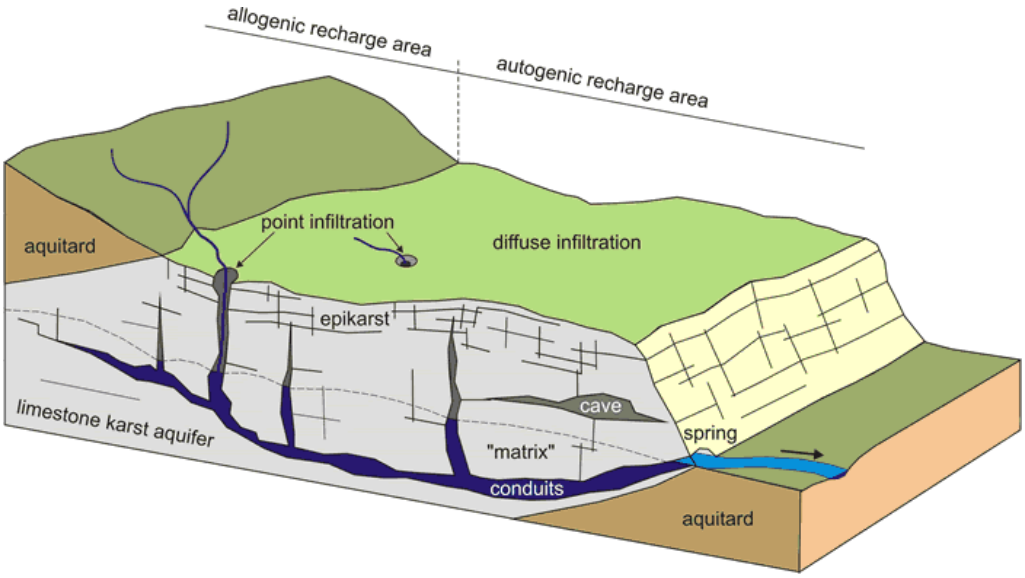

Figure 2.2: Example of limestone karst aquifer. Figure from Golscheider and Drew 2007

ture to be present in the aquifer that can take large amounts of water from a river or lake.

Recharge water that comes to the karst aquifer from neighbouring non-karst areas is called *allogenic recharge*. For example in figure 2.2 a river flowing on the upper aquitard and entering the limestone aquifer through the sinkhole is an allogenic recharge.

On the other hand, recharge water that flows directly to the karst area e.g. by precipitation is called an *autogenic recharge*

Water that flows through the aquifer reacts chemically with walls of the fractures widening them through dissolution. After going through the aquifer, water reemerges at lower level through emerging springs or resurgences.

There are also cases of ground formation resulting from human activity. Although not truly a karst process, a sinkhole that opened in 2010 in city of Guatemala was a result of combination of loose ground made of volcanic ash and inadequate draining system, that couldn't dissipate large amounts of water brought by tropical storm Agatha (Fletcher 2010).

2.4 Limestone dissolution

Chemical reactions that take place during the karstification process will be shown for limestone aquifers. Following description is taken form Dreybrodt and Gabrovšek 2002 which is based on Plummer, Wigley, and Parkhurst 1978.

With the pH of solution at about 7, limestone dissolves through the following slow reaction:

$$H_2O + CaCO_3 \longleftrightarrow Ca^2 + CO_3^{2-} + H_2O \tag{2.1}$$

Unfortunately, given very weak soubility of calcium carbonate in water[1], limestone dissolution would be extremely slow.

However, as the rain passes throught the atmosphere, it's picking up carbon dioxide that gets dissolved in water. Also, top layer of an aquifer, called *epikarst*, is usually covered by soil that is rich in substances that contribute with more carbon dioxide. With this aquired CO_2, small amounts of carbonic acid are produced:

$$H_2O + CO_2 \longrightarrow H^+ + HCO_3^- \tag{2.2}$$

Above reaction delivers a proton that bonds with carbonate detached during slow dissolution 2.1:

$$CO_3^{2-} + H^+ \longrightarrow HCO_3^- \tag{2.3}$$

Thanks to this, the ion activity product (CO_3^{2-}) (Ca^{2+}) is below the solubility constant of calcite. Calcium bicarbonate[2] produced during this process has orders of magnitude better soulibility in water[3] what greatly enhances the rate of limestone dissolution.

Equations (2.1) to (2.3) can be summed up with the following single equation:

$$CaCO_3 + H_2O + CO_2 \longrightarrow Ca^{2+} + 2\,HCO_3^- \tag{2.4}$$

[1] Only about 0.0013 g/100 mL in 25°C according to Aylward and Findlay 2008

[2] $Ca(HCO_3)_2$

[3] 16.6 g/100 mL in 20°C according to Aylward and Findlay 2008

2.5 Formation of speleothems

Speleothems are mineral deposits of various kind that form in a process reverse to dissolution. When water rich in calcium bicarbonate leaves small fissures in the aquifer and enters big, hollow areas, its pressure lowers and causes CO_2–degassing that is the major factor in precipitation of calcium carbonate on the surfaces of caves.

Loss of carbon dioxide leads to supersaturation in the solution reaction that is directly reverse to reaction 2.4 (Fairchild and Baker 2012):

$$Ca^{2+} + 2\,HCO_3^- \longrightarrow CaCO_3 + H_2O + CO_2 \tag{2.5}$$

Water evaporation plays marginal role in speleothem formation, what was shown by Holland, Kirsipu, and Oxburgh 1964.

Chapter 3

Related work

This chapter presents references to research work in domain of modelling processes governing evolution of karst aquifers along with small summaries. This description will be helpful in explaining design decisions made in programming project, especially in area of input data formats.

Some visualisation techniques used previously for caved terrains in various contexts will also be presented.

3.1 Modelling karst aquifers

As karst aquifers contain network of fractures (see chapter 2) a simulation of flow and chemical reactions in single fracture is the basic building block of presented models.

These fractures are connected to form larger, two or three–dimensional networks, that represent whole conduits.

First attempts to describe processes taking place during evolution of karst aquifers with numerical models took place in early 1980's (Hiller 2013, p. 3). Buhmann and Dreybrodt 1985a developed numerical model for ternary chemical system ($CaCO_3$–CO_2–H_2O) in open systems (where CO_2 is exchanged with atmosphere) and for closed ones (Buhmann and Dreybrodt 1985b).

Single fracture simulation

With these dissolution models in place, several models of single conduit simulation were presented (Hiller 2013, p. 4).

Single fracture is modelled as a circular conduit in the intersection of fissure and bedding plane (Kaufmann 2009) or as a space between two parallel walls of rock separated by aperture of some width (Dreybrodt and Gabrovšek 2002).

Two–dimensional simulations

Single conduit one–dimensional models were later expanded into second dimension by combining set of conduits into a connected network (Hiller 2013, pp. 4–5). In such networks, fractures are organized into uniform, regular structure.

Three–dimensional simulations

With more powerful computational resources, researchers started to look into three–dimensional models that could finally provide insight into evolution of real–life karst aquifers. Such models were proposed by Annable 2003; Kaufmann 2003; Kaufmann, Romanov, and Hiller 2010.

Work by Hiller 2013 summarizes current state of three–dimensional models. His thesis contains overview of modelling techniques and approaches. Simulation of real–live karst aquifer near dam–site is presented that matches observations. This shows validity of proposed models.

These three–dimensional models also store data as a uniform grid of fractures. This fact heavily influenced the design of data structures used throughout programming part of this thesis described in chapter 6.

3.2 Visualisation techniques

Rendering techniques that touched the issue of cave rendering never tried to provide both physical accuracy and visually appealing graphics.

In Geiss 2007 a method for rendering procedural terrains is described that in some circumstances can generate caved structures. Similar to programming project of this thesis, presented method uses Marching Cubes algorithm (see chapter 5) implemented on GPU[1]. Volume data is generated through carefully crafted density functions based on noise sampling and various scalar functions that are easily computable in shaders.

[1] Albeit in shaders, not with any GPGPU solution

Forstmann and Ohya 2005 proposed method of on–the–fly rendering of procedural terrains that may also contain caves. It uses hierarchy of nested clip–boxes for LOD[2] calculation to reduce workload of the GPU.

Hiller 2013 developed *KARSTTOOL* – a MATLAB application that is capable of plotting various parameters of three–dimensional fracture net computed by running karst evolution simulation with *KARSTAQUIFER* simulator introduced in Kaufmann 2009. This program doesn't strive to provide visually rich renders; it's aimed at researchers that want to visualise various parameters of simulated karst aquifer like CO_2 or HCO_3 concentration. Screenshot of KARSTOOL application is presented below.

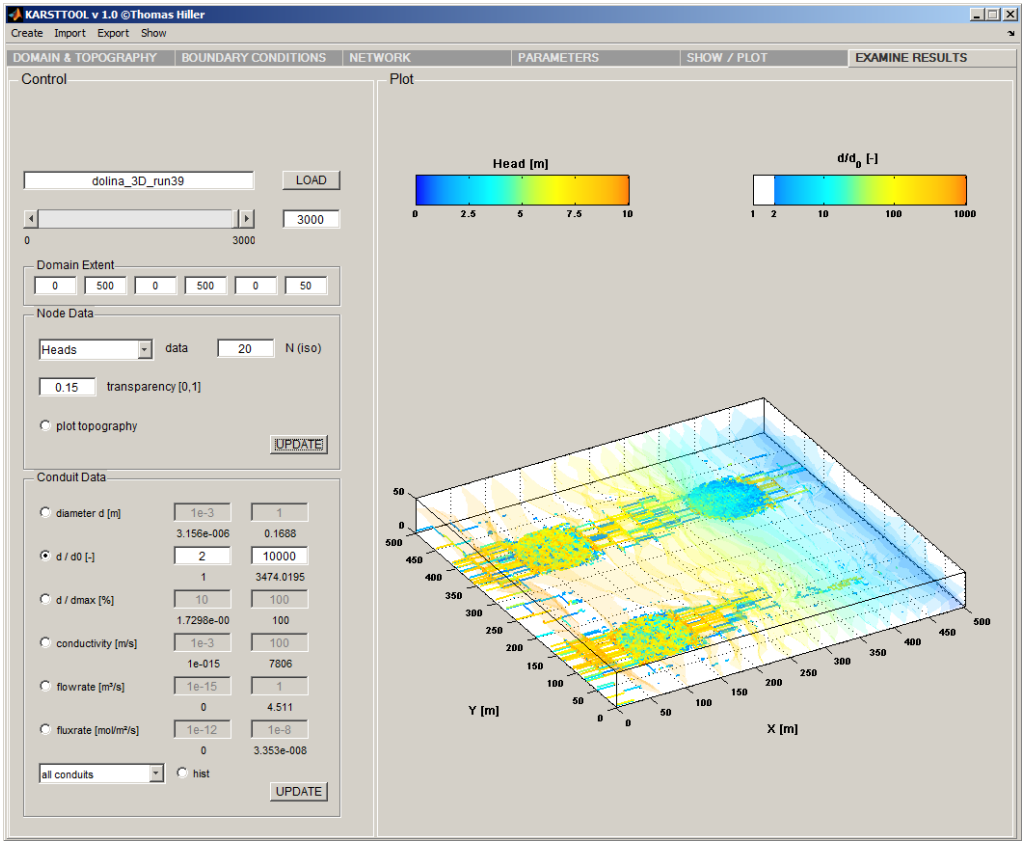

Figure 3.1: Screenshot of KARSTTOOL application. Results tab is shown that presents visualisation of simulation.

[2]Level of detail

Surveying software

Research around cave rendering is also done for cave surveying software. These specialized programs allow speleologists to capture data about structure of caves by manual or device–assisted measurement during exploration and plot the gathered data. Examples of open–source programs of this kind are *Therion*[3] and *Survex*[4]. More comprehensive list, albeit outdated, is available at British Cave Research Association Cave Surveying Special Group (http://csg.bcra.org.uk/software.html). Screenshot of 3D plot of graphic survey data from Therion is presented below.

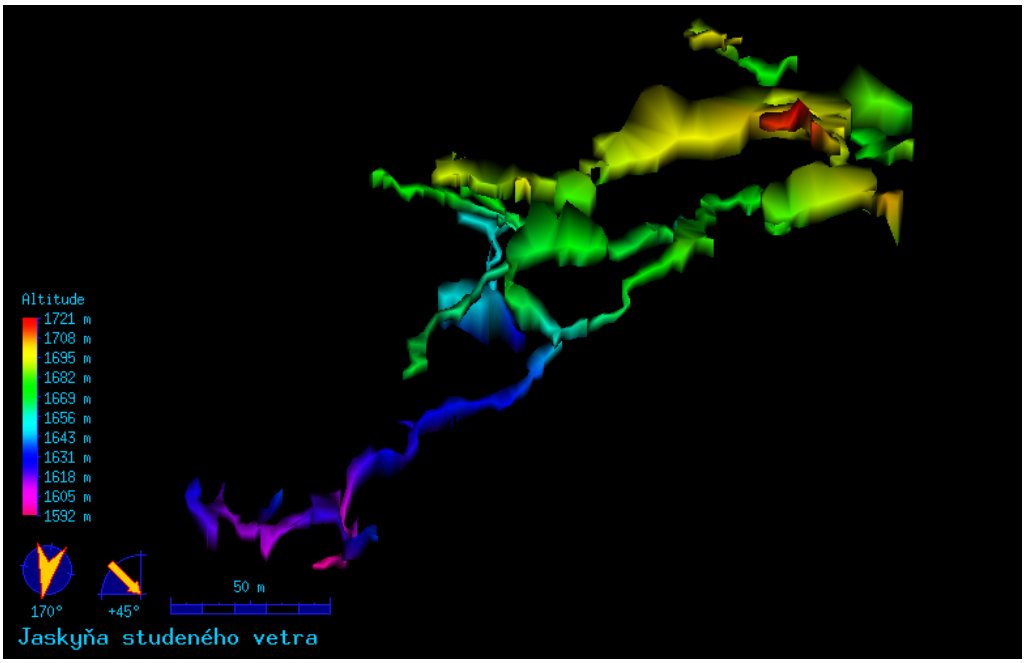

Figure 3.2: 3D visualisation of cave survey data from Therion application

[3] http://therion.speleo.sk/index.php
[4] http://www.survex.com

Chapter 4

OpenCL heterogenous programming platform

This chapter will describe OpenCL. A framework for writing applications that utilize computational capabilities of heterogeneous hardware environments offered by modern computers or embedded systems. Since the introduction of programmable pipeline in GPUs[1] an opportunity appeared to use capabilities of these devices to offload highly parallel, computationally intensive tasks from the main CPU[2]. OpenCL isn't however limited to GPUs. Multi–core processors can also be used through its runtime without manual thread management.

OpenCL is used in programming project of this thesis described in chapter 6. Metaballs and Marching Cubes algorithm are implemented with it. Marching Cubes algorithm and its OpenCL implementation are described in detail in chapter 5.

4.1 Introduction

This section is based on Kirk and Hwu 2010 and Gaster et al. 2012.

Beginnings of programmable GPUs

From early 1980s to late 1990s most graphic hardware was fixed–function. Dedicated graphics units exposed predefined, fixed set of functions that

[1] Graphics Processing Unit
[2] Central Processing Unit

were implemented in hardware or drivers. It wasn't possible to write custom code that would be executed on the GPU.

As the complexity of fixed–function APIs expanded, hardware vendors implemented them with general purpose processors that could run some limited instruction set on many execution units. This instruction set was used to implement graphics APIs like OpenGL or DirectX.

In 2001 NVIDIA released GeForce 3 graphics card that exposed this internal instruction set to users of OpenGL and DirectX APIs. ATI technologies followed with Radeon 9700 that could also run programs supplied by the user on the GPU. DirectX 8 and OpenGL introduced programmable vertex stage. With DirectX 9 another programmable stage was introduced, the *pixel shader*[3]. At this point, vertex and pixel shaders were implemented via separate chips in the GPU. In 2005, with release of XBox 360 first unified architecture was introduced, on which vertex and fragment shaders were run on the same processor.

Graphics processing, that these devices were build for, is very well suited for parallelization. Vertex shader stage takes list of vertices as its input and maps them onto the screen optionally defining colour of the vertex. Each vertex is processed independently making it possible to process many vertices at the same time.

Pixel shader stage receives position of the point and returns final colour of the pixel. This also is done independently for each pixel.

Early attempts at GPGPU[4]

With the unification of computational resources in the GPUs they started to resemble highly parallel computers. Researchers noted this fact, and tried to harness enormous parallel performance of these devices for workloads other than graphics.

GPUs of DirectX 9 era were still designed in graphics processing in mind. Although there were programmable stages in the pipeline, types of input and output parameters in each stage were severely limited. Moreover, the final result was generated as a pixel buffer, so the programmer had to map outputs of his algorithm to 2D screen space with pixel colour as output. Inputs to the pixel shader stage had to be supplied by textures.

[3] Or *fragment program* in OpenGL terminology

[4] General Purpose GPU

Even with these issues, researchers who managed to port their algorithms to GPUs reported great performance benefits (Kipfer and Westermann 2005).

CUDA

When working on Tesla GPU architecture, engineers at NVIDIA realized the potential in providing device's resources in more approachable way. Additional instructions and functionalities were added to the device. Among them, read and write operation with arbitrary offsets[5], synchronization barriers for groups of threads, atomic read/write operations. New parallel programming model was developed that defined hierarchy of threads.

To expose all these features to programmers new C–like language was created and named CUDA[6].

CUDA is capable of operating without any DirectX or OpenGL context. Device it's run on doesn't even have to be connected to any display output. CUDA programs are usually inlined in larger C or C++ programs and are called *kernels*. Special compiler called *nvcc* is used to compile kernel code. For more information about CUDA refer to official CUDA website[7].

Inception of OpenCL

On June 16th, 2008 Khronos Group announced formation of Compute Working Group (CWG) that was tasked with establishing open standard for programming heterogeneous CPU and GPU environments[8]. CWG consisted of many hardware and software vendors interested in standardization of such API.

Compute Working Group adopted proposal of Apple Inc. that submitted programming interface called Open Compute Language (OpenCL). Apple was already developing OpenCL for quite some time to have it ready for its upcoming Mac OS X Snow Leopard release.

[5]Shaders could only write to predestined places in memory, reserved for pixel output

[6]Compute Unified Device Architecture

[7]http://www.nvidia.com/object/cuda_home_new.html

[8]https://www.khronos.org/news/press/khronos_launches_heterogeneous_computing_initiative

On December 9th, 2008 final specification of OpenCL 1.0 was released[9]. Releases of conforming implementations from hardware vendors followed[10].

Specification

OpenCL specification is maintained by the Khronos Group. It consists of C API and Kernel language that is similar to C99. Khronos also releases official C++ wrapper API[11].

Unofficial bindings for various languages and frameworks also exist. Among others:

- PyOpencl[12] for Python
- JOCL[13] for Java.
- fortrancl[14] for Fortran
- gocl[15] wrapper for C applications based on GObject
- QtOpenCL[16] wrapper based on Qt library semantics.

4.2 Logical abstraction of computational resources

OpenCL aims to be API that lets hardware manufacturers expose various kinds of devices to the programmers in a consistent and abstracted way. To fulfil this requirement OpenGL defines logical hierarchy of computational resources. Hardware vendors map real hardware to this abstraction in their implementations of OpenCL.

[9] https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification

[10] http://www.khronos.org/conformance/adopters/conformant-products/#opencl

[11] This wrapper API is used in programming project of this thesis

[12] http://mathema.tician.de/software/pyopencl

[13] http://www.jocl.org/

[14] http://code.google.com/p/fortrancl/

[15] https://github.com/elima/gocl

[16] http://doc.qt.digia.com/opencl-snapshot/index.html

OpenCL defines one processor (*host*) that is coordinating execution of OpenCL kernels on one or more *devices*. Host is executing functions from C API portion of the specification. It's responsible for discovering available devices, setting up contexts for them, allocating memory on host and devices, queuing execution of kernels and initiating transfer of data between various memories.

Diagram of this logical structure is presented in Figure 4.1.

Platforms

On the top of the hierarchy there is a *Platform*. It is usually an implementation of OpenCL by a single vendor. To query list of platforms available in the system function `clGetPlatformIds()` must be called twice. Once with parameter `platforms` set to `NULL` to obtain number of platforms, and second time, with `platform` parameter set to array that will fit number of `cl_platform_id` structures equal or greater than the number in argument `num_platforms` retrieved on first invocation of this function[17].

Note however, that OpenCL is usually loaded as a dynamic library provided by vendor. These libraries will usually return only one platform.

To address this problem Khronos Group introduced `cl_khr_icd` extension that will look for list of installed OpenCL ICDs or Installable Client Drivers in place specific for given OS. If implementation supports this extension, new function `clIcdGetPlatformIDsKHR()` is available that will present to the user platforms from all vendors available on the system. This extension also makes sure, that function calls with OpenCL object created in certain platform will be routed to implementations in this platform.

Devices

Platform may contain one or more *devices*. Devices are units that actually execute the kernel code. Device may map for example to single GPU or CPU.

NVIDIA OpenCL implementation presents each GPU available in the system as separate device. OpenCL from AMD besides presenting GPUs also presents supported CPUs as devices.

[17]This pattern should be used for discovering other types of resources in OpenCL as well

Device is the last entity in hierarchy that has its distinct API object. Further elements cannot be operated on, and are just abstract concepts to which vendors map their hardware.

These elements are *compute units* which are comprised of *processing elements*. Devices can be queried for number of compute units they contain through `clGetDeviceInfo()` function.

Exact details of mapping are dependant on OpenCL vendor. Underlying architectures of GPUs, CPUs and DSPs[18] can differ greatly even within the same class of devices.

Specifics of implementation on selected devices supporting OpenCL will be presented in section 4.5.

4.3 Memory model

OpenCL provides layer of abstraction on device memory. Depending on the capabilities of given device, some of the memories described below may be unavailable.

Below are descriptions of memories defined by OpenCL specification. They are divided to Host–side and Device–side memories.

[18]Digital Singal Processors

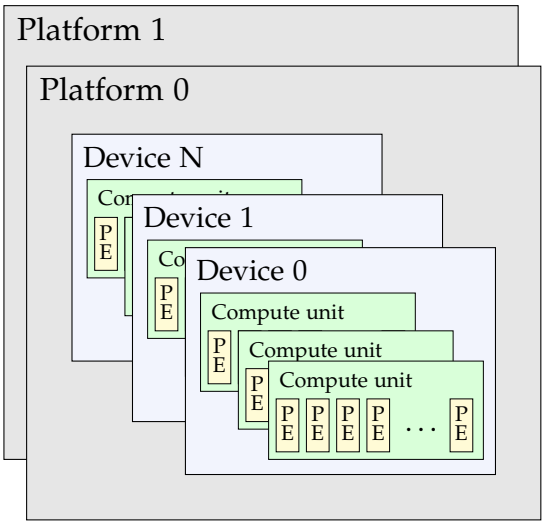

Figure 4.1: Logical partitioning of hardware in OpenCL

Host–side memory model

This is the model from the perspective of code that runs on the host. The one that executes OpenCL library functions. Three types of memory are distinguishable in this context:

Host memory
: Memory allocated by the host code with standard allocation techniques like `malloc()`. It's not managed by the OpenCL runtime.

Buffers
: Objects that are handles for global or constant memory allocated on host. Buffers are similar in nature to flat C arrays. They have linear addressing, so pointer arithmetic is possible in kernel code.

  Buffers differ however in one crucial point with host memory. Operations on them are asynchronous. Functions like `clEnqueueReadBuffer()` that transfer data between host and device return immediately, and the transfer starts in the background. Event mechanism may be used for synchronization (see section 4.4).

Images
: Images are somewhat similar to buffers, but differ in several ways. Unlike buffers, images can be multidimensional. They are not laid flatly in memory, but in a way that preserves spatial locality of points within the image[19], so they cannot be adressed directly but through sampling objects, that define access patterns to images. They are also limited in terms of supported datatypes to ones that are relevant in graphics; no arbitrary structures can be held in images.

  Images are abstraction of texturing units available in GPU shaders. Because of this heritage, images are not supported by every OpenCL implementation.

[19] For example by Z–order mapping (Gaster et al. 2012, p. 111-113)

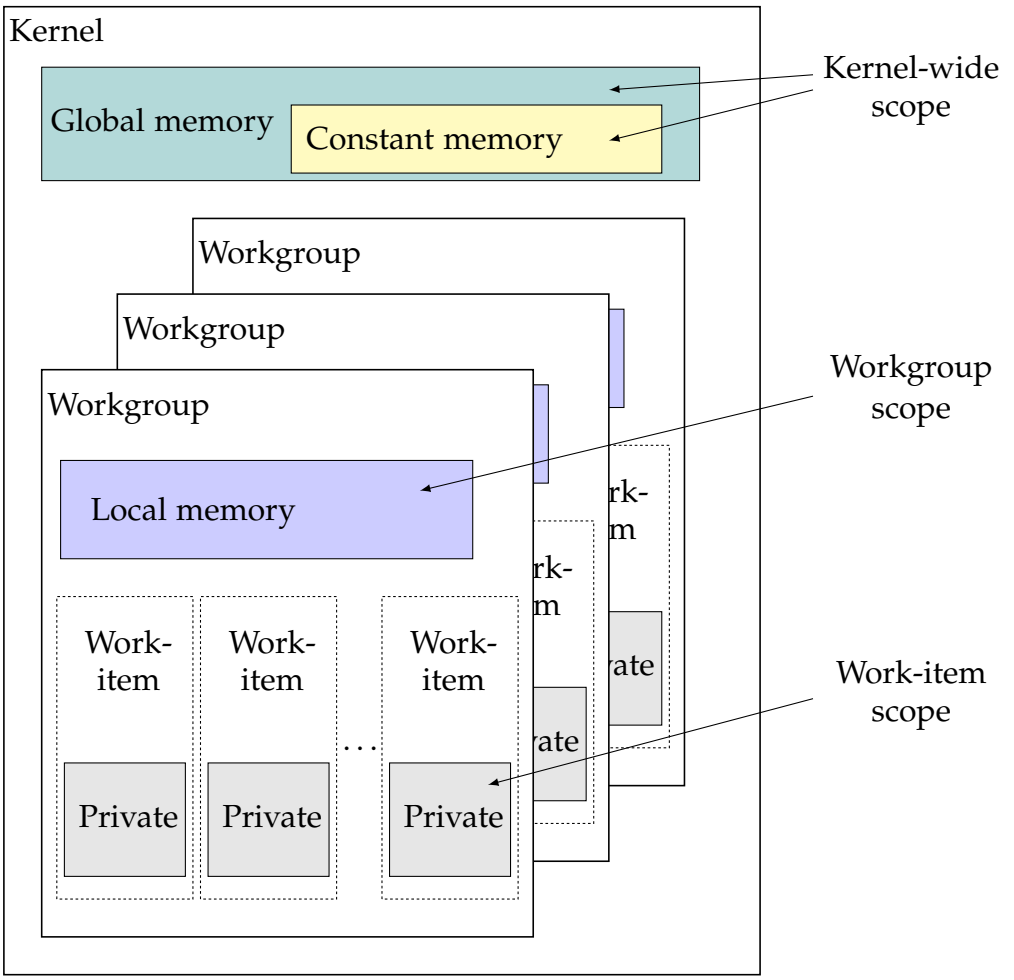

Figure 4.2: Device–side abstract memory model of OpenCL (Gaster et al. 2012)

Device–side memory model

These are OpenCL memory types, as seen from kernel perspective.

Global memory
: Kernel-wide memory that is visible to all compute units on the device. It is the only memory that can be used for transferring data between host and devices. This is also usually the slowest type of memory on the device by raw throughput.

Constant memory
: Memory designated for data that is going to be accessed simultaneously by many threads. Its contents cannot change throughout the lifetime of the kernel. If available it's usually implemented with specialized hardware and/or caching strategies.

Local memory
: Special memory area that resembles software–controlled cache. This area is valid only within a single workgroup (see section 4.4). It's expected to be much faster than global memory[20], so it can be used as a scratchpad area for threads in one workgroup.

  It may be implemented e.g. as separate chip in Compute Unit.

  Local memory can be declared for workgroup in two ways, either dynamically, by setting kernel parameter prefixed with `__local` to NULL with desired size parameter, or statically as an array in kernel with the same prefix.

Private memory
: Memory valid within a single thread. Every non-prefixed variable and all function arguments that are not pointers land in this memory. It may be implemented with registers[21]. If number of registers is exceeded, they may be spilled to global memory, what can have very detrimental impact on performance. Number of registers used by kernel must be thus carefully controlled.

  Private memory is the fastest available type of memory. For GPUs it's usually orders of magnitude faster than global or local memory.

4.4 Execution model

This section will describe OpenCL objects that manage execution of programs on the device and stages of such execution.

Context

Context is a structure that coordinates communication between host and devices and keeps information about memory objects.

[20] However it's not guaranteed

[21] Execution on GPUs isn't stack–based, so every variable is stored in very limited number of registers

It is created by providing list of devices to `clCreateContext()` function. Devices must all come from the same platform, as returned by `clGetDeviceIDs()`. There is a convenience function `clCreateContextFromType()` that creates context from all devices of given type from one platform (Munshi 2012).

Programs and Kernels

To execute kernel code on the device, it must be first provided to specialized compiler that translates human–readable source code to machine–specific instructions.

Kernel code must be fed to `clCreateProgramWithSource()` function in the form of pointer to character array. OpenCL implementation takes this textual representation and translates it in runtime. It's worth noting, that OpenCL kernel code is provided to the library in the source form. This may be not suitable for secret proprietary algorithms. For this reason, with OpenCL 2.0 Provisional specification Khronos released definition of common intermediate representation called SPIR[22].

There is another function that creates program objects named `clCreateProgramWithBinary()`. It takes binary representation of OpenCL code and creates program object from it. This function however, is highly device–specific. It can only be used for caching compiled programs the first time application is run so no compilation is needed during subsequent runs.

After creating program object, it isn't yet compiled. Compilation is triggered by `clBuildProgram()` function.

One program object may have been compiled from a source string containing many OpenCL kernels. Each function prefixed with `__kernel` in such string may be used to create separate kernel object with `clCreateKernel()` function by providing build program object and kernel function name, or with `clCreateKernelsInProgram()` function that creates kernel objects from all kernel functions present in program object.

Supplying arguments to kernels

Since kernels aren't normal host function, they aren't invoked as such. Because of that, arguments for kernels must be supplied in specific way.

[22]Standard Portable Intermediate Representation http://www.khronos.org/registry/cl/specs/spir_spec-1.2-provisional.pdf

Each argument must be set with separate call to function `clSetKernelArg()`. This style of argument setting resembles the way arguments are supplied to shaders in OpenGL or DirectX.

Command queues

With kernel arguments properly set up, it can be queued for execution on the device with `clEnqueueNDRangeKernel()`[23] function. Kernels are scheduled for execution on *command queues* — special objects that are tied to particular device and are used for management of tasks on this device.

There may be many command queues created with single device[24]. Conformant OpenCL implementation guarantees, that as long as these command queues don't use the same resources, like memories, programs and kernels, at the same time, no synchronization is needed. Otherwise, special care must be taken to ensure consistency. Using shared objects on many queues is described in Appendix A of the OpenCL specification (Munshi 2012).

Workgroups and threads

Exactly how many threads are started when kernel is enqueued in command queue is determined by *configuration* of the execution. Threads may be organized as regular 1D, 2D or 3D structure. Size of this structure for all threads is called *global work size*. This structure is further divided into smaller packets called *work-groups*. Exactly how task is divided into work–groups can be either specified by user or done implicitly by the implementation.

If workgroup size is specified explicitly, global work size must be evenly divisible by local work size in each dimension.

Threads within a work–group can be synchronized with barriers. Also, local memory is shared by every thread in the work–group.

OpenCL specification doesn't guarantee any order of execution of work–groups in a single kernel invocation. Particularly, execution of one work–group may be suspended while it waits for data from slow global memory, and another one, i.e. one ready to perform calculations, may start or resume execution.

[23] Enqueue N-dimensional range kernel

[24] For e.g. multiple application threads

Events and device–side relaxed consistency

Every OpenCL function that could possibly block, is called asynchronously. It schedules execution of operation in the command queue and immediately returns.

Synchronization of such tasks is done with *event objects* which are returned by all potentially blocking operations. Such objects can be either waited for with `clWaitForEvents()` or be passed to other blocking functions which in such case guarantee not to start their execution before all events passed to them finish.

OpenCL specification also defines *relaxed consistency* memory model (Gaster et al. 2012, pp. 114–115). Until the end of kernel execution, memory writes may not be visible to other work–items if fences are not used.

This gives following, three–point hierarchy of memory consistency:

- Memory operations are ordered predictably within single work–item. They won't be reordered by the compiler.
- For work–items within single work–group memory is guaranteed to be consistent only at barriers
- For work–items from different work–groups there is no consistency of memory guaranteed, there are however atomic integer operations on global memory available.

This relaxed model is needed to make it possible to implement OpenCL on wider variety of devices. Any stricter model would single out some class of devices.

Typical execution flow

Applications usually follow common pattern when offloading computation to OpenCL devices. There are two main ways this can be done, depending on whether results must be read back to the host memory or not.

When results of the computation must go back to the device, execution usually has the following steps:

1. Query platforms available on the system and choose one of them with `clGetPlatformIDs()` or `clIcdGetPlatformIDsKHR()`
2. Query devices available on chosen platform with `clGetDeviceIDs()`

3. Create context from selected devices with `clCreateContext()`
4. Create command queue for each device in the context with `clCreate-CommandQueue()`
5. Create program and kernel objects from kernel source code with `clCreateProgramWith[Source,Binary]()`, `clBuildProgram()` and `clCreateKernel()`
6. Create memory objects with input data and transfer it from host memory to the devices[25] with `clCreateBuffer()` and `clEnqueue-WriteBuffer()`
7. Set kernel parameters with `clSetKernelArg()`.
8. Enqueue kernel(s) execution with `clEnqueueNDRangeKernel()`
9. Read back data with `clEnqueueReadBuffer()`

Since OpenCL is quite often implemented with GPUs, it may be used to generate data later used for rendering. Results of computations of e.g. fluid simulation (Kolb, Latta, and Rezk-Salama 2004) or particle systems[26] don't have to be transferred back to the host memory; they are used for rendering frame of animation, and may be discarded, overwritten or used as input for next frame. For such use cases, two extensions were developed:

- `CL_KHR_gl_sharing` for sharing memory objects with OpenGL
- `cl_khr_d3d10_sharing` for sharing memory objects with Microsoft DirectX

With these extensions, round–trips between host and device memory are unnecessary. Above steps are thus a bit different, because data is not read back but is used as an input for rendering by graphics API.

[25] Actual transfer may not occur if `CL_MEM_ALLOC_HOST_PTR` flag is used when buffer is created and host and device share the same memory. This is possible for example on AMD hybrid APU (Accelerated Processing Unit) systems that share single memory

[26] http://software.intel.com/en-us/vcsource/samples/3d-fluid-simulation

4.5 Implementation on selected hardware

In this section, implementations of OpenCL on two different types of devices will be presented. One is an AMD Bulldozer CPU – an 8–core general purpose x86 processor and the other one is NVIDIA GTX580 – a high–end consumer GPU. High–level hardware architecture of both devices will be shown with mapping to OpenCL logical device hierarchy (see section 4.2). Some specific considerations that must be taken when designing OpenCL application for these devices will also be discussed.

OpenCL on AMD FX–8150 Bulldozer CPU

Bulldozer is a microarchitecture of AMD processors released on September 7, 2011[27]. Described processor is AMD FX–8150, a high–end consumer CPU with 8 x86 cores packed into 4 *modules*. Cores within single module share FPU unit, instruction decoding and fetching mechanisms, branch predictor, and 2MiB L2 data cache. Each core has its own 16KiB L1 data cache. AMD Implementation of the OpenCL runtime for CPUs is based on Gummaraju et al. 2010.

Entire processor is presented to the programmer as a single OpenCL device[28] By default, every core is used to perform computations. For each core, one thread for dispatching work–groups is created what effectively creates pool of threads with one thread pinned to each core.

Mapping to OpenCL logical hierarchy

If not divided into subdevices, each core of Bulldozer CPU is considered one OpenCL compute unit. Since CPU doesn't have any controllable cache, RAM is used for all kinds of OpenCL memories. However, special measures are taken, to ensure that private memory of work–items and local memory of work–groups is laid in a way that is optimal for cache locality.

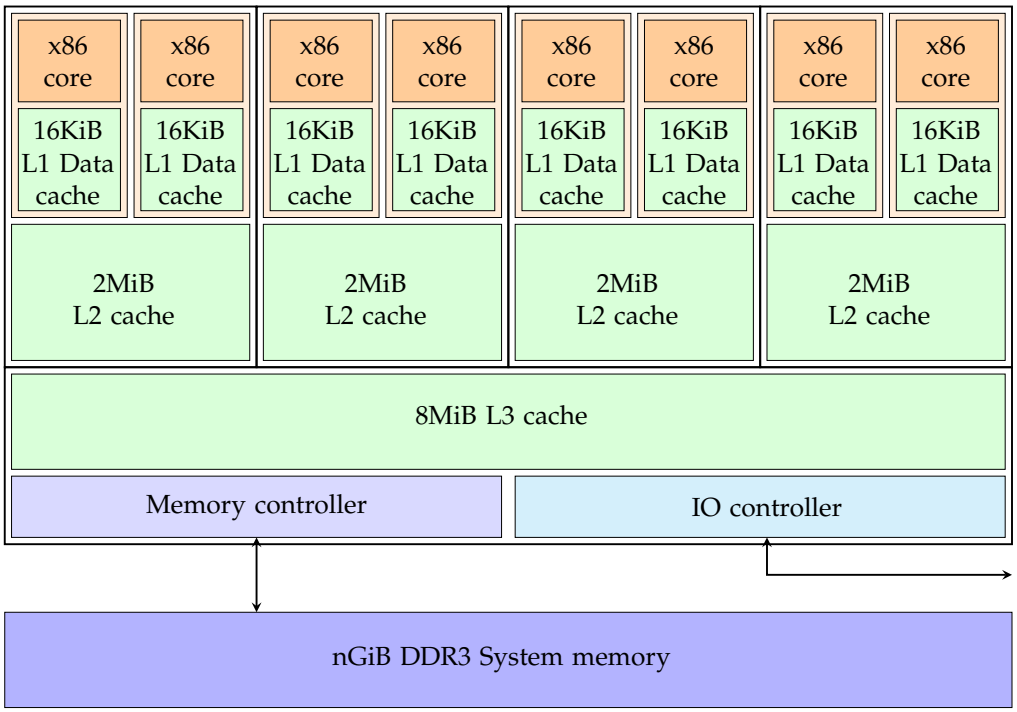

Figure 4.3: Block diagram of hardware architecture of AMD FX–8150 CPU. Device has 8 cores packed into 2–core logical modules sharing FPU unit, 2MiB of L2 data cache, instruction fetching/decoding units and branch predictor.

Execution model

Each core of the CPU executes work–groups assigned to it one after another. Within each work–group, each work–item is executed serially until barrier operation is reached or kernel is finished. When barriers are reached, work–item execution is suspended, its local state is saved through `setjmp` system call to be later restored with `longjmp`. Execution then moves to the next work–item within the work–group.

AMD OpenCL runtime provides modified version of `setjmp` and `longjmp` syscalls that work better with CPU branch predictor and main-

[27] http://www.amd.com/us/press-releases/Pages/amd-ships-bulldozer-processors-2011sep7.aspx

[28] However with *device fission* mechanism introduced in OpenCL 1.2 (and earlier with extension) it is possible to divide a device into subdevices based on e.g. memory characteristics

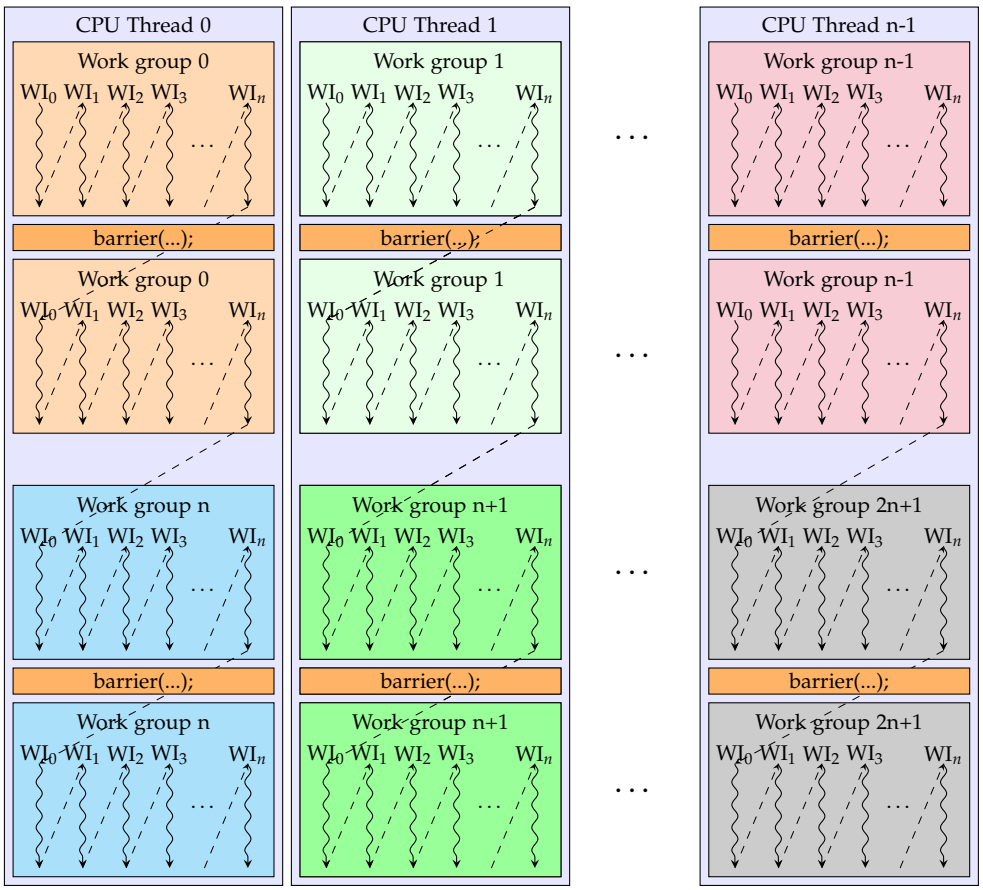

Figure 4.4: Execution model of OpenCL on AMD FX–8150 CPU.

tain program stack alignment. This approach is faster than system–level thread preemption because creating system threads is much more expensive than aforementioned calls.

Within each work–group vector operations from SSE[29] and AVX[30] extensions are heavily used to obtain instruction–level parallelism on vector types offered by OpenCL language specification.

Worth mentioning is the fact, that main program code is executed on the same processor as OpenCL kernels. Heavy calculations in the host part of the application may severely decrease performance of OpenCL computations.

[29] Streaming SIMD Extension
[30] Advanced Vector Extension

OpenCL on NVIDIA GTX580

NVIDA GeForce GTX 580 is the most advanced graphic card in GeForce series 500 lineup. It's a consumer–oriented GPU aimed at users with highest performance needs. GTX 580 is based on architecture called *Fermi* and will be presented based on Gaster et al. 2012, pp. 59–61 and whitepaper *NVIDIA's Next Generation CUDA Compute Architecture: Fermi* 2009.

Architecture

GTX 580 contains 16 Compute Units called *Symmetric Multiprocessors* (SMs). Each SM contains following elements (for graphical diagram see Figure 4.5):

32 *CUDA Cores*
: Compute units containing ALU[31] for integer arithmetic and FPU[32] for floating point operations conforming to IEEE 754–2008 standard. It also supports multiply–add and fused multiply–add operations.

16 load/store units
: These units calculate addresses for memory access. They provide access to data for 16 threads per clock cycle.

4 SFUs
: Special Function Units for execution of transcendental instructions like sine, cosine, reciprocal and square root

2 Warp Schedulers
: Thread scheduling units that choose 2 packs of 16 threads for concurrent execution.

Register file
: Area of very fast memory, 128KiB in size[33]

Shared memory/L1 cache
: 64 KiB of fast memory configurable as 16KiB of shared memory

[31] Arithmetic Logic Unit
[32] Floating Point unit
[33] Or 2^{15} 32-bit elements

and 48KiB of L1 cache or 48KiB of L1 cache and 16 KiB of shared memory

These 16 SMs are accompanied by following units in the chip:

- 6 64-bit GDDR5 memory units forming 384-bit memory interface that supports up to 6GiB of DRAM, 768KiB of L2 cache
- GigaThread engine for distributing blocks[34] among SMs
- host communication interface[35].

Mapping to OpenCL logical hierarchy

Single Fermi GPU is visible as one OpenCL device. SMs are equivalent to Compute Units, and CUDA cores are mapped to Processing Elements. GDDR5 DRAM implements global memory, per–SM shared memory is used for OpenCL local memory, register file is used for private memory, and constant memory is located on dedicated DRAM units with separate cache. Images are implemented using texture memory, the same one that is used for graphics processing with DirectX and OpenGL.

Execution model

When kernel is scheduled for execution on Fermi GPU it's enqueued in GigaThread engine that distributes work–groups among available SMs.

When work–group arrives at the SM, it's further divided into units of 32 threads called *warps*. At one cycle, two warps are selected that go to each Warp Scheduler. Both Warp Schedulers issue one instruction from each warp to one of following parts of SM:

- 16 CUDA cores[36]
- 16 Load/Store units
- 4 SFUs

Most of instructions in both warps can be issued independently, so this dual–scheduling technique helps in achieving peak hardware utilization.

[34] Or work–groups in OpenCL nomenclature
[35] PCI-Express bus
[36] In this case, these 16 threads form a half–warp

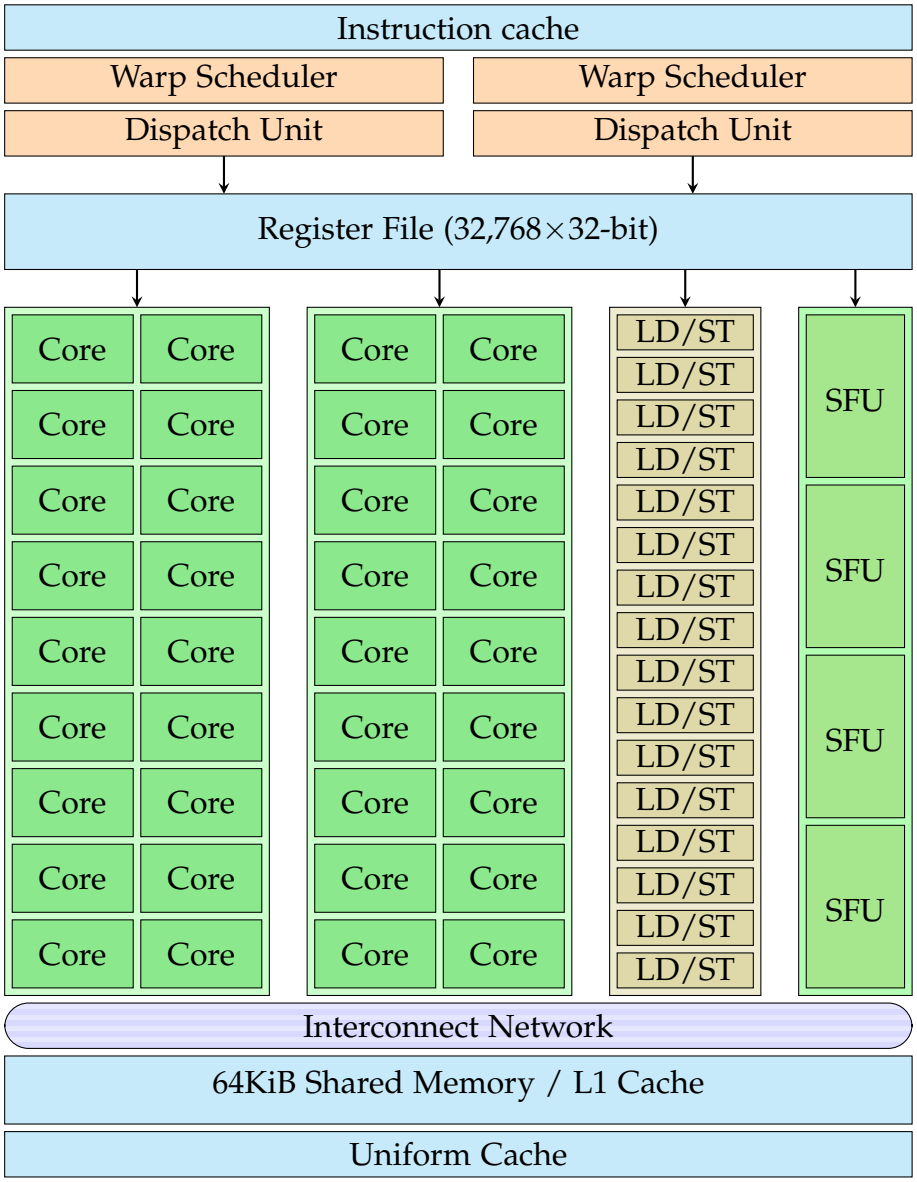

Figure 4.5: High–level overview of elements forming Fermi SM

Pitfalls of OpenCL programming on GPU

Execution model described in previous section clearly prefers execution of the same instruction on many threads. NVIDIA calls this model SIMT or „Single Instruction Multiple Thread“. When just one thread diverges in code path through e.g. `if` statement, every possible code path must be executed for each thread in a half–warp. That's why kernels with divergent code flows should be avoided.

Another problem is usage of registers. Since execution isn't stack–based like in x86 CPUs, every local variable must go to Register File on SM that is rather small. When too many registers are used, either less blocks will be able to be computed at the same time, or register spilling mechanism that uses orders of magnitude slower global memory as a backup location for overflowing variables will kick in. Both of these possibilities may significantly reduce performance.

As mentioned before, execution of threads on GPU isn't stack based. All function calls in kernels are inlined. Therefore, algorithms that rely on recursion with arbitrary depth may be impossible to implement without deep modifications.

Chapter 5

Isosurface extraction with Marching Cubes

This chapter will present a method of isosurface extraction that is later used in programming project. First, basic definitions will be established and rationale for generating graphics with volumetric data will be discussed. Next, brief history and high level overview of Marching Cubes will be presented. Technical details will follow with description of implementation on highly parallel GPGPU systems with OpenCL.

5.1 Definitions

Definition 1. *Density function* is a scalar function of the form $\mathbb{R}^3 \to \mathbb{R}$ or $\mathbb{R}^2 \to \mathbb{R}$ that defines value of some magnitude in a continuous space. An example of such function in 3D space is temperature defined in each point in the space. Height on a flat map on the other hand is a density function in 2D space.

Note that such defined density function doesn't have any connection to *probability density function*. It's in fact a special kind of *scalar field*. This definition however is commonly used in context of rendering isosurfaces (Geiss 2007).

Definition 2. *Isosurface* is a surface in three-dimensional space that consists of points that have the same value of *density function* called threshold value. Points in domain with density function value smaller than threshold value are considered to lie below the surface and points which density function value larger than threshold value are considered to

lie above the surface. Points with density function value equal to the threshold are considered to lie exactly on the isosurface.

Definition 3. *Isovalue* is a threshold value of density function that forms the *isosurface*.

5.2 Rationale for isosurface rendering

There are many applications which yield data as a density function. Some of them are listed below:

CT[1] scan. Result of such scan is a set of 2D slices with each slice consisting of array of scalar values (Lorensen and Cline 1987a).

Weather data Weather data, especially coming from weather models consists of scalar values of various parameters (temperature, humidity, etc.) on earth's surface

Arbitrary mathematical function It's often desirable to visualise mathematical function with multiple parameters on 2D and 3D plots. For example for educational purposes.

Procedural models Surfaces expressed by density function may be a source of visually interesting models that could be hard to model by hand.

Interactive presentation of such data may be very helpful while working with these applications. Ability to rotate, zoom and scale such surfaces is beneficial to understanding the data since human sight apparatus is naturally well equipped to process 3D objects and images.

5.3 Marching Cubes algorithm overview

History (Lorensen 2007)

Marching Cubes algorithm was invented in 1984 by William E. Lorensen and Harvey E. Cline. While being employed by General Electric they attended a seminar by GE's Medical Systems Business Group employee Carl Crawford. Mr Crawford described capabilities of the upcoming

[1] Computer Tomography

rendering engine called *Graphicon*, that rendered using polygons. He also challenged seminar attendees to find interesting usages for the device. Within a day Lorenson and Cline devised an algorithm that read volumetric medical data (essentially a density function) and produced triangle mesh representing isosurface.

General Electric submitted a patent application for the algorithm on June 5, 1985, which was granted on December 1, 1987 (Lorensen and Cline 1987b).

Partly due to existence of this patent, another algorithm called *Marching Tetrahedra* was invented to give graphics community another method of isosurface extraction, that is not encumbered by patents. *Marching Tetrahedra* also solves some ambiguities that are present in *Marching Cubes*.

Patent on *Marching Cubes* algorithm expired in 2005.

Algorithm description (Lorensen and Cline 1987a)

Marching cubes algorithm divides space on which it operates into a discrete lattice of cubes (interchangeably called *voxels*). For each cube, density function value is retrieved for each vertex of the cube. Density function may be calculated from the position of the vertex on the fly if it's defined as a mathematical function, or it may be extracted from some external volumetric data source (e.g. result of CT scan).

Next, for each vertex of the cube it's determined whether value at its position is larger or smaller than requested isovalue. If the value on the vertex is smaller vertex is below the surface. Otherwise it's above it.

Being above or below the surface will be called the *sign* of the vertex. If vertices on the ends of given cube's edge are of different signs, than it's certain that the surface crosses the edge.

For each cube, there are $2^8 = 256$ possible combinations of sings of the vertices. Combination of these signs is called the *index* of this cube (see Figure 5.2).

When the index of the cube is known, pre-generated LUTs[2] are consulted to determine how many polygons and in what configuration should be emitted for this cube.

Original version of Marching Cubes algorithm divides all 256 possible combinations of vertices into 15 cases. These cases are presented

[2]Look-Up Tables

in Figure 5.1. Remaining combinations are derived from these cases through applying symmetries, rotations, and switching all signs of cube verices.

Another LUT is consulted that maps cube index to edge numbers on which generated vertices will lie. Vertex is then emitted for each edge that is crossed by the isosurface.

Process is repeated for all cubes in the lattice and emitted polygons (possibly with normal vectors for lighting) are the output of the algorithm.

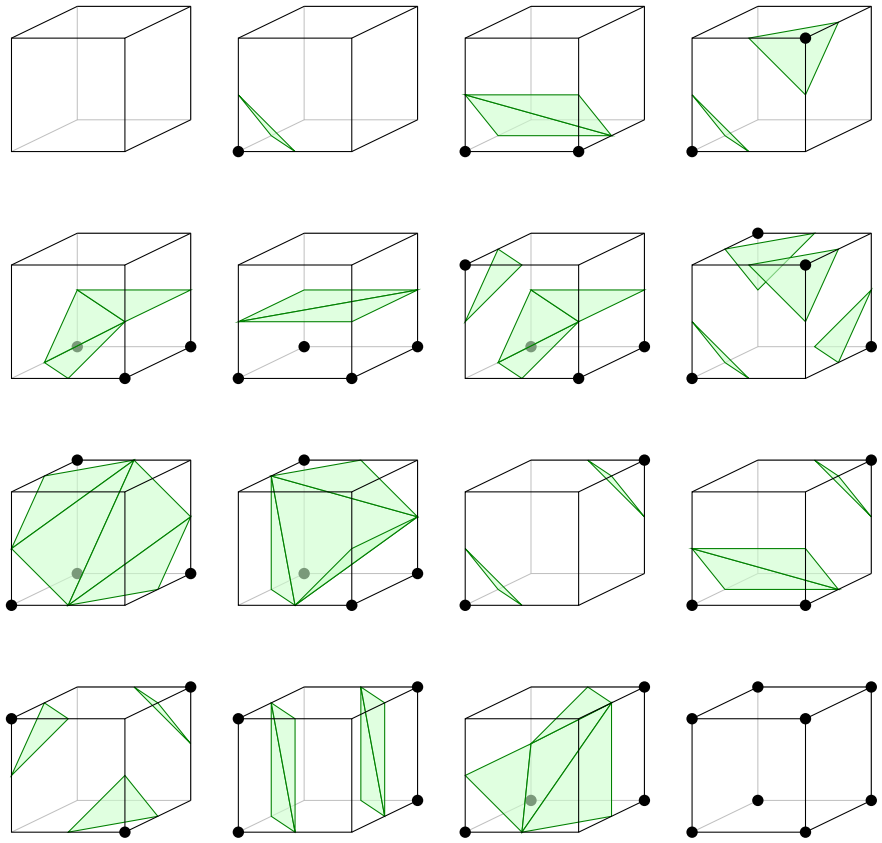

Figure 5.1: All cases in traditional Marching cubes algorithm. Vertices with density function above threshold value have black circles on them. Symmetries, rotations, and complementary cases (with exception of cases 0 and 255) were omitted for brevity.

Cube indexing

Operation on a single cube begins with evaluating density function on each vertex of the cube.

Index of the cube is calculated through operation described under Figure 5.2.

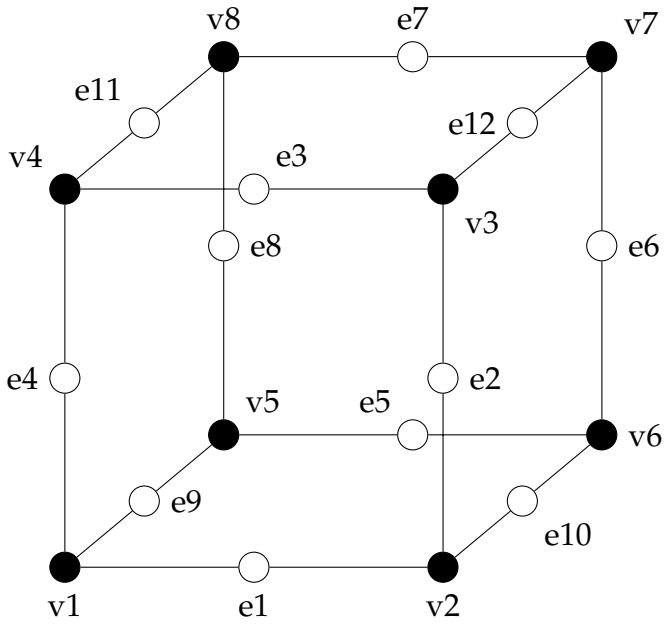

Figure 5.2: Numbering of vertices and edges in Marching Cubes. Cube index is derived by concatenation of bits: `index = v8|v7|v6|v5|v4|v3|v2|v1` where each v*i* is logical result (0 or 1) of operation of comparing density function value at *i*-th vertex with threshold value (`value(i) > threshold`).

Emitting polygons

When index of the cube is known, LUT is consulted that maps index to list of edges on which vertex in given cube must be emitted.

Listing 5.1: Index to edge list LUT. Notice that for indices 0 and 255 no geometry is emitted

```
1 unsigned char mcTriangleTable[256][16] = {
        {255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
           255, 255, 255, 255, 255, 255},
```

```
3           {0, 8, 3, 255, 255, 255, 255, 255, 255, 255, 255,
               255, 255, 255, 255, 255},
            ...
5           {0, 3, 8, 255, 255, 255, 255, 255, 255, 255, 255,
               255, 255, 255, 255, 255},
            {255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
               255, 255, 255, 255, 255, 255}
7   };
```

For each edge on the list vertex is emitted between its two ends in place proportional to the linear interpolation of density function at the vertices.

Each three vertices form a polygon. In the listing above, value 255 marks an end of the list for given index.

Next, each polygon is saved in a list for later usage, or directly fed to rendering device.

5.4 Implementation on GPU with OpenCL

Following implementation is based on example code from NVIDIA CUDA SDK example[3].

As described in section 5.3 each cube is processed by the algorithm independently. This possibly makes this algorithm a good candidate for massive parallelization offered by general purpose GPU programming. There are however some obstacles to overcome.

First, LUTs used by the algorithm are obviously too big to fit into registers. Reading them from global memory would also be problematic, because they are accessed in a random manner, depending on cube index. Tables could be copied to local memory for faster access, but the cost of copying for each block could be substantial.

Another problem is storage of the output. It is not known beforehand how many polygons will be emitted by each cube. With sequential implementation it isn't a problem because data generated by each cube may be just appended to single result array. With many cubes being processed at the same time, this approach will not work.

[3] http://docs.nvidia.com/cuda/cuda-samples/index.html#marching-cubes-isosurfaces

Stages in GPU implementation

Kernel execution is divided into the following stages:

1. voxel classification
2. compacting
3. triangle generation

All operations are performed on cube lattice flattened to 1D array. Each stage of the execution will be described below.

Voxel classification

In this stage, all voxels are classified as to whether they will produce any geometry or not, and how many vertices is given voxel going to produce.

Results are written to two arrays: `voxelOccupied` which contains 1 if given voxel produces any geometry and 0 otherwise, and `voxelVerts` that holds number of vertices produced by this voxel:

```
__kernel
void classifyVoxel(
        __global uint *voxelVerts,
        __global uint *voxelOccupied,
        uint4 gridSize,
        float4 voxelSize,
        float isoValue,
        uint numVoxels,
        __read_only image2d_t numVertsTex)
{
        uint i = get_global_id(0);
        float4 cubeValues[8];
        /* Here, values on each vertex of the voxel are
           inserted into cubeValues array */

        int cubeIndex = getCubeIndex(cubeValues, isoValue);
        uint numVerts = read_imageui(numVertsTex,
           tableSampler, (int2)(cubeIndex, 0)).x;
        if (i < numVoxels) {
                voxelVerts[i] = numVerts;
                voxelOccupied[i] = (numVerts > 0);
        }
}
```

Note that additional lookup table (`numVertsTex`) that maps cube index to number of vertices it produces is used. This LUT is accessed as a texture.[4]

Index of the cube is calculated as described in Figure 5.2:

```
int getCubeIndex(float4 *cubeValues, float isoValue)
{
        int cubeIndex;
        cubeIndex =  (cubeValues[0].w < isoValue);
        cubeIndex += (cubeValues[1].w < isoValue) << 1;
        cubeIndex += (cubeValues[2].w < isoValue) << 2;
        /*...*/
        cubeIndex += (cubeValues[7].w < isoValue) << 7;
        return cubeIndex;
}
```

Compacting

To overcome problem highlighted in section 5.4 a special compacting operation is performed on arrays from previous step. First, a prefix-sum array is computed in parallel on the GPU[5] on `voxelOccupied` and

[4] All other LUTs are accessed this way as well. This partially mitigates one problem described in section 5.4

[5] Implementation based on Harris, Sengupta, and Owens 2007, as included in NVIDIA CUDA SDK

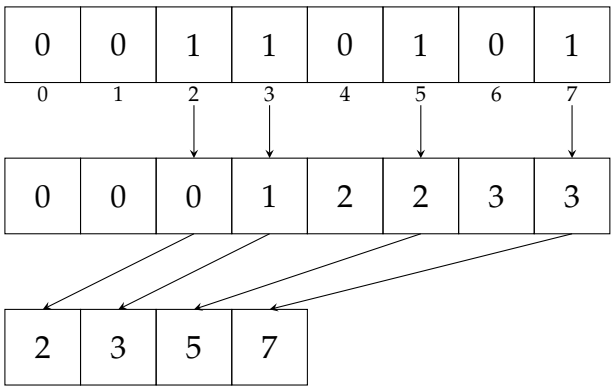

Figure 5.3: Example of compaction procedure. Index of each non empty voxel is saved in `compactedVoxelArray`. Place to which index should be stored is read from `voxelOccupiedScan` array.

`voxelVerts` arrays resulting in `voxelOccupiedScan` and `voxelVertsScan` arrays.

Definition 4 (Prefix-sum operation (Blelloch 1990))**.** Operation that takes binary associative operator $\oplus$ with identity I and an array of n elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the array $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

This operation is sometimes called *scan* operation.

By reading the last elements of `voxelOccupied` and `voxelOccupiedScan` and adding them, the number of voxels that will produce geometry can be obtained. These voxels are called *active voxels*.

Thanks to voxel compaction, the most computation–intensive kernel, i.e. `generateTriangles` kernel that generates final geometry can be computed only for voxels that are not empty. Since in many cases, space on which Marching Cubes is working is rather sparse, this approach can bring enormous performance gains.

Let call the number of active voxels n. Compaction kernel creates an array of length n that will contain indices of non empty voxels. For illustration of this process refer to Figure 5.3.

Below is the code that performs the compaction.

```
__kernel
void compactVoxels(
        __global uint *compactedVoxelArray,
        __global uint *voxelOccupied,
        __global uint *voxelOccupiedScan,
        uint numVoxels
)
{
        uint i = get_global_id(0);
        if(voxelOccupied[i] && (i < numVoxels)) {
                compactedVoxelArray[voxelOccupiedScan[i]] =
                    i;
        }
}
```

Scan operation on `voxelVerts` array gives `generateTriangles` kernel an offset to the result array where data for given voxel should be written. For an example refer to Figure 5.4.

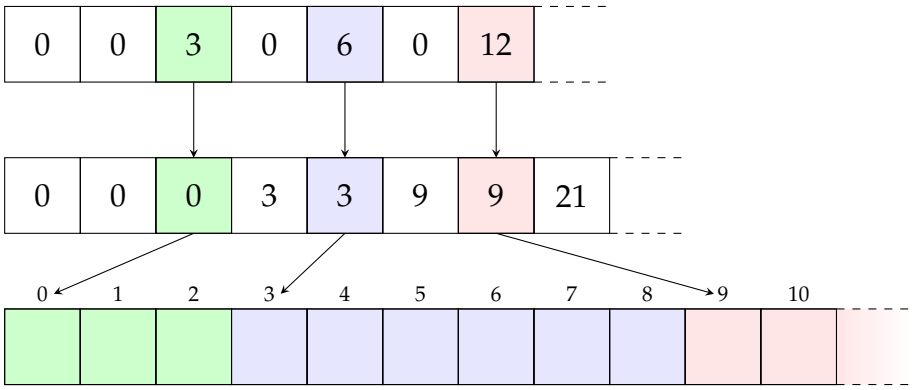

Figure 5.4: Scan operation on `voxelVerts` array gives indices to final result array for every active voxel.

Generating triangles

Triangle generation is handled by `generateTriangles` kernel. It figures out the number of voxel it's working on from `compactedVoxelArray`. Next, array in local memory is allocated for computing locations of vertices and normal vectors on each of the 12 edges on the cube.

```
1 __local float4 vertList[12*NTHREADS];
  __local float4 normList[12*NTHREADS];
```

Positions are calculated even though certain edges may not have surface border on them. However, using `if` statement in kernel in this context would cause divergence in execution of parallel threads.

From now on, algorithm is the same as in sequential version. List of edges on which vertices lie is read from LUT mentioned in section 5.3. This table is accessed as a 2D texture. Agressive caching strategy implemented in GPUs for texture data access allows the kernel to retrieve the data without much penalty. Positions of vertices and normals on edges read from LUT are read from `vertList` and `normList` arrays, and are written into result vertex position and normal arrays on indices read from `voxelVertsScan` array.

Chapter 6

Programming project description

6.1 Introduction

Programming project of this thesis is a set of command line programs, collectively called *karstgen*, that take the description of karst cave fracture net and generate polygon mesh in simple and popular Wavefront OBJ textual file format[1].

Models created this way may be opened in 3D editing program for further editing and examination.

Karstgen can also create models for *Vorticity* game engine that was created by the author together with mr Michał Siejak for graphics related courses[2] during licenciate studies at Adam Mickiewicz University in Poznań.

6.2 Architecture

Karstgen was created with Unix Philosophy in mind (Raymond 2003). It is made of two programs named *blobber* and *mcblob* that have clearly defined responsibilities and communicate through simple textual data format. Both programs may take input either from files or from standard input so they can be piped together with shell pipes. Data flow of karstgen is presented in Figure 6.1.

[1] http://www.martinreddy.net/gfx/3d/OBJ.spec

[2] Computer Graphics and Visualiation, summer semester 2009/2009 and Group Project, summer semester 2009/2010

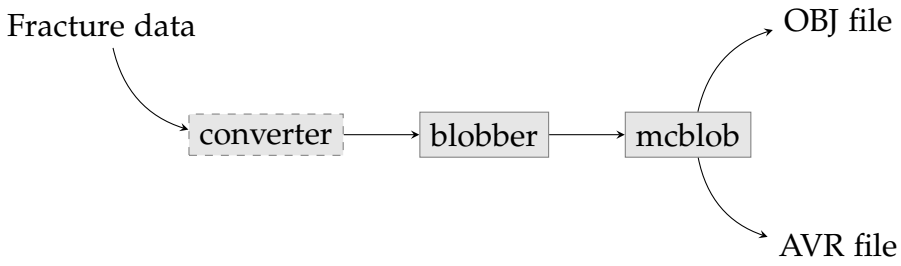

Figure 6.1: Data flow of karstgen program. Converter part is required when fracture net description is other than expected by blobber.

Blobber

Blobber takes description of a fracture net in a simple textual format and generates list of metaballs (see Equation 6.3). It can optionally tilt positions and sizes of metaballs in random but adjustable manner for more natural–looking results. Blobber also controls quality of the final geometry. For information about runtime parameters invoke:

```
./blobber --help
```

Mcblob

Output generated by blobber is consumed by program named *mcblob*[3] which is general purpose tool that may be used to generate geometry from a list of metaballs in 3D space through OpenCL–accelerated Marching Cubes algorithm.

6.3 Implementation details

Metaballs

Implementation heavily relies on rendering with metaballs. Metaball in 3D space is a scalar function in the form (Blinn 1982, p. 4):

$$f(x,y,z) = Te^{\frac{B}{R^2}r^2 - B} \tag{6.1}$$

[3] **M**arching **C**ubes from **blob**s

where r is distance from point (x, y, z) to the centre of the metaball, B is „blobiness" factor that controls tendency to „melt" with other metaballs, T is isovalue that will be used for rendering and R is the radius of the metaball if it was isolated from other blobs. This equation is basically a Gaussian bump with expected value in the middle of the metaball.

If more than one metaball is present in the scene, density function (see Definition 1) is in the form:

$$d(x, y, z) = \sum_{i=0}^{n} f_i(x, y, z) \tag{6.2}$$

where n is the total number of metaballs in the scene and f_i is function 6.1 of the i-th metaball.

Metaballs were discovered by Jim Blinn during his work on visualisation of molecular structures (Blinn 1982). Density function was derived from equation defining density of electron field of hydrogen atom as used in quantum mechanics.

This „melting" property visible when metaballs are close to each other gives somewhat „organic" look and feel of structures generated with them (see Figure 6.2).

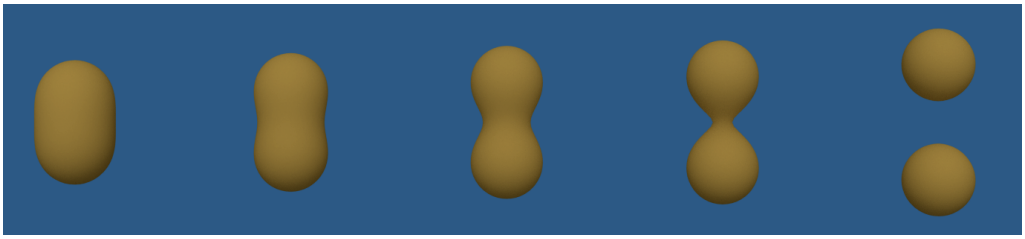

Figure 6.2: Two metaballs at various distances showing how they are „melting" together when getting closer to each other. Geometry was generated with *mcblob* program and final image was rendered with Blender 2.68 with Cycles renderer. Input file for mcblob that generates this model is included with the thesis in file `fig_metaballs.in` in kartsgen examples.

Overview

Both blobber and mcblob are implemented in C++ language with latest C++11 version of the standard. Build system used to compile the code is

CMake[4] – meta build system that can generate native projects for various IDEs[5] and actual build systems. Executables use *Boost Program Options* library for parsing command line arguments and providing help.

Karstgen uses unit testing framework *Google Test*[6]. Documentation is automatically generated from sources with Doxygen tool.

Blobber

For vector data structures blobber uses *GLM*[7] – a mathematical library that resembles GLSL[8].

It reads information about diameters of fractures in fractures network and places blobs along these fractures with diameters roughly the same as of these fractures.

Blobber works on data structure named `DataPoint`:

```
struct DataPoint
{
        int x, y, z;
        float midDiam;
        std::vector<float> xData;
        std::vector<float> yData;
        std::vector<float> zData;
};
```

This structure can describe three fractures originating in index (x, y, z) in the fracture net and going along each axe in ascending direction. Each fracture is described as a vector of uniformly distributed diameters. If there is only one diameter in a vector it is assumed that the fraction it represents has the same diameter along its whole length. When no diameters are present in some vector, it is assumed that there is no fracture in this direction. Additional field `midDiam` is a diameter of blob that should be placed in the intersection of the three fractures (see Figure 6.3).

[4] Cross–platform make
[5] Integrated Development Environment
[6] http://code.google.com/p/googletest/
[7] http://glm.g-truc.net/0.9.4/index.html
[8] OpenGL Shading Language

Data points are packed into a structure representing whole fracture network called `FractureNet`:

```
struct FractureNet
{
        //size of net in number of dataPoints
        int x; int y; int z;

        //length of single fracture in each direction
        float xLen; float yLen; float zLen;

        std::map<std::tuple<int, int, int>, DataPoint>
            dataPoints;
};
```

Since fracture net is usually quite sparse, dictionary structure[9] is used to store data points instead of array or vector. Keys in this dictionary are tuples containing 3D index in a fracture net. This way, given one data point it is easy to find its neighbours.

[9] `std::map` from C++'s Standard Template Library

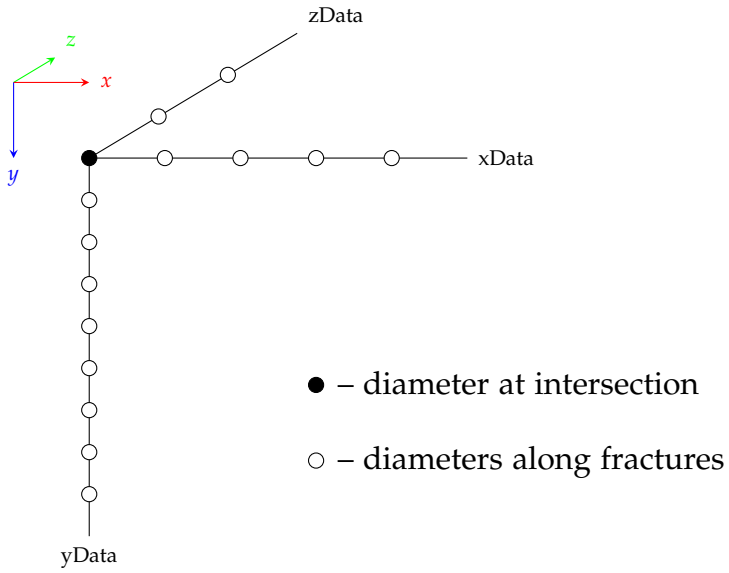

Figure 6.3: Graphical representation of `DataPoint` – a basic structure that blobber works on. In this example, there are 4 diameters in x direction, 2 in z direction and 8 in y direction.

Placement of blobs

Blobber works on one data point at a time through `blobsFromDataPoint()` function. First, one point is placed in the intersection of the axes with diameter equal to `midPointDiam`. Then, vector for each non–empty axe is processed by `blobsOnVector()` function. Besides the vector of diameters, this function takes `midPointDiam` of the data point, and optionally `nextDpMidDiam` which is `midPointDiam` of the next data point along this axis if such data point exists. Fast lookup of next data point is possible thanks to dictionary storage of data points. If there is no neighbour data point at the end of the axis, `nextDpMidDiam` is considered to be 0.

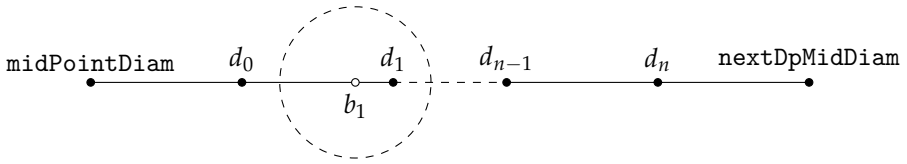

Figure 6.4: Blobs are placed along fracture defined as set of diameters $(d_0, d_1, \cdots, d_{n-1})$. In this example diameter of blob b_1 will be a linear interpolation of diameters d_0 and d_1 proportional to distance from these diameters.

Function `blobsOnVector()` places blobs on the fracture line until it's fully covered. Diameters of blobs are determined in a way described in Figure 6.4.

Format of the input to blobber is described in generated documentation:

```
\doc\blobber\html\index.html
```

Output format is the same as input format of mcblob program and is described in its help:

```
./mcblob --help
```

Mcblob

Mcblob uses OpenCL (chapter 4) to calculate density function from list of blobs provided in input, generates polygon mesh of isosurface definded by this function using Marching Cubes algorithm (chapter 5) accelerated with OpenCL and saves results to file in one of two formats.

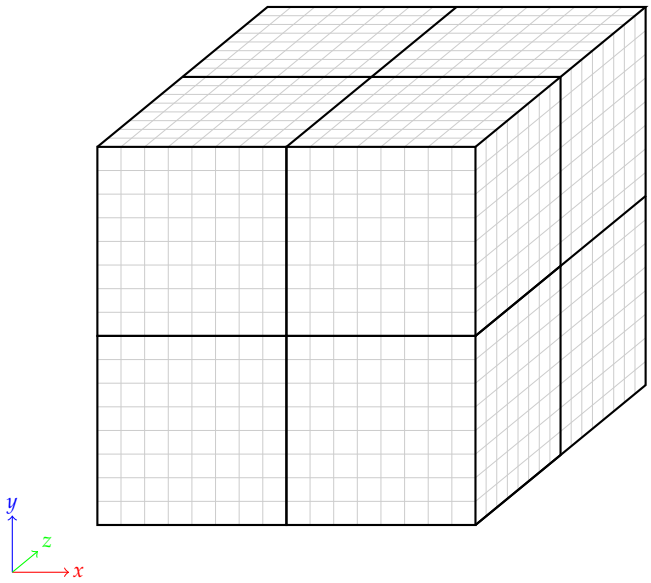

Figure 6.5: In this configuration, domain is divided into $2 \times 2 \times 2 = 8$ grids (blocks), and each one of them consists of $8 \times 8 \times 8 = 512$ voxels totalling $512 \times 8 = 4096$ voxels.

Bounds of 3D space within which geometry will be generated is defined in input. It's divided into blocks which are worked on one at a time. For each block, density function is calculated from blobs, followed by generating geometry.

Finally, when geometry from all blocks is calculated, mcblob writes output to disk in one of two supported formats.

Calculating density function

Each block of voxels is kept by mcblob in a class called `Grid`. It contains values of density function on the vertices of the cubes in a block. These values are stored as one–dimensional array, that is transferable between host and device memory. If numbers of voxels on axes x, y and z are v_x, v_y and v_z respectively, `Grid` will store 1D array of `float4` values that is $(v_x + 1) \times (v_y + 1) \times (v_z + 1)$ elements long.

Four floats are kept for each vertex instead of one to facilitate computation of normal vectors. In case of `Grid` components x, y and z store density function values in positions shifted by small value along re-

spective axes – a gradient of density function in position of the vertes. Finally, w component stores value at the vertex itself.

Density function of a set of blobs is calculated by method:

```
Blob::runBlob()
```

that adds array of blobs to the grid according to Equation 6.2. For better performance, this method utilizes constant memory. Device is queried for size of constant memory via `cl::Device::getInfo()`. As mentioned in chapter 4 constant memory is efficient for data that is simultaneously accessed by many threads. In case of `Blob` program, every thread iterates over all blobs. If size of blob data exceeds constant memory size, input is divided into packages that are within the limit, and kernel is simply invoked multiple times, until all blobs are processed.

This method of implementation was inspired by MRI[10] reconstruction program for CUDA described in Kirk and Hwu 2010, in chapter 8.

Generating geometry

When density function of all blobs is calculated for a single grid, such grid is submitted to:

```
MarchingCubes::compute()
```

method that runs OpenCL–powered Marching Cubes implementation (see section 5.4). Once the geometry for all blocks is generated, it's passed to one of two exporter functions that write the results to disk in format selected by the user. Wavefront OBJ output can be imported by virtually every 3D graphics software and AVR can be read by aforementioned Vorticity game engine.

Using `blobber` and `mcblob` together

Blobber and mcblob are naturally fit to be executed in shell via piping. To run karstgen with provided examples, go to folder where it was compiled and type:

```
cat [input] | ./blobber | ./mcblob -o out.obj
```

Where `[input]` is path to a file with description of fracture net. Examples are located in `examples\blobber`. This will generate `out.obj` file that can be imported to 3D graphics program.

[10] Magnetic resonance imaging

To randomly disturb positions of blobs by 15% and sizes by 10% of their diameter, invoke karstgen in the following manner:

```
cat [input] | ./blobber -p 15 -s 10 | ./mcblob -o out.obj
```

6.4 Example outputs

Below are screenshots of outputs of karstgen with references to input files used to produce them.

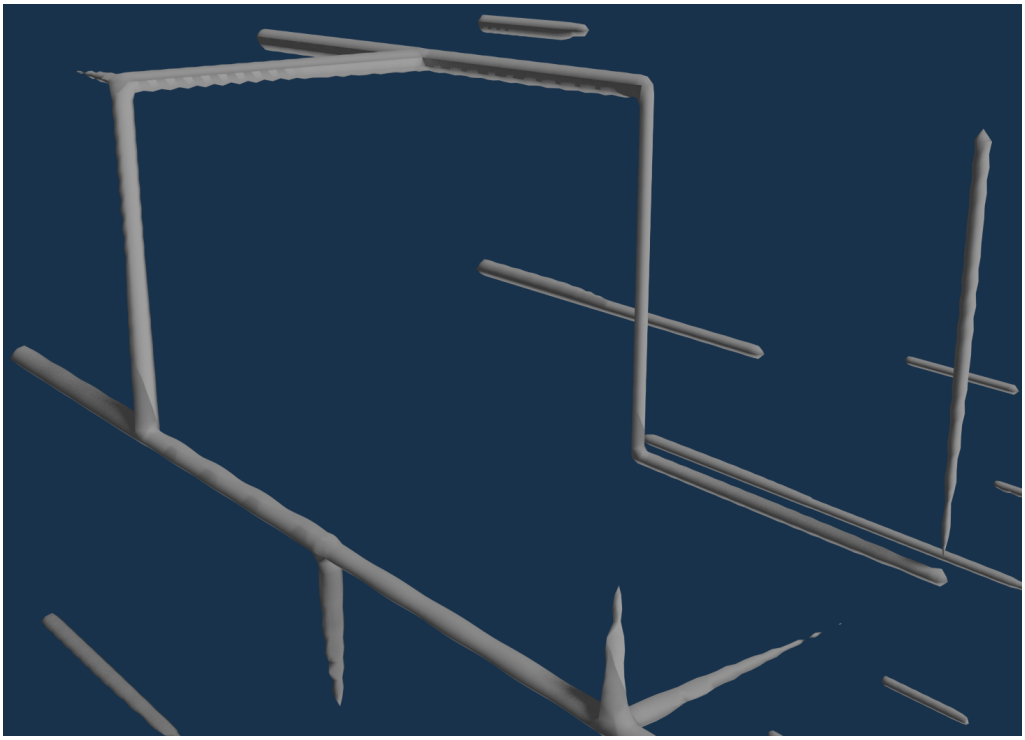

Figure 6.6: Rendering of result of simulation generated by KARSTAQUIFER tool (Kaufmann 2009). Input data file courtesy of Mr Thomas Hiller PhD from Free University of Berlin. Available in file `examples\blobber\hiller.in`. Be advised, that due to very large domain, computations done by karstgen may take several hours. Figure rendered with Blender renderer.

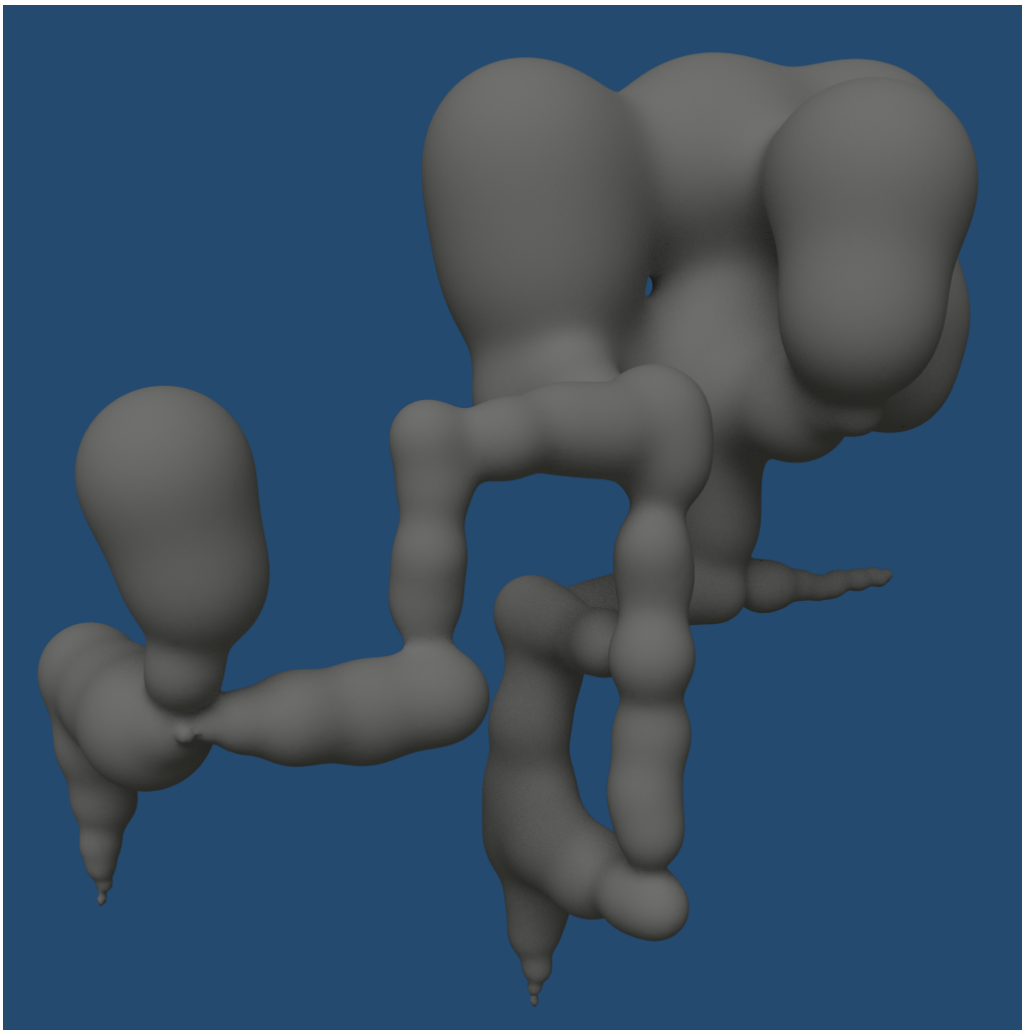

Figure 6.7: Render of synthetic blobber input file created by author. Rendered with Blender Cycles renderer. Input file available at `examples\blobber\synthetic.in`

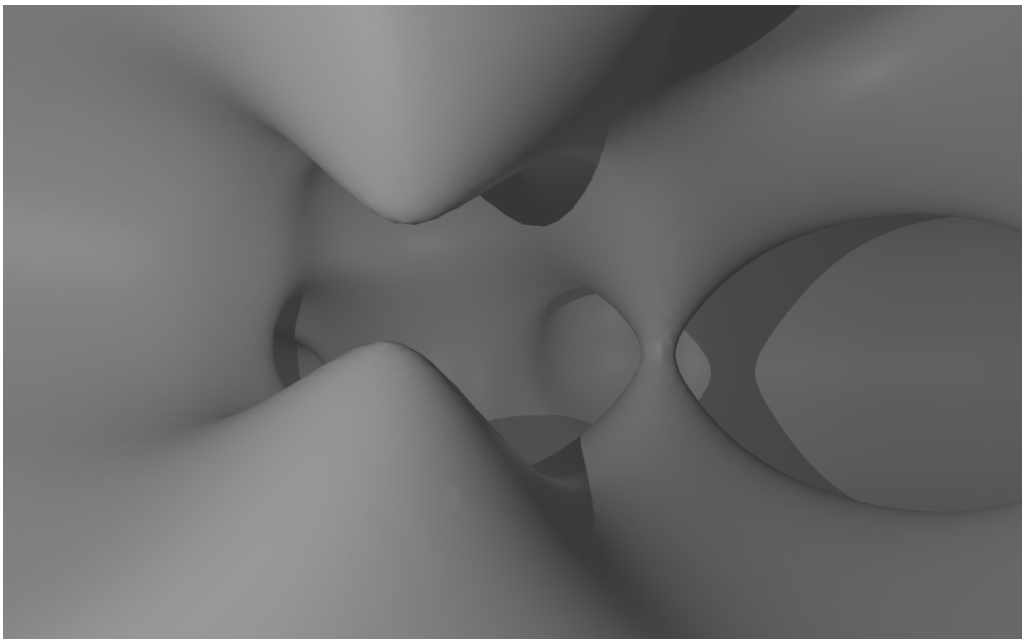

Figure 6.8: Interior of the cave generated for Figure 6.7. Rendered with Blender renderer.

Chapter 7

Conclusions and further work

Programming project accompanying this thesis, called *karstgen*, provides usable way to represent karst simulation data as a three–dimensional polygon mesh. It consumes data that is similar in structure to formats used as output of these simulations. Since karstgen is able to produce models in popular OBJ file format, its results can be imported by most 3D modelling software suites for further modifications and examinations.

Method of geometry generation through placing of blobs along an aquifer fractures and rendering them with Marching cubes proved to yield positive visual results. Possibility of adding randomness to generated meshes helps in achieving visually pleasing (Figure 6.8) meshes that could be used in e.g. video games. Given simplicity of karstgen input file format it would be easy to write application aimed at artist that could be used for creating caved terrains by providing input data to karstgen.

To increase performance, karstgen uses OpenCL (see chapter 4) to accelerate computations on GPU. With multi–vendor support for this standard, it is highly probable, that OpenCL will be supported in the future.

7.1 Possible development of karstgen

Right now, karstgen produces only the primary geometry of fracture network based on diameters. To create more realistically–looking output, microstructure of the cave wall would have to be introduced by e.g. bump mapping.

No material data of cave walls is generated. Karstgen produces only the mesh which can be later textured in external program. Additional data, like amount of water flowing through fracture or calcium concentration, that could be passed to blobber and later to mcblob by simulation software could be helpful in procedural material generation.

Because Marching Cubes algorithm is executed independently for every voxel in the domain, almost all[1] vertices are created twice. Adjacent voxel don't know about each other, and don't share vertices. Mesh created this way is larger than needed. Additional step that removes doubled vertices could be introduced[2].

Even though computation is accelerated with GPU, rendering of large dataset could be very time–consuming. As described in section 6.3 each block is computed independently. In current implementation, data for one block is roughly 10MB in size. It's constrained mainly by abilities of GPU implementation of prefix sum algorithm that limits size of single block to $64 \times 64 \times 64$ voxels. With modern GPUs having at least 1GB of video ram, many more blocks could be uploaded to GPU and their synchronization could by coordinated by OpenCL event mechanism. This could reduce context–switching significantly. Also, density function calculation could be greatly shortened. Typically, metaballs that are far from currently calculated location have little or no impact on the density function. That is why domain could be partitioned into chunks in which only relevant metaballs are taken into account. To further improve performance some other, possibly less computationally expensive function with similar characteristics could be used for single metaball than one showed in Equation 6.2.

[1] Except the ones on the boundary of the domain
[2] Such functionality is available in e.g. Blender

Bibliography

Annable, William K. (2003). "Numerical Analysis of Conduit Evolution in Karstic Aquifers". Ph.D. University of Waterloo (cit. on p. 16).

Aylward, Gordon H. and Tristan John Victor Findlay (2008). *SI Chemical Data*. John Wiley & Sons Australia, Limited. ISBN: 9780470816387 (cit. on p. 13).

Blelloch, Guy E. (Nov. 1990). *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University (cit. on p. 47).

Blinn, James F. (July 1982). "A Generalization of Algebraic Surface Drawing". In: *ACM Trans. Graph.* 1.3, pp. 235–256. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/357306.357310 (cit. on pp. 50, 51).

Buhmann, Dieter and Wolfgang Dreybrodt (1985a). "The kinetics of calcite dissolution and precipitation in geologically relevant situations of karst areas: 1. Open system". In: *Chemical Geology* 48.1–4, pp. 189 – 211. ISSN: 0009-2541. URL: http://www.sciencedirect.com/science/article/pii/0009254185900464 (cit. on p. 15).

— (1985b). "The kinetics of calcite dissolution and precipitation in geologically relevant situations of karst areas: 2. Closed system". In: *Chemical Geology* 53.1–2, pp. 109 –124. ISSN: 0009-2541. URL: http://www.sciencedirect.com/science/article/pii/0009254185900245 (cit. on p. 15).

Dreybrodt, Wolfgang and Franci Gabrovšek (2002). "Basic processes and mechanisms governing the evolution of karst". In: *Evolution of karst: from prekarst to cessation*. Coronet Books (cit. on pp. 13, 16).

Fairchild, Ian J. and Andy Baker (2012). *Speleothem Science: From Process to Past Environments*. Blackwell Quaternary Geoscience Series. John Wiley & Sons. ISBN: 9781444361063 (cit. on p. 14).

Field, Malcolm S. (2002). *A Lexicon of Cave and Karst Terminology with Special Reference to Environmental Karst Hydrology*. United States Environmental Protection Agency (cit. on pp. 9, 10).

Fletcher, Dan (June 2010). *Massive Sinkhole Opens in Guatemala City*. URL: http://newsfeed.time.com/2010/06/01/giant-sinkhole-opens-in-guatemala-city/ (cit. on p. 12).

Ford, Derek C. and Paul Williams (2007). *Karst Hydrogeology and Geomorphology*. John Wiley & Sons. ISBN: 9780470060056 (cit. on p. 6).

Visualizing Large Procedural Volumetric Terrains Using Nested Clip-Boxes (2005). SIGGRAPH 2005 (cit. on p. 17).

Gaster, B. et al. (2012). *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Elsevier Science. ISBN: 9780124055209 (cit. on pp. 19, 25, 26, 30, 35).

Geiss, Ryan (2007). "Generating Complex Procedural Terrains Using the GPU". In: Nguyen, Hubert. *Gpu gems 3*. First. Addison-Wesley Professional. Chap. 1. ISBN: 9780321545428 (cit. on pp. 16, 39).

Golscheider, Nico and David Drew (2007). *Methods in Karst hydrogeology*. IAH international contributions to hydrogeology. Taylor & Francis Group. ISBN: 9780415428736 (cit. on p. 12).

Gummaraju, Jayanth et al. (2010). "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, pp. 205–216 (cit. on p. 32).

Harris, Mark, Shubhabrata Sengupta, and John D. Owens (2007). "Parallel Prefix Sum (Scan) with CUDA". In: Nguyen, Hubert. *Gpu gems 3*. First. Addison-Wesley Professional. Chap. 39. ISBN: 9780321545428 (cit. on p. 46).

Hill, Carol A. and Paolo Forti (1997). *Cave Minerals of the World*. t. 2. National Speleological Society. ISBN: 9781879961074 (cit. on p. 10).

Hiller, Thomas (2013). "Modelling the Evolution of Karst Aquifers in Three Dimensions". Ph.D. Free University of Berlin (cit. on pp. 15–17).

Holland, H.D., T.V. Kirsipu, and U.M. Oxburgh (1964). "On some aspects of the chemical evolution of cave waters". In: *Journal of Geology* 72 (cit. on p. 14).

Kaufmann, Georg (2003). "Numerical models for mixing corrosion in natural and artificial karst environments". In: *Water Resources Research* 39.6, n/a–n/a. ISSN: 1944-7973. URL: http://dx.doi.org/10.1029/2002WR001707 (cit. on p. 16).

— (2009). "Modelling karst geomorphology on different time scales". In: *Geomorphology* 106.1–2, pp. 62 –77. ISSN: 0169-555X. URL: http://

www.sciencedirect.com/science/article/pii/S0169555X08004091 (cit. on pp. 16, 17, 57).

Kaufmann, Georg, Douchko Romanov, and Thomas Hiller (2010). "Modeling three-dimensional karst aquifer evolution using different matrix-flow contributions". In: *Journal of Hydrology* 388.3–4, pp. 241 –250. ISSN: 0022-1694. URL: http://www.sciencedirect.com/science/article/pii/S0022169410002441 (cit. on p. 16).

Kipfer, Peter and Rüdiger Westermann (Mar. 13, 2005). "Improved GPU Sorting". In: Nguyen, Hubert. *Gpu gems 2*. Addison-Wesley Professional. Chap. 1. ISBN: 978-0321335593 (cit. on p. 21).

Kirk, David B. and Wen-mei W. Hwu (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123814723, 9780123814722 (cit. on pp. 19, 56).

Kolb, A., L. Latta, and C. Rezk-Salama (2004). "Hardware-based simulation and collision detection for large particle systems". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. HWWS '04. Grenoble, France: ACM, pp. 123–131. ISBN: 3-905673-15-0. URL: http://doi.acm.org/10.1145/1058129.1058147 (cit. on p. 31).

Lorensen, William E. (2007). *Marching Cubes - MC Wiki*. Website maintained by co-author of Marching cubes algorithm. URL: http://www.marchingcubes.org/index.php/Marching_Cubes (cit. on p. 40).

Lorensen, William E. and Harvey E. Cline (Aug. 1987a). "Marching cubes: A high resolution 3D surface construction algorithm". In: *SIGGRAPH Comput. Graph.* 21.4, pp. 163–169. ISSN: 0097-8930. URL: http://doi.acm.org/10.1145/37402.37422 (cit. on pp. 40, 41).

— (Dec. 1, 1987b). "System and method for the display of surface structures contained within the interior region of a solid body". US 4710876 A. General Electric Company (cit. on p. 41).

Marshak, Stephen (2007). *Earth: Portrait of a Planet*. 3rd ed. W. W. Norton & Company (cit. on p. 11).

Migoń, Piotr (2010). *Geomorphological Landscapes of the World* (cit. on p. 9).

Munshi, Aaftab, ed. (2012). *The OpenCL Specification, version 1.2*. Khronos OpenCL Working Group. URL: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (cit. on pp. 28, 29).

NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009). Whitepaper. NVIDIA Corporation. URL: http://www.nvidia.com/content/

PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (cit. on p. 35).

Plummer, L. N., T. M. L. Wigley, and D. L. Parkhurst (1978). "The kinetics of calcite dissolution in water systems at 5°C to 60°C and 0.0 to 1.0 atm CO_2". In: *American Journal of Science* (cit. on p. 13).

Raymond, Eric Steven (2003). *The Art of UNIX Programming*. Addison-Wesley professional computing series. Pearson Education. ISBN: 9780132465885 (cit. on p. 49).