Technische Universität München

Fakultät für Mathematik
Lehrstuhl für Geometrie und Visualisierung

Domain Parallel Machines

An Abstraction of GPU Shader Programming and Applications in Mathematics

Aaron Montag

Vollständiger Abdruck der von der Fakultät für Mathematik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende(r):	Prof. Dr. Peter Gritzmann
Prüfer der Dissertation:	1. Prof. Dr. Dr. Jürgen Richter-Gebert
	2. Prof. Dr. Christian Mercat
	3. Prof. Dr. Dr. h.c. Gert-Martin Greuel

Die Dissertation wurde am 16.10.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Mathematik am 26.03.2020 angenommen.

Abstract

In this thesis, a computational model is introduced and studied, which abstracts and idealizes computers with access to fragment shaders. A variety of algorithms naturally applicable on this model is presented. The presented work shows that while the set of functions computable by this model remains the same, the running times can be drastically reduced through parallelization compared to conventional models. Some of the algorithms designed for the model can be approximated using fragment shaders. A criterion for this approximability is derived. The practical part describes how complex fragment shader programs can be generated automatically from a description in a high-level language. A raycaster for algebraic surfaces and a tool for conformal image deformation are presented as mathematical applications of this conversion process.

Zusammenfassung

In dieser Arbeit wird ein Rechenmodell, das Computer mit Zugriff zu Fragment Shader abstrahiert und idealisiert, eingeführt und untersucht. Es wird eine Reihe von Algorithmen vorgestellt, die auf diesem Modell auf eine natürliche Weise angewendet werden können. In der Arbeit wird gezeigt, dass zwar der Umfang der durch dieses Modell berechenbare Funktionen gleich bleibt, die Laufzeiten aber durch Parallelisierung im Vergleich zu herkömmlichen Modellen drastisch verkürzt werden können. Einige der für das Modell entworfenen Algorithmen lassen sich mithilfe von Fragment Shadern approximieren. Ein Kriterium für diese Approximierbarkeit wird in der vorliegenden Arbeit hergeleitet. Als praktischer Teil wird beschrieben, wie sich komplexe, in einer Hochsprache beschriebene, Fragment Shader Programme automatisiert generiert werden können. Als mathematikdidaktische Anwendung dieses Umwandlungsprozesses werden ein Raycaster für algebraische Flächen und ein Werkzeug zur konformen Bilddeformierung vorgestellt.

Acknowledgment

Ich möchte mich bedanken bei

der Technischen Universität München, der Landesregierung, die mich während meiner Promotion unterstützt haben. Diesen Institutionen verdanke ich das Vertrauen und Privileg, finanziell unabhängig promovieren zu können. Das universitäre Umfeld war für das Entstehen dieser Arbeit unabdingbar. Einen Dank der Gesellschaft, die Herausbildung solcher Institutionen ermöglicht und unterstützt.

Felix Woitzel, aus dessen Idee der grafikkartengestützten Feedbackschleifen die Grundlagen für diese Dissertation gekeimt sind.

Jürgen, für die zahlreichen Inspirationen und Unterstützungen, die weit über den Inhalt der Dissertation hinaus gehen.

Meinen Kollegenkreis des Lehrstuhls, Abishek, Ben, Benno, Bernhard, Carsten, Diane, Fabian, Hermann, Jutta, Katharina, Lena, Martin, Michael, Uli, Stefan, Tim, Vanessa, Zi, für das angenehme, herzliche Arbeitsumfeld, Diskussionen und gemeinsame Mahlzeiten. Danke für die Unmenge an klasse Aktionen mit Euch.

Lena, für das Korrekturlesen weiter Teile dieser Arbeit und zahlreiche Verbesserungsvorschläge. Außerdem war es schön, mit dir ein Büro zu teilen!

Christian Mercat, Jürgen Richter-Gebert and Gert-Martin Greuel for going through all the details of this thesis. In particular, thank you, Christian, for sending me a list of typos and making this thesis better.

Meinen Eltern und meinen Lehrern und Lehrerinnen, die mich dorthin gebracht haben, wo ich jetzt bin.

Jelena, für ihre fortdauernde Unterstützung. Hvala ti na čitanju mog rada. Hvala ti i za korekture i pametne savjete. Kad mi je bilo teško, hrabrila si me i bila moj oslonac. Uljepšala si mi period pisanja doktorata, i na tomu sam ti zahvalan!

Contents

Chapter 1.

Introduction

Most modern computers have access to massively parallel computational units like a GPU (Graphics Processing Unit). In recent history, designing programs that utilize the GPU became an important task because GPUs drastically accelerate a wide set of algorithms. The number of possible new applications raises exorbitantly. For instance, ray-tracing algorithms visualize scenes in video games in real-time. However, there are also several non-visual applications for GPUs: General-purpose computing on graphics processing units (GPGPU) is, among others, often found in linear algebra, signal and image processing, physical simulations, and machine learning (Owens et al., 2007; Singh and Reddy, 2014).

For engineers in the field of GPU programming, the focus is set on the design of efficient algorithms for a given problem. A major point of view of this thesis is rather that of a theorist studying the limits of a general computational model in which certain massive parallelism inspired by GPU shader programs is possible. A *precise* mathematical definition of the studied computer is essential in order to make general statements about the possible computations. Moreover, this definition should be on a *suitable level of abstraction* for the studied context. Suitable abstractions can be helpful to focus on the essential mechanisms.

Shader programming is often perceived as a low-level technical process, including many detail in a way that the understanding of the overall picture gets lost. An attempt to transfer existing shader programs into a formal description would, therefore, yield a model of a wrong level of abstraction. This discrepancy is also present in other contexts as, for example, in the field of numerical analysis, where an exact specification of a physical computer would be too elaborate. Several constraints, such as the necessity of replacing real numbers by floating-point numbers, are often neglected in the first step of designing a numerical algorithm. Historically, a prominent formal description of a computer is given by Turing machines. However, Turing machines are an unsatisfactory basis for many numerical algorithms. For instance, consider the Newton-Raphson method which iteratively searches for roots of a function by operating on real or complex numbers. A Turing machine can only store objects encoded as a finite string over a finite alphabet such as integers or rational numbers. In this context, other computational models are more suitable. Blum, Shub, and Smale (1989) gave a possible definition for algorithms in numerical analysis by introducing a model for computations over *real numbers*. Instances of their model will be referred to as *(uniform) BSS machines*. With a precise definition of their computational model valuable insights into their capabilities and limitations are gained. For instance, it is shown that no algorithm exists in their model of real computation that can decide whether a given complex number belongs to the famous Mandelbrot set or not. In addition, they reduced questions about complexity classes of their model to questions about algebraic geometry.

Similar to the approach of Blum et al. (1989) for numerical analysis, an *abstract* and *idealized* formalization for algorithms that utilize shaders of the GPU (or comparable parallel units) is introduced in the following work. We require of the model to be both a *precise* and a *suitable* abstraction for the context of several algorithms designed for a particular class of computers having access to massively parallel computational units. Using this model, we want to explore the possibilities of such machines.

BSS machines overcame one discontinuity in the model of classical computation: On Turing machines and today's physical computers, all stored values are finitely encoded making them discrete. Blum et al. (1989) introduced a computational model that allows saving values of an arbitrary ring, such as $\mathbb{R}$, in its variables. In general, the states of the computer are no longer finitely representable. If a ring, such as $\mathbb{R}$, is used, then continuity in the values of the stored numbers is yielded. However, variables and parallel processors remained discrete. Only a finite number of variables can be stored at the same time. BSS machines allow continuity in the values. We want to achieve continuity *both* in the values *and* their storage location.

Therefore the computational possibilities under the assumption of no discretization between variables and parallel processors shall be studied. Technology itself provides motivation for this by observing how the GPU fragment shader generates images. Let us pretend that these images are *not composed of discrete pixels*. Instead, the GPU could compute the image at every continuous coordinate simultaneously. One of the central question of this thesis is: Which possibilities would open up having a computer that can store such generated infinitely-fine "textures" and use them for consecutive computations? We will call such machines Domain Parallel Machines (DPMs).

1.1. Main results and contributions

The first part of this work is on *foundations* of DPMs.

DPMs can, aside from their abstract definition, be considered as an abstraction of computers with access to fragment shader code (Chapter 2). However, this theoretical model appears also in physical systems such as idealized video-feedback loops (Chapter 4). Under certain continuity conditions, the computation of a DPM can be *approximated* by a physical computer with access to the fragment shader on the GPU. The DPM computation can be considered as a limit of taking higher and higher resolutions for parallelized fragment shader programs (Chapter 6). Developing an algorithm in our model is more appropriate for (spatial) parallel mathematical procedures. Many of these algorithms can be approximated on fragment shaders by dropping the continuity in value and space through discretization and resampling. These approximations give often rise to suitable and efficient algorithms that can be executed on physical GPUs (Chapter 3).

It turns out that the computational power of DPMs remains the same as the computational power of BSS machines, in the sense that the *induced classes of decidable problems of both models are equal*. However, many problems can be solved *faster* by DPMs than by any BSS machine (Chapter 5). In particular, for any finite-dimensional BSS machine computing a function in running time $\Theta(t(n))$ there exists a DPM that simulates the same computation in $\Theta(\log t(n))$.

The second part is on *implementation* of a system that eases the programming of GPU programs by abstraction in a high-level language. It relies on a transcompilation system of a high-level programming language in a low-level programming language of the GPU. The challenges lie in *type inference* (Chapter 7).

The third part is on *applications* of our implementation. The applications, which could be developed without almost no technical hurdles due to the transcompilation-implementation, include a raycaster and a path tracer for implicitly defined surfaces (Chapter 8), a real-time visualization program of analytic landscapes (Section 8.5) and a program to deform images given mathematical formula (Chapter 9). In particular, a tool for deforming spherical footage and applying spherical Droste effects on it in real-time has been developed (Section 9.3). These implementations require non-trivial mathematical and numerical foundations. However, the applications presented in this thesis turned out to be widely used in mathematics communication and have a non-negligible impact. IMAGINARY and the New York Times have published the raycaster for algebraic surfaces at Valentine's Day. The shape of a heart has been rendered as an algebraic surface on their websites (and IMAGINARY plans to use the program in several further installations). Ponce Campuzano (2019) has used the application to visualize algebraic surfaces in his interactive electronic book *Complex Analysis – A Visual and Interactive Introduction*. The deformation program to generate spherical Droste effects in real-time has been presented by the mathematics communicator Matt Parker on stage several times to a broad audience.

1.2. Overview of the individual chapters

In Chapter 2, we will first give two equivalent definitions of DPMs. One is an *adaption of the BSS-model*. The other one corresponds to a grammar of *pseudo-code* similar to GPU shader programs. However, the pseudo-codes idealize GPUs in the sense that the pixels are abstracted away. We use this previously developed pseudo-code model in Chapter 3 to describe several given mathematical algorithms in a natural way. This shows the computational strength of the DPM model. In Chapter 4, several examples are presented where the DPM model has (approximating) realizations in physical-world. First steps in runtime complexity of DPMs and relations to other computational models are investigated in Chapter 5.

In Chapter 6, conditions are derived under which GPUs can approximate DPMs. In Chapter 7, a simplified scheme for GPU shader programming is developed, which has been realized through *CindyGL*. This scheme leverages the creation for WebGL applications within the context of a dynamical geometry software. Finally, in Chapter 8, applications of this framework in the visualization of mathematics are presented. A method to render *implicit surfaces* (*algebraic* and *non-algebraic*) by raycasting with a practical realization is introduced. A slight adaption results in a path tracing algorithms for implicit surfaces using a Monte Carlo integration. The methods provide the possibility to render *analytic landscapes* of complex functions almost for free. Another important application of the framework is studied in Chapter 9, namely deformation of images, also possible in a conformal way. This can be used to render generalized *Droste-effects*. In Section 9.3, the same schemes are applied to *spherical* footage, as a contribution to communication of mathematics.

1.3. Applets

Besides theory, several applets are part of this thesis. In the text, they are indicated via the symbol ▷ and a number. That means, the applet is found on the companion storage device under that given number. Alternatively, all the applets can be downloaded from `https://aaron.montag.info/dissertation/applets.zip`. As a

heuristic evidence that this archive has not been modified after the publication, the SHA-256 hash of this zipped-archive at time of submission is
`4b18569e14c19986184fa9ffc24f4a3e4dd116e8faf36204ab42b458e97b5217`.

Due to browser security, several of these applets only work if they are hosted on a webserver. Currently the applets work best with Firefox or Chrome.

In the PDF-version of the thesis, the numbers are clickable, and an online version of the applets can be viewed immediately. Furthermore, any applet with number $k \in \mathbb{N}$ can also be opened online following the address
`https://aaron.montag.info/dissertation/k`.

All these applets were created by the author himself. However, they rely on the frameworks CindyJS (von Gagern, Kortenkamp, Kranich, Montag, Richter-Gebert, Strobel, and Wilson, 2019) and CindyGL (Montag and Richter-Gebert, 2016), which are available under the Apache 2 license.

1.4. Terminology and notation

We do not include 0 to the natural numbers, i.e. $\mathbb{N} = \{1, 2, 3, 4, \dots\}$ and write $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Often it is convenient to use the set of the first n integers. For $n \in \mathbb{N}_0$ we set

$$[n] := \begin{cases} \emptyset & \text{if } n = 0 \\ \{1, 2, \dots, n\} & \text{if } n \geqslant 1 \end{cases}$$

and

$$[n]_0 := \begin{cases} \emptyset & \text{if } n = 0 \\ \{0, 1, \dots, n-1\} & \text{if } n \geqslant 1. \end{cases}$$

We write R for an arbitrary ring. In most cases, R can be thought of as $\mathbb{R}$. In this work, several functions with fixed domain and co-domain are considered. With

$$R^D = (D \to R) = \{f : D \to R\} = \bigtimes_{d \in D} R$$

we denote the space of all functions from the domain D to R. If R is a field, then R^D becomes an R-vector space.

With $\mathbb{C}^\times = \mathbb{C} \setminus \{0\}$ we denote the multiplicative group of $\mathbb{C}$ and with $\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$ the extended complex numbers are indicated that can also be considered as the Riemann sphere.

Part I.

Foundations: Domain-Parallel-Machines (DPMs)

Chapter 2.

Two equivalent definitions for DPMs

We give two, at the first glance, very different definitions of computational models, which extend existing models. After closer inspection, it will turn out, that both models are only different representations of a common concept. The exact definition of both computational models are lengthy, but we will shortly paraphrase them here:

Model 1 is an idealization of a classical programmable computer, that can, aside from standard computations, run a constant-time-program on a (possibly uncountable) domain of data points in parallel simultaneously. With constant-time, we mean that the running time of the program has an upper bound that is independent of the particular data point that the program has as input. It idealizes typical computers with the GPU corresponding to the parallel unit. Our model is an idealization in the sense that it can store numbers in arbitrary precision and the parallel unit has an infinite number of parallel processors. The parallel operation in our model can be considered as an idealization of the fragment shader of the GPU in the sense that it can describe and run calculations on an infinitely fine-grained sampled resolution.

Model 2 is a finite-dimensional machine having functions as primitive data types. The model is obtained using the definition of a finite-dimensional BSS machine (Blum et al., 2012) and replacing its registers, which in the BSS model can only save real numbers, with registers, that optionally can also store entire functions. Thus, massive point-wise operations are allowed.

We will call representatives of both models Domain-Parallel-Machines (DPM).

Both models can compute elementary operations and comparisons of real numbers. In fact, as we will show in Section 5.1, the computational power (as far as the input can be translated) is equivalent to the computational power of conventional BSS machines. However, the *computational time* often can be drastically reduced with DPMs.

Note that for both BSS machines and DPMs the term *computational time* actually expresses numbers yielded as a result of a formal definition, which, however, aims to give a reasonable asymptotic statement of the real running time of the (imaged) physical realization.

2.1. DPMs as extended finite-dimensional BSS machines

Now, let us start with the definition of the DPM as a finite-dimensional machine having functions as primitive data types (which corresponds to Model 2 of the previous section). Heuristically, a DPM has the following properties:

- A DPM performs operations over a specified mathematical ring R, such as $\mathbb{R}$ or $\mathbb{C}$.
- A DPM has access to a finite number of possible infinite domains D. $\mathcal{D}$ is the set of all such domains.
- A DPM has access to a finite set of registers, that store constants in R or functions $f : D \to R$ for some $D \in \mathcal{D}$. For example, a DPM with the domain $D = [0, 1]^2 \in \mathcal{D}$ might access an idealized infinitely fine sampled square-shaped texture.
- A DPM can compute and store a function $f : D \to R$ on one of its registers if there is a constant-running time program that can compute $f(d)$ for each data point $d \in D$. Possibly, there are infinitely many data points in D. We assume that all these operations run in parallel for every $d \in D$ and hence say that the entire operation to compute $f : D \to R$ also has fixed cost in running time. As we assume that the DPM instantaneously stores all values when a new function $f : D \to R$ is defined, accessing $f(d)$ in consecutive operations for a $d \in D$ is counted as a constant-time operation. (Unlike a "real evaluation" on a classical computational model that might take more time depending on the construction of f). In order to emphasize this behaviour we will use the notation $f[d]$ instead of $f(d)$ in code.
- Similar to a Turing machine, a DPM has a finite directed graph that, together with the data stored in the registers, determines the current state. The currently active node determines which operation the DPM applies on its registers. All nodes except the termination nodes have at least one next node which corresponds to the "flowchart" that the DPM is following.

In the definition of a DPM, sometimes we want to use *functions to a ring* and *plain ring elements* interchangeable. In order to to so, we will introduce a simplifying notation.

Notation 1. Let R be a ring. With $* := R^0$, we denote the one-point space. We write $* = \{x_0\}$. Functions from $*$ to R take a single R-value at x_0 and thus can be identified with R. More formally, the isomorphy $R \cong \{f : * \to R\} = R^*$ holds via the isomorphism $R^* \to R, f \mapsto f(x_0)$. By a slight abuse of notation, we can consider "zero-ary" functions $f : * \to R$ as constants in R and vice-versa.

2.1.1. Finite dimensional DPMs

The following definitions can be considered as an expanded adoption of the definition of a *finite-dimensional BSS machine* from Blum et al. (2012):

Definition 2 (finite-dimensional DPM). Let R be a ring (or a field) and

$$\mathcal{D} \subset \{D \mid D \subset R^k \text{ for some } k \in \mathbb{N}\}$$

finite with $* \in \mathcal{D}$. An R-DPM M over the domains $\mathcal{D}$ is an abstract machine that contains:

- A finite collection $h_1, \ldots, h_n$ of registers, which store values in R or functions from some $D \in \mathcal{D}$ to R. By Notation 1 and $* \in \mathcal{D}$, we can consider every h_i with $i \in [n]$ as a function from $D_i \in \mathcal{D}$ to R. Every register has a fixed domain.

  The space
  $$\mathcal{S}_M = \bigtimes_{i \in [n]} \{f : D_i \to R \mid f \text{ function}\}$$
  denotes the *state space* of M and $\mathcal{S}_M$ is associated to M. $(h_1, \ldots, h_n) \in \mathcal{S}_M$ and $\mathcal{S}_M$ is a module over R (and a vector space if R is a field).

- The machine has two further R-modules (or vector spaces) which both are subspaces of $\mathcal{S}_M$, namely the *input space* $\mathcal{I}_M$ and the *output space* $\mathcal{O}_M$.

- A finite directed graph $G = (V, E)$ with for special node types: *input*, *computation*, *branch* and *output*:
  - The unique *input node* has no ingoing edge and a single outgoing edge. Associated with it is an embedding $I : \mathcal{I}_M \to \mathcal{S}_M$.
  - Each *output node* η has no outgoing edge and a projection $O_\eta : \mathcal{S}_M \to \mathcal{O}_M$.
  - A branch node has exactly two outgoing edges. Attached to every branching node η is an index $i_\eta \in [n]$ such that $h_{i_\eta} \in R$ (i.e. $D_{i_\eta} = *$). There is one edge associated with $h_{i_\eta} \geqslant 0$ to the node β_η^+ and a second edge associated with $h_{i_\eta} < 0$ to β_η^-. If R is not ordered, then these conditions are to be replaced with $h_{i_\eta} = 0$ and $h_{i_\eta} \neq 0$.

- Every computation node η has exactly one outgoing edge to the node β_η. With each computation node η a map $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ out of `constant`, `project`, `copy`, `add`, `subtract`, `multiply`, `ifelse`, and `compose` is associated, which are going to be defined in Definition 4. If R is a field, then the map `divide` is allowed for computation nodes. All these operators are essentially finitely parametrized.

In order to define the different possible operations $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ for a computation node η, we will introduce the following notation to denote the replacement of a single component of a tuple.

Notation 3 (Replacing components of a tuple). Let $h = (h_1, \ldots, h_n)$ be a n-tuple, then $h[_i\!\leftarrow g]$ denotes the n-tuple that is obtained from h with the i-th component *replaced* with g, i.e.

$$(h[_i\!\leftarrow g])_k = \begin{cases} g & \text{if } k = i \\ h_k & \text{otherwise.} \end{cases}$$

This notation of a replacement in one component will be used to define the different possibilities for a map $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ of a computation node η. Each map $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ will modify only a single register of the state space.

Definition 4 (The different operations of a DPM). Let $i, j, k, l \in [n]$, $\alpha \in R$ and $m \in [\dim D_k]$. Let $\pi_m : D_k \to R$ denote the projection on the m-th component of D_k. The following maps $\mathcal{S}_M \to \mathcal{S}_M$, which modify the k-th entry in the state space, are only defined and applicable if $D_i = D_j = D_l \supset D_k$:

$$\begin{aligned}
\texttt{constant}(\alpha, k) &: \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow (\boldsymbol{\alpha} : D_k \to R, x \mapsto \alpha)] \\
\texttt{project}(m, k) &: \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow (\pi_m : D_k \to R)] \\
\texttt{copy}(i, k) &: \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow h_i] \\
\texttt{add}(i, j, k) &: \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow h_i + h_j] \\
\texttt{subtract}(i, j, k) &: \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow h_i - h_j] \\
\texttt{multiply}(i, j, k) &: \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow h_i \cdot h_j]
\end{aligned}$$

In the case of R being a field, there is an additional operator

$$\texttt{divide}(i, j, k) : \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k\!\leftarrow h_i / h_j]$$

In order to avoid undefined behavior, `div` is defined to evaluate to zero at the corresponding points where the denominator would become zero.

If the domain D_k is non-trivial, $\boldsymbol{\alpha}$ will be considered as a *constant function* and the operators $+, -, \cdot$ and $/$ will be interpreted *point-wise*.

For more complex computations on a non-trivial domain, often the computation of conditional values in parallel is required. Therefore, we also allow the following operator:

$$\texttt{ifelse}(l, i, j, k) : \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k \leftarrow \operatorname{if}(h_l \geqslant 0, h_i, h_j)]$$

where

$$\operatorname{if}(h_l \geqslant 0, h_i, h_j) : D_k \to R, x \mapsto \begin{cases} h_i(x) & \text{if } h_l(x) \geqslant 0 \\ h_j(x) & \text{if } h_l(x) < 0 \end{cases}.$$

In the case if R is not ordered, then $\geqslant$ and $<$ have to be replaced by $=$ and $\neq$. The application of $\texttt{ifelse}(l, i, j, k)$ differs from the branching nodes in two different ways: Firstly, it can run in parallel on an entire, possible non-trivial, domain $D \in \mathcal{D}$ and secondly, the values of h_l can only affect the values of h_k stored in the registers. But the values of h_k have no (direct) influence on the currently active node.

A further important possible type of $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ is the composition (which later will be interpreted as reading from the storage at a given location):

$$\texttt{compose}(\{i_1, \dots, i_l\}, j, k) : \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[_k \leftarrow h_j \circ (h_{i_1}, \dots, h_{i_l})],$$

where

$$\begin{aligned} &h_j \circ (h_{i_1}, \dots, h_{i_l}) : D_{i_1} \times \cdots \times D_{i_l} \to R, \\ &\qquad (x_1, \dots, x_l) \mapsto \begin{cases} h_j(h_{i_1}(x_1), \dots, h_{i_l}(x_l)) & \text{if } (h_{i_1}(x_1), \dots, h_{i_l}(x_l)) \in D_j \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

This operator requires that $D_k \subset D_{i_1} \times \cdots \times D_{i_l}$. Again, if the value of $(h_{i_1}, \dots, h_{i_l})$, which in general is a value in R^l, lies outside of the domain D_j, `compose` will evaluate to zero at these points.

The operation `compose` can be used both for composition of functions and evaluation: The composition $f \circ a$ with $a : * \to R$ corresponds to the evaluation $f(a)$. Later in pseudo-code, we will use the notation $f[a]$ to clarify that this does not trigger a real evaluation, but is to be interpreted as reading the value of the register f at the "index" a. This operator, together with the assumption that it also has a constant running time, gives a lot of additional strength to the model.

Remark 5 (Relation to finite-dimensional BSS). If $\mathcal{D} = \{*\}$, then only constants can be stored and the preceding definitions of a DPM coincide with the definition of a *finite-dimensional BSS machine over R* (Blum et al., 1989), where computation nodes η attach either polynomials or (in the case if R is a field) rational maps $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$.

Proof. In the case $\mathcal{D} = \{*\}$, for any node η the operation g_η = `compose` is not applicable. Any node η with g_η = `ifelse` can be replaced by three nodes, where one node does the branching based on a single value and the other two nodes perform the computation for each of the branches. Every remaining operation is polynomial (or rational if R is a field). Furthermore, every polynomial (or rational) map can be built from a sequence of elementary operations. In this case, without loss of generality, one can associate with every computation node η a polynomial (or rational) map $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ and this definition coincides with the definition of a *finite-dimensional machine over R*. □

Remark 6 (finite dimensional vs uniform). Blum et al. (1989) extend their *finite-dimensional* model to a *uniform* model. This uniform model has an infinite number of registers and there is a *"fifth node"* which allows to read from a register with an index computed at running time. This formalization enables algorithms which solve problems with *inputs of arbitrarily large size* and the model becomes *universal*, i.e. there is a universal BSS machine that can simulate any other uniform BSS machine that is specified through its input.

We will avoid introducing such a uniform model for DPMs because once there is a register with domain $D \in \mathcal{D}$ such that $\mathbb{N} \hookrightarrow D$, random access is already possible in our "finite-dimensional" DPMs, which have a fixed number of available registers. Algorithms on a DPM could use a register $\mathbb{N} \to R$ as an input. However, our model is not universal in the sense that there is a DPM machine that could simulate any other DPM machine. In order to do so, a uniform DPM model must be allowed to define the types of its registers at running time. Since we do not need this ability in this work, and this would make DPMs even more powerful abstract concept, rather far away from physical realizations, we omit such a construction here.

Furthermore, with our current "finite-dimensional" definition of a DPM, we can *control* the power of a DPM by specifying the set of allowed domains $\mathcal{D}$.

2.1.2. Computations of a DPM

So far, we have given an abstract definition of a DPM, which heuristically corresponds to the "program-code" of a given DPM (Later in Section 2.2 we will show how to translate DPMs into program-code and vice-versa). Now, we are going to explain how a DPM performs the computation. For that, we first define a *computing endomorphism*, which describes a single step of the computation of a DPM. Then we will define the (partial) *input-output* map of a DPM for those inputs where the DPM "halts". These definitions almost coincide with the definitions from Blum et al. (2012).

Definition 7 (computing endomorphism). We associate M with a natural *computing endomorphism* H. It describes a single step of the computation of a DPM with the nodes V:

$$H : V \times \mathcal{S}_M \to V \times \mathcal{S}_M$$

.

V can be decomposed as follows:

$$V = \{\eta_0\} \cup \mathcal{C} \cup \mathcal{B} \cup \mathcal{T}$$

where η_0 is the unique starting node, $\mathcal{C}$ are the computation nodes, $\mathcal{B}$ are the branch nodes and $\mathcal{T}$ are the output nodes. We will define a computing endomorphism H for each of the types of nodes separately.

The starting node does not modify the registers $h \in \mathcal{S}_M$ and the DPM proceeds with the unique next node β_{η_0}:

$$H(\eta_0, h) := (\beta_{\eta_0}, h)$$

Any computation nodes $\eta \in \mathcal{C}$ has also an unique next node β_η and an associated computation $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$. We set

$$H(\eta, h) := (\beta_\eta, g_\eta(h)) \text{ for } \eta \in \mathcal{C}$$

Any branching node $\eta \in \mathcal{B}$ does not modify the registers. However, based on the value of h_{i_η} either node β^+_η or β^-_η is "visited" next. If R is ordered then we set

$$H(\eta, h) := \begin{cases} (\beta^+_\eta, h) & \text{if } h_{i_\eta} \geqslant 0 \\ (\beta^-_\eta, h) & \text{if } h_{i_\eta} < 0 \end{cases} \text{ for } \eta \in \mathcal{B}.$$

If there is no order on R, then

$$H(\eta, h) := \begin{cases} (\beta^+_\eta, h) & \text{if } h_{i_\eta} = 0 \\ (\beta^-_\eta, h) & \text{if } h_{i_\eta} \neq 0 \end{cases} \text{ for } \eta \in \mathcal{B}.$$

In our applications, H is not necessarily defined for the output nodes $\eta \in \mathcal{T}$. However, to make $H : V \times \mathcal{S}_M \to V \times \mathcal{S}_M$ defined everywhere, we set

$$H(\eta, h) := (\eta, h) \text{ for } \eta \in \mathcal{T}$$

Iterating the endomorphism H indeed corresponds to running code step-by-step on the machine.

We will now define a computation path and an input-output map of a DPM.

Definition 8 (state trajectory, computation path, input-output map, halting set, halting time). Let the input $x \in \mathcal{I}_M$ be given. It induces an *initial point* $z^0 = (\eta_0, I(x)) \in V \times S_M$ and the iteration of the computing endomorphism H on z^0 generates a sequence

$$z^0, z^1 = H(z^0), z^2 = H(z^1), \dots .$$

This sequence corresponds to the computation of the machine given the input $x \in \mathcal{I}_M$. Let $\pi_V : V \times \mathcal{S}_M \to V$ and $\pi_{\mathcal{S}_M} : V \times \mathcal{S}_M \to \mathcal{S}_M$ be the projections on the nodes and the states (i.e. the register values). From that we define the sequences $\eta^k = \pi_V(z^k)$ and $x^k = \pi_{\mathcal{S}_M}(z^k)$. The later sequence in $\mathcal{S}_M$

$$x^0 = I(x), x^1 = \pi_{\mathcal{S}_M}(H(I(x), \eta_0)), x^2 = \pi_{\mathcal{S}_M}(H^2(I(x), \eta_0)) \dots$$

is called the *state trajectory* of x and the first sequence in V

$$\eta^0 = \eta_0, \eta^1 = \pi_V(H(I(x), \eta_0)), \eta^2 = \pi_V(H^2(I(x), \eta_0)) \dots$$

is called the *traversed computation path* γ_x.

The computation path might or might not eventually reach an output node. If it does not reach an output node, we say that M does not halt. Otherwise, there must be a minimal T such that η^T is an output node. We define such a T as the *halting time* $T_M(x)$. The *halting set* $\Omega_M \subset \mathcal{I}_M$ of M is the set of all input values $x \in \mathcal{I}_M$ that induce a computation path eventually ending in an output node.

The input-output map $\phi_M : \mathcal{I}_M \to \mathcal{O}_M$ is a partial map defined on Ω_M via

$$\phi_M(x) = O_{\eta^{T_M(x)}}(x^{T_M(x)}),$$

where the sequences x^k and η^k are induced as above. In words, we first apply the input map $I : \mathcal{I}_M \to \mathcal{S}_M$ to compute the initial registers and we think of the unique starting node η_0 as active. We apply the computing endomorphism $H : V \times \mathcal{S}_M \to V \times \mathcal{S}_M$ Iteratively and obtain new active nodes and values for the registers. Eventually, we will reach an output node η. There we apply the corresponding output map $O_\eta : \mathcal{S}_M \to \mathcal{O}_M$ on the current state registers and obtain the value for $\phi_M(x)$.

This is the formal definition of Model 2 introduced in the beginning of this Chapter. In the next section we will show that this Model is equivalent to Model 1, and several examples of DPMs will follow.

2.2. Programming DPMs

In this section, we show how a DPM can be "programmed" in pseudo-code. The first purpose of this section is to set the basics of describing DPMs in a more accessible way that is similar to programming in existing high-level languages. With this, we settle an equivalent description of concrete DPMs, which lets us easily express various applications of DPMs.

The second purpose of this section is to emphasize the proximity of the previously defined abstract model with real existing hardware. In particular, DPMs can be considered as an idealization and generalization of GPU fragment shader programs. Several of the algorithms, which follow in this text, and are designed for the theoretical DPM-model, can be transferred to real hardware. The pseudo-code representation we are going to develop is close to existing programming paradigms. However, our theoretical consideration of an idealized model might lead to different thinking processes. It can help to invent algorithms that are usable in reality and are beyond the toolbox of commonly employed programming techniques.

In the following, we describe several steps how the description of a DPM can be changed, at the preservation of its calculated function and changing its running time at most up to a constant factor. To achieve this, we are allowed to combine a fixed finite number of operations into a single operation that is counted as a single step in the runtime. In addition, we demand that the computing power does not increase with the introduction of new concepts. Each step can be considered as a small Lemma stating that the presented representation of DPMs is equivalent to the previous representation. By equivalent, we mean that every "program" of one representation can be converted in another program of the other representation and vice versa such that the running time differs only by a constant (which might depend on the program, but not on the input). All in all, we will get an equivalent description to DPMs that will allow us to program DPMs easily and without loss of precision in readable pseudocode.

2.2.1. Step 1: Rename variables and encode tuples.

First, the names $h_1, \dots, h_n$ of the program registers, which we are going to interpret as variables, can be replaced by arbitrary names like a, b, n, var, counter etc. Names like f or g might denote functions, which are in our context registers $h_k : D_k \to R$ with non-trivial domains $D_k \neq *$.

Second, several variables/functions $a_1, a_2, \dots, a_k : D \to R$ of the same domain $D \in \mathcal{D}$ can be combined to a single variable/function $\boldsymbol{a} = (a_1, a_2, \dots, a_k) : D \to R^k$. Technically, this is a renaming process. The entries of these encoded tuples can still be accessed and we set $\boldsymbol{a}_i = a_i$.

Even more, when specifying a DPM over the ring $R = \mathbb{R}$, a complex number $z \in \mathbb{C}$ can be interpreted as a pair $z = (z_1, z_2) \in R^2$ (more precisely, as a function $* \to R^2$) and can be stored by any DPM over R having at least two registers.

The description of the `comp`-operator often becomes simpler after combining several variables into one tuple: In Definition 4 $\texttt{comp}(\{i_1, \dots, i_l\}, j, k)$ computes the function

$$(x_1, \dots, x_l) \mapsto h_j(h_{i_1}(x_1), \dots, h_{i_l}(x_l))$$

for $h_{i_j} : D_{i_j} \to R$ and saves it to the register h_k. With a notation that allows for tuples, `comp` can be considered as an operator that computes $f \circ g$ where g represents the complicated function $(g_1, \dots, g_n) = (h_{i_1}, \dots, h_{i_l}) : D_{i_1} \times \cdots \times D_{i_l} \to R^l$ and f is another function with a possible multi-dimensional co-domain. If g has the domain $*$, g can be considered as a tuple of constants and we will write $f[g]$ instead, since the composition $f \circ g$ is more like an evaluation of the function f at the *point* g.

2.2.2. Step 2: Allow for concatenated computation nodes.

If a and b are computation nodes with $\beta_a = b$, then they can be merged to a single computation node provided that there is no third node $c \in V \setminus \{a\}$ with $\beta_c = b$: The nodes a and b can be replaced by a computation node d, with $\beta_d := \beta_b$, that computes the composed function $g_d := g_b \circ g_a : \mathcal{S}_M \to \mathcal{S}_M$, where $g_a, g_b : \mathcal{S}_M \to \mathcal{S}_M$ are the functions that are associated with a and b. g_d is possibly not listed in Definition 4. However, if one allows finite compositions of functions associated with the computation nodes, one obtains an equivalent concept that allows for more compact representations. After the replacement, all edges that lead to a within the computation graph have to be replaced by edges that lead to d.

Using the composition of such functions as computation nodes can be, in particular, used to recompute the value of entire tuples that were introduced in Step 1 within a single computation step.

2.2.3. Step 3: Transfer the “flowchart” of a DPM into goto-code.

Without loss of generality, for a given DPM with associated graph $G = (V, E)$ we can assume that:

(1) $V = \{\eta_0\} \cup \{1, \dots, N\}$

(2) $\eta_0 \in V$ is the unique starting node with no ingoing edges and its next node β_{η_0} is 1.

(3) Every computation node $n \in V$, either has $\beta_n = n + 1$ and g_n modifies a single variable/tuple; or $g_n = \mathrm{id} : \mathcal{S}_M \to \mathcal{S}_M$.

Requirement (1) is achieved by renaming the nodes. Property (2) again, can be obtained by reordering the nodes and replacing all edges that lead to η_0 by edges that lead to 1, which does not change the computed function, because by definition η_0 does not change the registers. In order to transfer a DPM into a machine that fulfills (3), possibly additional intermediate computation nodes that do not modify the registers, but only perform a "jump" to another node, have to be introduced.

These assumptions on the graph $G = (V, E)$ of the DPM give rise to a natural way to represent a DPM as pseudo-code (and vice-versa): Each node $n \in V \setminus \{\eta_0\}$ corresponds to the n-th line of the pseudo-code. Since $\beta_{\eta_0} = 1$, the interpretation starts with the first line. Every line contains one of the following operations:

- **goto** k for some line number $k \in \{1, \dots, N\}$.
- $a \leftarrow \langle \mathit{operation} \rangle$ for some variable (or tuple) a and $\langle \mathit{operation} \rangle$ corresponds to a single (component-wise) operation of Definition 4 and, depending on the operator, applied on some other variables (tuples). This line is only allowed if the occurring variables (or tuples) exist and have a type out of $D \to R^k$ some $D \in \mathcal{D}$, $k \in \mathbb{N}$ which makes the assignment well-defined.
- **if**$(a \geqslant 0)$ **goto** k **else goto** m for some variable $a : * \to R$ and line numbers k and m.
- **Return** a for some variable/tuple a.

goto k corresponds to the computaion nodes η that have $g_\eta = \mathrm{id} : \mathcal{S}_M \to \mathcal{S}_M$ and $\beta_\eta = k$.

If the n-th line of the code is $a \leftarrow \langle \mathit{operation} \rangle$, then the line represents the computation node n and $\langle \mathit{operation} \rangle$ encodes the map $g_n : \mathcal{S}_M \to \mathcal{S}_M$, which will change the register a. After executing this line, the DPM continues the computation from the unique next node $\beta_n = n + 1$, which corresponds to the execution of the next line. If it is not clear from the context, the type of the variable v has to be specified the first time it appears, i.e. the domain $D \in \mathcal{D}$ and (if a is a tuple) a number $k \in \mathbb{N}$ such that $a : D \to R^k$.

A line **if**$(a \geqslant 0)$ **goto** k **else goto** m corresponds to a branch node that tests on a and has $\beta_\eta^+ = k$ and $\beta_\eta^- = m$. The else-branch can be omitted if $\beta_\eta^- = \eta + 1$.

Return a corresponds to the output map $O_\eta : \mathcal{S}_M \to \mathcal{O}_M$ for some node η, which is a projection of a specified set of components of $\mathcal{O}_M$.

The input map will be specified by a set of variables that will be set in the beginning. All other variables in the register will be initialized to zero, which corresponds to the embedding $I : \mathcal{I}_M \to \mathcal{S}_M$.

As a short example for a DPM-goto-program over the ring $\mathbb{R}$, consider Algorithm 1, which computes the polynomial $p(z) = \sum_{k=0}^{n} z^k$. In order to distinguish functions $\mathbb{R} \to \mathbb{R}$ from constants $* \to \mathbb{R}$ we used **bold** letters for functions.

Algorithm 1: A DPM-goto-program over $\mathbb{R}$ with $\mathcal{D} = \{*, \mathbb{R}\}$ that computes the polynomial $p(z) = \sum_{k=0}^{n} z^k$ everywhere

```
Input: $n \in \mathbb{N}$ (Note: This will initiate a register $n : * \to \mathbb{R}$)
Output: The $\mathbb{R} \to \mathbb{R}$-function $z \mapsto \sum_{k=0}^{n} z^k$
1 $\boldsymbol{z} \leftarrow \pi_1 : \mathbb{R} \to \mathbb{R}$
2 $\boldsymbol{p} \leftarrow \mathbf{0} : \mathbb{R} \to \mathbb{R}$
3 $one \leftarrow 1$
4 $\boldsymbol{zk} \leftarrow \mathbf{1} : \mathbb{R} \to \mathbb{R}$
5 $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{zk}$
6 $\boldsymbol{zk} \leftarrow \boldsymbol{zk} \cdot \boldsymbol{z}$
7 $n \leftarrow n - one$
8 if $n \geqslant 0$ then goto 5
9 return $\boldsymbol{p}$
```

2.2.4. Step 4: Use expressions containing intermediate values

Registers are often needed to compute intermediate values, and these registers are not used at any other places. To reduce the number of statements and increase the readability, we allow composed expressions. Auxiliary variables have to be introduced for translating these composed expressions to a representation of Step 3. We can still count the number of evaluated lines as the running time. This does not change the asymptotic running time because only a finite number of elementary-operations are merged into a single operation.

Formally, we allow the following grammar for such a program that has to (at least) fulfill the rule $\langle program \rangle$:

$\langle nonzero\text{-}digit \rangle$ ::= 1 | 2 | 3 | 4 | 5 |6 | 7 | 8 | 9

$\langle fixed\text{-}integer \rangle$ ::= 0 | $\langle nonzero\text{-}digit \rangle$ | $\langle nonzero\text{-}digit \rangle\langle fixed\text{-}integer \rangle$

$\langle varname \rangle$ ::= $\langle string \rangle$ | $\langle varname \rangle_{\langle fixed\text{-}integer \rangle}$

$\langle$*list-of-expr*$\rangle$ ::= $\langle$*expr*$\rangle$ | $\langle$*expr*$\rangle$,$\langle$*list-of-expr*$\rangle$

$\langle$*expr*$\rangle$ ::= $\langle$*varname*$\rangle$
| ($\langle$*list-of-expr*$\rangle$)
| $\langle$*expr*$\rangle$ + $\langle$*expr*$\rangle$
| $\langle$*expr*$\rangle$ − $\langle$*expr*$\rangle$
| $\langle$*expr*$\rangle$ · $\langle$*expr*$\rangle$
| $\langle$*expr*$\rangle$ / $\langle$*expr*$\rangle$
| $\langle$*expr*$\rangle$ ∘ $\langle$*expr*$\rangle$
| $\boldsymbol{\alpha}$ for some $\alpha \in R^k$
| $\pi_{\langle\textit{fixed-integer}\rangle}$($\langle$*expr*$\rangle$)
| $\langle$*expr*$\rangle$[$\langle$*expr*$\rangle$]
| **if** $\langle$*expr*$\rangle$ $\geqslant 0$ **then** $\langle$*expr*$\rangle$ **else** $\langle$*expr*$\rangle$

$\langle$*statement*$\rangle$ ::= **goto** $\langle$*fixed-integer*$\rangle$
| $\langle$*statement*$\rangle$
$\langle$*statement*$\rangle$ (two statements separated by a new line)
| $\langle$*varname*$\rangle$ ← $\langle$*expr*$\rangle$ (if register: parallel assignment)
| **if**($\langle$*expr*$\rangle$ $\geqslant 0$) **goto** $\langle$*fixed-integer*$\rangle$
| **if**($\langle$*expr*$\rangle$ $\geqslant 0$) **goto** $\langle$*fixed-integer*$\rangle$ **else goto** $\langle$*fixed-integer*$\rangle$
| **Return** $\langle$*expr*$\rangle$

$\langle$*program*$\rangle$ ::= $\langle$*statement*$\rangle$

However, not every string build by the rule $\langle$*program*$\rangle$ is a valid program. Also here, the variables must exist and the types of the expressions must match.

2.2.5. Step 5: Use advanced control structures

Using **goto** k and the conditional jump **if**($\langle$*expr*$\rangle$ $\geqslant 0$) **goto** k, more advanced control structures can be modeled. For instance, the code fragment

```
while ⟨condition⟩ do
    ⟨loop-body⟩
```

can be considered as "syntactic sugar" for the fragment

```
A   if not ⟨condition⟩ then goto B
    ⟨loop-body⟩
    goto A
B
```

More advanced control structures such as **do**…**while**, **if**…**else**…, **for** and **repeat**, etc. and their equivalence programs containing only **goto** and **if**($\langle expr \rangle \geqslant 0$) **goto** k are presented in many standard lectures on theoretical computer science, for instance in Erk and Priese (2008). The command **goto** can be entirely avoided if these advanced control structures are used instead. Formally, we can include these control structure in the rules for $\langle statement \rangle$.

2.2.6. Step 6: An operation for point-wise parallel programs

To make the programming of the "parallel-part" of DPMs easier, we add here a programming paradigm, which allows for a description of (composed) point-wise operations as a small program, which will be evaluated in parallel. For that we introduce the following additional rule for $\langle statement \rangle$ in our syntax:

$\langle statement \rangle$ ::= …
| **compute** $\langle variable \rangle[\langle runningvar \rangle]$ **everywhere as** $\langle parallel\text{-}statement \rangle$

where

$\langle runningvar \rangle$ ::= $\langle varname \rangle$ | $(\langle list\text{-}of\text{-}varnames \rangle)$

$\langle list\text{-}of\text{-}varnames \rangle$ ::= $\langle varname \rangle$ | $\langle varname \rangle$,$\langle list\text{-}of\text{-}varnames \rangle$

$\langle parallel\text{-}statement \rangle$ ::= **Return** $\langle expr \rangle$
| $\langle parallel\text{-}statement \rangle$ $\langle parallel\text{-}statement \rangle$ (two parallel statements separated by a new line)
| $\langle varname \rangle \leftarrow \langle expr \rangle$ (only for registers $* \to R^k$)
| **if** $\langle expr \rangle \geqslant 0$ **then** $\langle parallel\text{-}statement \rangle$
| **if** $\langle expr \rangle \geqslant 0$ **then** $\langle parallel\text{-}statement \rangle$ **else** $\langle parallel\text{-}statement \rangle$
| **repeat** $\langle fixed\text{-}integer \rangle$ **times** $\langle parallel\text{-}statement \rangle$

For a register $\mathbf{v} : D \to R^k$, $D \subset R^n$, the statement

```
compute $\mathbf{v}[x_1, \ldots, x_n] : D \to R^k$ everywhere as
  $\langle parallel\text{-}statement \rangle$
```

means that the return value of ⟨*parallel-statement*⟩, which might depend on the values of the local "running variables" $x_1, \dots, x_n$, is point-wise computed at once and the result is stored to the register $v : D \to R$ such that $\boldsymbol{v}[x_1, \dots, x_n]$ has the value of ⟨*parallel-statement*⟩, which might contain $(x_1, \dots, x_n) \in D$ as running variable.

⟨*parallel-statement*⟩ allows less powerful expressions than ⟨*statement*⟩, because ⟨*parallel-statement*⟩ is not allowed to have conditional jumps or "intelligent loops". Furthermore, all assignments $a \leftarrow$ ⟨*expr*⟩ are only allowed to have atom variables/tuples $(* \to R^k)$ on the left side instead of functions. The variables that are assigned inside a ⟨*parallel-statement*⟩ are not allowed to occur at any other position of the program outside of the current compute-everywhere-block.

The running time of the entire statement will be counted as constant.

We will shortly explain how

```
compute $\boldsymbol{v}[x_1, \dots, x_n] : D \to R^k$ everywhere as
    ⟨parallel-statement⟩
```

can be transferred to an equivalent code that does not need this statement, and we obtain a form equivalent to the representation in Step 5.

1. For every $k \in [n]$ a line

   $\boldsymbol{x}_k \leftarrow \pi_k : D \to R$

   will be introduced. The "running variables" will be interpreted as a parallel array.

2. All **repeat**-loops are unrolled. The body of the repeat-loop is replicated a constant number of times. The asymptotic number of operations remains the same because the number of repetitions is a fixed number in the code.

3. Every "parallel variable" b, that is defined within ⟨*parallel-statement*⟩, is by definition $b : * \to R^k$. Without the compute-everywhere statement, we need a register that stores possibly different values for b based on different values of $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n) = \boldsymbol{x}$. For that we introduce a register $\boldsymbol{b} : D \to R^k$ and for any expression expr we use $\phi(\text{expr})$ to denote the transformed expression where every parallel variable b has been replaced by $\boldsymbol{b}$.

4. For every **if**-block k we provide two masks: $\text{ifmask}_k, \text{elsemask}_k : D \to \{0, 1\} \subset R$ that evaluate to 1 only for points making the corresponding branch active. To make the still following transformation rules simpler, we say that all the code is contained in a general **if**-block with number 0 and we execute

$$\text{ifmask}_0 \leftarrow 1 : D \to R$$
$$\text{elsemask}_0 \leftarrow 0 : D \to R$$

in the beginning.

For every **if**-block (which we will number by some $k \in \mathbb{N}_{\geqslant 1}$) and associated condition $c_k \geqslant 0$, that is contained in the if/else-branch of the "parent"-**if**-block with number p, we compute

$$\begin{aligned} \text{ifmask}_k &\leftarrow \text{[if/else]mask}_p \cdot \textbf{if}(\phi(c_k) \geqslant 0, 0, 1) \\ \text{elsemask}_k &\leftarrow \text{[if/else]mask}_p \cdot \textbf{if}(\phi(c_k) \geqslant 0, 1, 0) \end{aligned}$$

instead of the declaration of the k-th if-block.

5. Every assignment

$$a \leftarrow \text{expr}$$

is contained within the if/else-branch of a block l. By definition $a : * \to R^k$. We will instead perform a computation on the register $\boldsymbol{a} : D \to R^k$. The assignment is replaced with

$$\boldsymbol{a} \leftarrow \textbf{if}(\text{[if/else]mask}_l, \phi(\text{expr}), \boldsymbol{a}).$$

The replacement of the assignment will leave the register $\boldsymbol{a} : D \to R^k$ unchanged at those places where the current branch of block l is not active and recompute term at all other places.

6. In order to handle the returning of values, a helping variable running : $D \to \{0, 1\} \subset R$, indicating whether a value has not been returned yet for the given data-point $x \in D$. running is initialized with

$$\text{running} \leftarrow 1 : D \to R$$

Now any parallel statement

$$\textbf{Return } \text{expr},$$

contained in the if/else block l, will be replaced with

$$\begin{aligned} \boldsymbol{v} &\leftarrow \textbf{if}(\text{running} \cdot \text{[if/else]mask}_l, \phi(\text{expr}), \boldsymbol{v}) \\ \text{running} &\leftarrow \textbf{if}(\text{running} \cdot \text{[if/else]mask}_l, 0, \text{running}) \end{aligned}$$

As an example using the compute-everywhere-statement, an algorithm that computes the characteristic function of a disk with radius r is presented in Algorithm 2. Using the translation-rules from above, the same algorithm is translated to Algorithm 3, a representation of Step 5. In the future, we will keep using the shorter representations as in Algorithm 2.

We will consider the representation that we obtained in Step 6 as Model 1 that has been introduced at the beginning of this Chapter. Since all the steps describe mutually equivalent models, Model 1 and Model 2 express the same computational concept.

Algorithm 2: Computation of a disk for a DPM over $\mathbb{R}^2$ (high-level)

```
Input: $r \in \mathbb{R}$
Output: The characteristic function $\chi : \mathbb{R}^2 \to \{0, 1\}$ of a centered disk with radius $r$
compute $\chi[x, y], \chi : \mathbb{R}^2 \to \mathbb{R}$ everywhere as
    $v \leftarrow x \cdot x + y \cdot y - r \cdot r$
    if $v \geqslant 0$ then
        return 0
    else
        return 1
return $\chi$
```

Algorithm 3: Low-level translation of Algorithm 2

```
Input: $r \in \mathbb{R}$
Output: The characteristic function $\chi : \mathbb{R}^2 \to \{0, 1\}$ of a centered disk with radius $r$
$\boldsymbol{x} \leftarrow \pi_1 : \mathbb{R}^2 \to \mathbb{R}$
$\boldsymbol{y} \leftarrow \pi_2 : \mathbb{R}^2 \to \mathbb{R}$
$\text{ifmask}_0 \leftarrow 1 : \mathbb{R}^2 \to \mathbb{R}$
$\text{elseifmask}_0 \leftarrow 0 : \mathbb{R}^2 \to \mathbb{R}$
$\text{running} \leftarrow 1 : \mathbb{R}^2 \to \mathbb{R}$
$\boldsymbol{v} \leftarrow \textbf{if}(\text{ifmask}_0, \boldsymbol{x} \cdot \boldsymbol{x} + \boldsymbol{y} \cdot \boldsymbol{y} - r \cdot r, \boldsymbol{v})$
$\text{ifmask}_1 \leftarrow \text{ifmask}_0 \cdot \textbf{if}(\boldsymbol{v} \geqslant 0, 1, 0) : \mathbb{R}^2 \to \mathbb{R}$
$\text{elsemask}_1 \leftarrow \text{ifmask}_0 \cdot \textbf{if}(\boldsymbol{v} \geqslant 0, 0, 1) : \mathbb{R}^2 \to \mathbb{R}$
$\chi \leftarrow \textbf{if}(\text{running} \cdot \text{ifmask}_1, 0, \chi)$
$\text{running} \leftarrow \textbf{if}(\text{running} \cdot \text{ifmask}_1, 0, \text{running})$
$\chi \leftarrow \textbf{if}(\text{running} \cdot \text{elsemask}_1, 1, \chi)$
$\text{running} \leftarrow \textbf{if}(\text{running} \cdot \text{elsemask}_1, 0, \text{running})$
```

Chapter 3.

Mathematical algorithms suitable for DPMs

In this chapter, we undertake the first steps towards estimating the computational strength of DPMs.

We will give several examples of mathematical algorithms that can run on a DPM machine and utilize benefits in this model over classical ones. The examples will differ in the domains the DPM needs to access: The first example utilizes only finite domains. The second one requires an infinite but discrete domain. The last utilize infinite and continuous domains.

If we restrict $\mathcal{D}$ to contain finite domains only, then there is a straight-forward translation of the machine into a program for a conventional computer (aside from the possibility of real arithmetic): Registers with a finite set as domain can be considered as arrays. for-loops can replace every compute-everywhere-as-loops. Furthermore, these compute-everywhere-as loops over a finite domain are natural candidates for parallelization on many different architectures.

If the domains become infinite, then the programs can still be computed on a conventional computer, as we shall see in Section 5.1. In some cases, however, at the cost of an exponential increasing running time. If the domains are continuous, then the results might be approximated by shader code as described in Chapter 6.

In this chapter, we will sometimes use the term *computable* and consider running times. Here, with *computable*, we do not mean Turing-computable in the classical sense, but we mean that there is a finite-dimensional BSS machine over the given ring that computes the specified problem. According to that, the running time coincides with the asymptotic number of required arithmetic operations over that ring. If a function f is $O(1)$-computable, we mean that the evaluation of f takes constant time on a finite-dimensional BSS machine. Therefore, if f is $O(1)$ computable it is also possible to compute f on a DPM in parallel. These times for computation might differ from the computational time of Turing machines because for them, for instance, reading and writing $n \in \mathbb{Z}$ takes $\Omega(\log |n|)$ operations (due to the representation over a finite alphabet such as the binary representation). A BSS machine over the ring $\mathbb{Z}$, however, always needs constant time to process an integer.

3.1. Fast discrete Fourier transformation

We will first start with an example for a DPM machine that has only finite domains in $\mathcal{D}$: A modification of the Cooley Tukey algorithm (Cooley and Tukey, 1965) to compute the discrete Fourier transform (DFT) in $\Theta(\log n)$ steps on a DPM.

Let $x \in \mathbb{C}^N$ a vector of complex numbers. For simplicity, we assume that N is a power of two and that the indexation of vectors starts with 0. The DFT of x is defined as the vector $X \in \mathbb{C}^N$ defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{3.1}$$

where k is ranging from 0 to $N-1$. We will write $X = \mathrm{DFT}_N(x)$. The naive computation of the vector $X \in \mathbb{C}^n$ given $x \in \mathbb{C}^n$ by evaluating Equation (3.1) component-wise would take $\Theta(n^2)$ operations. The algorithm of Cooley and Tukey (1965) brings the computation down to $\Theta(n \log n)$ operations on a BSS machine. We will modify this algorithm to obtain an algorithm suitable for a DPM over $\mathcal{D} = \{*, [N]_0\}$ that requires only $\Theta(\log n)$ DPM operations.

We assume that $N > 1$ (In the trivial case $N = 1$ we have $X = x$). As a power of two, then N is even and we can use a formula from Cooley and Tukey (1965) to split the summation formula for X_k in the summands with even and with odd indices, and it will turn out, that X_k can be composed by using the DFTs of the even- and odd-indexed parts of x.

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk}}_{\mathrm{DFT}_{N/2}(\text{even-indexed part of } x)_k} + e^{-\frac{2\pi i}{N} k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk}}_{\mathrm{DFT}_{N/2}(\text{odd-indexed part of } x)_k} = E_k + e^{-\frac{2\pi i}{N} k} O_k. \tag{3.2}$$

This formula can be nicely used to compute the Discrete Fourier Transformation recursively. Inspired by the classical recursive approach, we will transfer this formula into an algorithm for a DPM over the ring $\mathbb{C}$ and the domains $\mathcal{D} = \{*, [N]_0\}$ that needs only $\Theta(\log n)$ steps.

Let us specify that, after s steps, a register $X : [N]_0 \to \mathbb{C}$ stores $\frac{N}{2^s}$ Fourier transferred vectors of certain sub-vectors of length 2^s of $x \in \mathbb{C}^n$. In order to derive the algorithm mathematically, we will write X^s for the value of X after s steps.

In the beginning ($s = 0$) the sub-vectors are interpreted of length $1 = 2^0$ and therefore the DFT of each of them will be the same and $X^0 = x$.

In the later steps, the $\sigma := \frac{N}{2^s}$ sub-vectors of x will not lie consecutively after each other but lie interlaced such that the first σ entries of x and X^s contain the first component of each of the σ vectors and the next σ entries contain the second component and so on. The distance between the components of one subvector within x and X^s will be always σ.

Let us specify this mathematically. The k-th component (indexation of a vector starts with 0) of the δ-th vector will correspond to the index $\sigma \cdot k + \delta$ and we demand the following formula to be invariant for every $\delta \in [\sigma]_0$:

$$(X^s_{\sigma\cdot k+\delta})_{k\in[2^s]_0} = \mathrm{DFT}_{2^s}\left((x_{\sigma\cdot k+\delta})_{k\in[2^s]_0}\right) \tag{3.3}$$

In step s with $\sigma = \frac{N}{2^s}$, where the DFTs of the σ subvectors of x are computed and stored in X^s, Equation (3.1) and Equation (3.2) can be transfered to the following equation:

$$\begin{aligned} X^s_{\sigma\cdot k+\delta} &= \sum_{n=0}^{2^s-1} x_{\sigma\cdot n+\delta} e^{-\frac{2\pi i}{2^s}nk} && (3.4)\\ &= \sum_{m=0}^{2^s/2-1} x_{\sigma\cdot 2m+\delta} e^{-\frac{2\pi i}{2^s/2}mk} + e^{-\frac{2\pi i}{2^s}k}\sum_{m=0}^{2^s/2-1} x_{\sigma\cdot(2m+1)+\delta} e^{-\frac{2\pi i}{2^s/2}mk} && (3.5)\\ &= \hat{E}_k + e^{-\frac{2\pi i}{N/\sigma}k}\hat{O}_k. && (3.6) \end{aligned}$$

where $\hat{E}_k$ and $\hat{O}_k$ are the k-th components of the “even”/“odd” indexed paths of the DFTs of the subvector $(x_{\sigma\cdot k+\delta})_{k\in[2^s]_0}$, which are already stored in $X^{s-1}_{\sigma\cdot(2\cdot k)+\delta}$ and $X^{s-1}_{\sigma\cdot(2\cdot k+1)+\delta}$ from the previous step. Alltogeter we obtain

$$X^s_{\sigma\cdot k+\delta} = X^{s-1}_{\sigma\cdot(2\cdot k)+\delta} + e^{-\frac{2\pi i}{N/\sigma}k} X^{s-1}_{\sigma\cdot(2\cdot k+1)+\delta} \quad , \tag{3.7}$$

which will be used in Algorithm 4.

If complex numbers are approximated with two real numbers, this algorithm can be naturally converted to shader-code of the GPU using a texture that has only one pixel in one direction. An implementation in CindyScript using colorplot is given in Applet ▷1. It is also possible to extend this algorithm to more dimensions (an implementation

Algorithm 4: The FFT for DPMs over $\mathbb{C}$ and $\mathcal{D} = \{*, D\}$, where $D = [N]_0$ and N is a power of 2

```
Input: A function $x : D \to \mathbb{C}$
Output: The (discrete) Fourier transformation $X : D \to \mathbb{C}$ of $x$
1  $X \leftarrow x$ // The DFT of a 1-vector is the vector
2  $\sigma \leftarrow \frac{N}{2}$ // distance between elements of vector
3  while $\sigma \geqslant 1$ do
4      compute $X[l], X : D \to \mathbb{C}$ everywhere as
5          $\delta \leftarrow \mod(l, \sigma)$ // shift
6          $k \leftarrow \frac{l-\delta}{\sigma}$ // local index within recursion
7          $\hat{E}_k \leftarrow X[\sigma \cdot (2 \cdot k) + \delta]$
8          $\hat{O}_k \leftarrow X[\sigma \cdot (2 \cdot k + 1) + \delta]$
9          return $\hat{E}_k + e^{\frac{-2 \cdot \pi \cdot i}{N/\sigma} \cdot k} \cdot \hat{O}_k$
10     $\sigma \leftarrow \frac{\sigma}{2}$
11 return $X$
```

for two-dimensions can be found in Applet ▷2), which opens the door for many real-world applications where fast image processing is required. For instance, the auto-correlation of an image, that means the image folded with itself, can be efficiently computed using the fast DFT. The auto-correlation gives powerful information that can be used to determine the periodicity of a periodic image.

3.2. (Possible infinite) cellular automata

Scholz (2014) gives a picturesque overview and description of the wide range of applications of two-dimensional cellular automata. Starting from the famous Game of Life, he shows, how cellular automata on a pixel-based grid can be used to build models of forest fires, traffic simulations, the predator-pray model (that locally approximates the Lotka-Volterra equation), the Lattice Gas model (which has the Navier Stokes equation as limit) and approximations to reaction-diffusion equations.

DPMs can simulate several cellular automata “naturally”. Unlike conventional computers, also cellular automata on an *infinite* domain can be reproduced on DPMs.

First, we will start with the formal definition of a cellular automaton. Definition 9 up to Definition 12 are from Hadeler and Müller (2017).

Definition 9. Consider a finitely generated group G with a finite set of generators $\sigma \subset G$.

The *Cayley graph* $\Gamma = \Gamma(G, \sigma)$ has the elements of G as vertices and two vertices $a, b \in G$ are connected iff there is a generator $s \in \sigma$ such that $a = bs$ or $b = as$. The neutral element $e \in G$ is called *the origin* of the Cayley graph. The vertices of Γ are called the cells of the grid.

For instance, the abelian group $G = \mathbb{Z}^2$ can be generated through the generators $\sigma = \{(0,1),(1,0)\}$. These generators will lead to a Cayley graph that looks like an infinite 2-dimensional grid. On the computer, it is possible to represent this Cayley graph. However, it is impossible to attach arbitrary (finite) data to each vertex of this infinite Cayley graph. For these purposes, often the graph $G = \mathbb{Z}_n \times \mathbb{Z}_m$ for some $n, m, \in \mathbb{N}$, a product of cyclic groups, is considered. If we use the same generators $\sigma = \{(0,1),(1,0)\}$, the Cayley graph becomes a finite grid identified with itself in both directions; The Cayley graph is a subset of the topological torus. The Cayley graph of $G = \mathbb{Z}_n \times \mathbb{Z}_m$ is often used on the computer because it is finite.

A cellular automaton assigns states to each cell/vertex of Γ. The states of the cells are iteratively computed according to a local rule. The rule defines the new state of a cell depending on its neighborhood. The definition of such neighborhoods follows here:

Definition 10. Let G be a finitely generated group and $\Gamma = \Gamma(G, \sigma)$ a Cayley graph. Let D_0 be a neighborhood of e (i.e., any finite set of vertices of Γ). The neighborhood of a point $g \in \Gamma$ is given by

$$D_g = gD_0$$

As an example, the *Moore neighborhood* in $G = \mathbb{Z}_n \times \mathbb{Z}_m$ for some $n, m, \in \mathbb{N} \cup \{\infty\}$ is defined as $D = \{-1, 0, 1\} \times \{-1, 0, 1\}$. The *von Neumann neighborhood* is defined as $D = \{(1,0),(0,1),(-1,0),(0,-1),(0,0)\}$ (Hadeler and Müller, 2017).

In the following definition, states are assigned to each cell of a Cayley graph:

Definition 11. Let $\Gamma(G, \sigma)$ be a Cayley graph. We define functions on Γ with values in a finite set E which we call the set of elementary or local states, or the "alphabet". A function

$$u : \Gamma \to E$$

is called a (global) state. The set $E^\Gamma = \{u : \Gamma \to E\}$ is the state space. If g is a vertex (cell) then $u(g) \in E$ is called the state of the cell.

Now we have all definitions to define a cellular automaton.

Definition 12. Let G be a finitely generated group, $\Gamma = \Gamma(G, \sigma)$ a Cayley graph. Let D_0 be a finite subset of Γ, called the neighborhood of the unit element $e \in G$ and E a finite set of local states. The function $f_0 : E^{D_0} \to E$ describes the local rules of the cellular automaton. We define

$$f : E^\Gamma \to E^\Gamma, \quad u \mapsto f(u)$$

with

$$f(u)(g) = f_0((u \circ g)|_{D_0}) \quad \text{for } g \in \Gamma.$$

The tuple (Γ, D_0, E, f_0) is called a cellular automaton.

Iterating such a local determined $f : E^\Gamma \to E^\Gamma$ simulates a cellular automaton. Cellular automata on finite groups can be easily simulated on a conventional computer: f can be computed pointwise on each of the finite number of cells. However, simulating cellular automatons on an infinite grid, in general, needs either some boundary conditions or periodicity. For instance, one can demand that the global state of a cellular automaton in the beginning has finite support. More precisely, one can demand that almost all cells are in a resting state $r \in E$, i.e $f_0(u_0) = r$ whenever $u_0 \in E^{D_0}$ fulfills $u_0(g) = r$ for all $g \in D_0$. If almost all cells are in a resting state, then also in consecutive steps, almost all cells will remain in such resting states (Hadeler and Müller, 2017). This restriction gives rise to a representation on a conventional computer.

Using a DPM makes it is also possible to simulate more general cellular automata on an infinite domain. However, there are still some restrictions on the finitely generated group G: It has to be *euclidean embeddable*, which we define here:

Definition 13. Consider a finitely generated group G with a finite set of generators $\sigma = \{s_1, \dots, s_k\} \subset G$. We call G *euclidean embeddable* if there is a $D \subset \mathbb{R}^n$ for some $n \in \mathbb{N}$, an injective function $\iota : G \to D$, and transformations $T_1, \dots, T_k : D \to D$ such that every $T_i : D \to D$ and its inverse $T_i^{-1} : D \to D$ are $O(1)$-computable and

$$\iota(s_i \cdot g) = T_i(\iota(g))$$

The transformations induced by the generators give rise to a geometric interpretation of the embedding $\iota : G \to D \subset \mathbb{R}^n$ and determine ι up to the value $\iota(e)$. Many groups are euclidean embedded once they have a geometric correspondence. For instance, a group G acting on a hyperbolic, euclidean or elliptic space X such that G acts freely on one orbit Gx for some $x \in X$ is euclidean embeddedable if the action is $O(1)$-computable. Since every such space X can be embedded into some $\mathbb{R}^n$, such an injection $\iota : G \to R^n$ is induced by taking the corresponding points of the orbit Gx embedded into the $\mathbb{R}^n$.

Since G is finitely generated over the generators $\sigma = \{s_1, \dots, s_k\}$, every group element $g \in G$ can be written as a finite product via elements of σ. Let $g = s_{i_1} \cdot \dots \cdot s_{i_l}$ Then

$$\iota(g) = \iota(s_{i_1} \cdot \dots \cdot s_{i_l} e) = (T_{i_1} \circ \dots \circ T_{i_l})\iota(e).$$

A minimal D can be always choosen by setting $D = \iota(G)$. G induces a transitive group action on $\iota(G)$ via the transformations T_i for each generator s_i. $\iota(G)$ can be considered as the orbit of $\iota(e)$ under G.

For instance, the group $\mathbb{Z}^2$ with the generators $s_1 = (0, 1)$ and $s_2 = (1, 0)$ is embeddable in $D = \mathbb{R}^2$ by using the identity $\mathrm{id} : \mathbb{Z}^2 \to \mathbb{Z}^2 \subset \mathbb{R}^2$ as ι and defining the transformations as shifts: $T_1(x, y) := (x + 1, y)$, $T_2(x, y) = (x, y + 1)$.

Unfortunately, we cannot expect that every finitely generated group is euclidean embeddable in our sense: Suppose it was, then we could also solve the word problem for any finitely generated group. The word problem in a group G with a finite set of generators $\sigma \subset G$ is the problem of deciding whether a word w over σ represents the identity in G (Epstein, 1992).

Proposition 14. There are finitely generated groups G that are not euclidean embeddable.

Proof. Suppose every group G was euclidean embeddable. Given a word $w = w_{i_1} \dots w_{i_l}$ that represents the group element $g = s_{i_1} \cdot \dots \cdot s_{i_l} e \in G$ we can compute

$$\iota(g) = \iota(s_{i_1} \cdot \dots \cdot s_{i_l} e) = (T_{i_1} \circ \dots \circ T_{i_l})\iota(e)$$

and check whether $\iota(g) = \iota(e)$. Since $\iota : G \to D$ is injective by definition this check is equivalent to the check whether $g = e$ and the word problem could be solved. However, Novikov (1954) showed that the word problem can not be decided for every finitely generated group G. □

We want to simulate cellular automata on DPMs. Cellular automata, in general, need an infinite amount of space.

Proposition 15. Let G be euclidean embedabble with $\sigma = \{s_1, \dots, s_k\} \subset G$ as set of generators and Cayley graph Γ and let (Γ, D_0, E, f_0) be a cellular automaton. Let $D = \iota(G) \cong \Gamma$. Then the cellular automaton (Γ, D_0, E, f_0) can be simulated on a DPM having access to the domain D.

Proof. For every $d \in D_0$ exists a representation $d = s_{i_1} \cdot \dots \cdot s_{i_l}$ and hence also computable transformations $t_d := T_{i_1} \circ \dots \circ T_{i_l}$. Then $t_d(\iota(e)) = \iota(d)$ and for every $g \in G$ also $t_d(\iota(g)) = \iota(dg)$. Then the iteration of the cellular automaton can be simulated as in Algorithm 5 on a D-DPM. Each step in the **for**-loop computes one iteration of $f : E^\Gamma \to E^\Gamma$. □

Algorithm 5: Simulating a cellular automaton on a DPM on D

```
Input: The initial states $u : \Gamma \cong D \to E$, the number of iterations $n \in \mathbb{N}$
Output: The state $u : \Gamma \cong D \to E$ after $n$ iterations
1 for $i = 1$ to $n$ do
2     compute $u[x], u : D \to E$ everywhere as
3         return $f_0(u(t_{d_1}(x)), \ldots, u(t_{d_m}(x)))$ where $D_0 = \{d_1, \ldots, d_m\}$
4 return $u$
```

3.3. Subset Sum problem in linear time

Given a set $x_1, \ldots, x_n \in R$ and a number $a \in R$, the Subset Sum problem (sometimes also refereed to as the *Knapsack Problem*) is the decision problem of checking whether there is a set $S \subset [n]$ with $\sum_{i \in S} x_i = a$.

The problem, considered over the ring $\mathbb{Z}$, where integers are encoded through their binary representation, is proven to be NP-complete for classical Turing machines. For general rings R, the Subset Sum problem can be decided by BSS machines. Blum et al. (2012) showed that if branching is done on equality checks only, then it is an intrinsic property of the problem to require at least exponential running time on any BSS machine. However, it is an open question, whether for ordered rings and branching on $<$, there is a BSS machine deciding the Subset Sum problem faster.

On a DPM, parallelism can be utilized. This enables to decide the Subset Sum problem *in linear time* on this architecture without using any order structure of R:

Algorithm 6: Solving the Subset Sum problem on a DPM over the ring R with domain R in linear time

```
Input: $n \in \mathbb{N}, x_1, \ldots, x_n \in R, a \in R$
Output: 1 or 0 if there is a set $S \subset [n]$ with $\sum_{i \in S} x_i = a$
1 compute $\sigma[x], \sigma : R \to \{0, 1\} \subset R$ everywhere as
2     if $x = 0$ then return 1 else return 0
3 for $k = 1$ to $n$ do
4     compute $\sigma : R \to \{0, 1\} \subset R$ everywhere as
5         return $\max(\sigma[x], \sigma[x - x_k])$
6 return $\sigma[a]$
```

By induction, we see that, after k iterations, the register σ in Algorithm 6 stores the characteristic function of $\{\sum_{i \in S} x_i \mid S \subset [k]\}$. Therefore, Line 6 returns the answer of the decision problem, namely $a \in \{\sum_{i \in S} x_i \mid S \subset [n]\}$.

Definition 16 (NP, $\mathrm{P}_{\mathrm{DPM}(\mathbb{Z})}$)**.** Let NP be the class of all decision problems that are non-deterministic polynomial for a Turing machine (NP in the classical complexity theory).

Furthermore, let $\mathrm{P}_{\mathrm{DPM}(\mathbb{Z})}$ be the class of all decision problems that can be solved by a DPM M over the ring $\mathbb{Z}$ and the domains $\mathcal{D} = \{*, \mathbb{Z}\}$ in polynomial time. More specifically, M should read only one register $input : \mathbb{Z} \to \{-1 \cong \square, 0, 1\}$ as input such that $n = \max\{|k| \mid input[k] \neq \square\} \in \mathbb{N}$ exists (This corresponds to the finite input of a Turing machine) and the running time of M should be polynomial in n.

Lemma 17.

$$\mathrm{NP} \subset \mathrm{P}_{\mathrm{DPM}(\mathbb{Z})}$$

Proof. Since Subset Sum is NP-complete in the classical theory, every problem of NP can be reduced to Subset Sum in polynomial time. A DPM with $\mathbb{Z} \in \mathcal{D}$ can simulate a Turing machine on this domain in the same asymptotic time. Therefore, the reductions to the Subset Sum problem can be done on a DPM as well in polynomial time. The numbers on the Turing Machines are wlog. encoded in their binary representation. Encoding them into elements of $\mathbb{Z}$ is possible in linear time in their length, which again is limited by the elapsed polynomial running time. Since, $\mathbb{Z} \in \mathcal{D}$, the DPM then can solve the obtained instance of the Subset Sum problem using Algorithm 6 in linear time and an answer for the original problem is obtained in polynomial time. □

3.4. Generation of objects with self-similarity

DPMs offer a suitable formalism for the description of algorithms that iteratively generate mathematical objects with self-similarities. Famous representatives of such objects are fractals or periodic patterns. An approximation by physical computers for IFSs and Kleinian groups has already been presented in Montag (2014). It is new to transfer these concepts to DPMs.

3.4.1. Iterated Function Systems

The following definition of an IFS and its limit set is paraphrased from Barnsley (1988). These definitions lay the mathematical foundations for IFS for complete metric space (X, d). For us, the case $X = \mathbb{R}^n$ with the norm-induced metric is of interest. In this setting, a DPM with domains $\mathcal{D} = \{*, X = \mathbb{R}^n\}$ can generate the limit sets of the IFS.

Definition 18 (hyperbolic iterated function system). A finite set of transformations w_1, …, $w_n : X \to X$ generates a *hyperbolic iterated function system* (IFS) on the complete metric space (X, d) if for every $i \in [n]$ the function $w_i : X \to X$ is a contraction.

An IFS acts on the space $(\mathcal{H}(X), h)$, which we are going to define here:

Definition 19 (The metric space $(\mathcal{H}(X), h)$). Let (X, d) be a metric space.

$$\mathcal{H}(X) := \{C \subset X \mid C \neq \emptyset \text{ compact}\}$$

denotes the set of all nonempty compact sets in X. $\mathcal{H}(X)$ can be equipped with the Hausdorff-metric $h(A, B) := \inf\{\epsilon \in \mathbb{R}_{>0} : A \subset B + \epsilon \wedge B \subset A + \epsilon\}$ where $A + \epsilon := \{x \in X \mid \exists a \in A : d(x, a) \leqslant \epsilon\}$.

Now we define the action of the IFS on the space of compact sets with Hausdorff metric.

Definition 20 (Hutchinson operator). For a given hyperbolic IFS with transformations $w_1, \ldots w_n$, the so-called *Hutchinson operator* is defined as:

$$\begin{aligned} W : \mathcal{H}(X) &\to \mathcal{H}(X) \\ C &\mapsto \bigcup_{i=0}^{n} w_i(C) \end{aligned}$$

This action turns to be out to be a contraction and we can apply Banach fixed-point theorem.

Theorem 21 (unique limit set of an IFS, from Barnsley (1988)). *An IFS defined over a complete metric space (X, d) induces a unique fixed-point $\Lambda \in \mathcal{H}(C)$ of W and for every $C \in \mathcal{H}(X)$ the convergence*

$$\lim_{n \to \infty} W^n(C) = \Lambda$$

with respect to the Hausdorff Metric h holds. We call Λ the limit set *of the IFS.*

Proof. We need two elementary ingredients that are proved in (Barnsley, 1988):

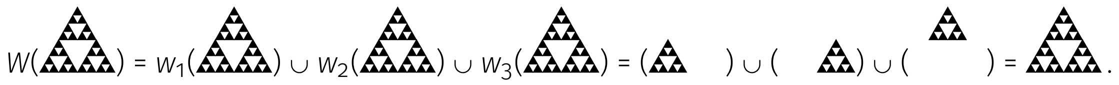
- Since all the functions $w_k : X \to X$ are contractions, also $W : \mathcal{H}(X) \to \mathcal{H}(X)$ is a contraction.

- The completeness of (X, d) carries over to $(\mathcal{H}(X), h)$.

The theorem then follows immediately by applying the Banach fixed-point theorem for the contractive map $W : \mathcal{H}(X) \to \mathcal{H}(X)$. □

In particular, any IFS on the Banach space $\mathbb{R}^n$ has an unique fixed point.

Example 22 (The Sierpinski triangle). On $X = \mathbb{R}^2$, consider the hyperbolic IFS with $w_1 : x \mapsto \frac{1}{2}x$, $w_2 : x \mapsto \frac{1}{2}x + (1, 0)$ and $w_3 : x \mapsto \frac{1}{2}x + (\frac{1}{2}, \frac{\sqrt{3}}{2})$. Then the Sierpinski triangle is a fixed point of the induced Hutchinson operator W, because



From Theorem 21 follows that the Sierpinski triangle is the unique fixed set for the defined hyperbolic IFS. Also, iterating the Hutchinson operator W on any nonempty compact set $C \subset \mathbb{R}^2$ gives rise to an arbitrary good approximation of the Sierpinski triangle.

Let an IFS over $X = \mathbb{R}^n$ and bijective affine transformations $w_1 : X \to X, \ldots, w_n : X \to X$ be given. It is straightforward to translate this machinery of Theorem 21 to a DPM with the domains $\mathcal{D} = \{*, X\}$ that performs the iterative applications of the Hutchinson operator. Any set $C \in \mathcal{H}(X)$ can be characterized by its characteristic function $\chi_C \in \{0, 1\}^X$ defined through

$$\chi_C(c) = \begin{cases} 1 & \text{if } c \in C \\ 0 & \text{if } c \notin C \end{cases}.$$

The Hutchinson operator $W : \mathcal{H}(X) \to \mathcal{H}(X)$ with $W(C) = \bigcup_{i=0}^{n} w_i(C)$ can be translated to $\tilde{W} : \{0, 1\}^X \to \{0, 1\}^X$ by setting

$$\left(\tilde{W}(\chi)\right)(p) := \max_{i \in [n]} \{\chi(w_i^{-1} p)\}.$$

This definition fulfills

$$\tilde{W}(\chi_C) = \max_{i \in [n]} \{\chi_C \circ w_i^{-1}\} = \max_{i \in [n]} \{\chi_{w_i(C)}\} = \chi_{\bigcup_{i=0}^{n} w_i(C)} = \chi_{W(C)}$$

Hence $\tilde{W}$ mimics the set operations of the Hutchinson operator on the level of characteristic functions. The DPM in Algorithm 7 iterates $\tilde{W}$ on its given input.

The IFS-theory is not restricted to sets. Barnsley (2006) extends the theory to measures and pictures. DPMs can also be used to compute the sequence of iteratively deformed measures (provided the measures have density functions; The registers of a DPM can store these density functions) and pictures (which also can be encoded in registers). In the second case, a single image with compact support is kept as "seed" and the orbit of the IFS acting on the picture is rendered. With this, even simple IFS, which for instance contain a single contractive transformation and thus has only a

Algorithm 7: Approximating the limit set of a hyperbolic IFS on a DPM over X

Input: The characteristic function $\chi : X \to \{0, 1\}$ of a non-empty compact set $C \subset X$, the number of iterations $m \in \mathbb{N}$ and a hyperbolic IFS on X through the inverse functions $w_1^{-1}, \dots, w_n^{-1}$

Output: The characteristic function of $W^m(C)$, which approximates the limit set $\Lambda = \lim_{m\to\infty} W^m(C)$ for big m where $W : \mathcal{H}(X) \to \mathcal{H}(X)$ is the Hutchinson operator for the IFS $w_1, \dots, w_n$

```
1 for $j = 1$ to $m$ do
2     compute $\chi[x], \chi : X \to \{0, 1\}$ everywhere as
3         $v \leftarrow 0$
4         for $i = 1$ to $n$ do
5             if $\chi[w_i^{-1}x] = 1$ then $v \leftarrow 1$
6         return $v$
7 return $\chi$
```

single point as the limit set, reveal more information such as a visible spiral (see for instance the second image of Figure 3.1d).

The requirements for the transformations to be contractions can also be dropped. Then there is no general mathematical justification for convergence anymore. However, this enables the visualization of wallpaper groups. Furthermore, on a compact set, after a finite number of iterations, a static image is generated.

A scheme for IFS that act on images can be described with Algorithm 8. Images are specified as functions $\chi : \mathbb{R}^2 \to \mathbb{R}^4$. The first three components of $\sigma(x)$ indicate the red, green, and blue-component of each image point x. The fourth component is used to specify the support of the image. Only on those spots $x \in \mathbb{R}^2$ where is $\sigma_4(x) > 0$ the image has some visible interpretation. At spots outside of the support, σ_4 attains 0, and the color is of no importance. This component can be interpreted as alpha-value for transparency. Given a set of arbitrary transformations, the orbit picture after a given number of iterations can be computed.

If the pictures are transferred such that some of them overlap at a spot, which image is to be “put in the front”? If the supports of the images transformed within Algorithm 8 overlap, the behavior of the algorithm depends on the given order of the transformations. By multiplying the fourth component of the colors obtained in Line 6 by a factor between 0 and 1, a more precise order can be specified. In the given Algorithm 8, it is only ensured that the seed picture σ where σ_4 attains 1 is always in front.

Algorithm 8: Generation of orbit-images for generalized IFS on a DPM over $\mathbb{R}^2$

```
Input: A seed-image $\sigma : \mathbb{R}^2 \to [0,1]^4$, the number of iterations $m \in \mathbb{N}$ and a
       generic IFS on $X$ through the inverse functions $w_1^{-1}, \dots, w_n^{-1}$
Output: The orbit picture after $m$ iterations
1 $\chi \leftarrow \sigma$
2 for $j = 1$ to $m$ do
3     compute $\chi[x], \chi : \mathbb{R}^2 \to [0,1]^4$ everywhere as
4         $b \leftarrow \sigma(x)$
5         for $i = 1$ to $n$ do
6             $o \leftarrow \chi[w_i^{-1}x]$
7             if $o_4(x) > b_4(x)$ then $b \leftarrow o$
8         return $b$
9 return $\chi$
```

3.4.2. Interactive realization on the GPU

If the registers of Algorithm 8 are replaced through textures of finite size, the algorithm can be efficiently approximated on a computer with GPU. A visually pleasing effect is obtained, if the approximated images χ are displayed in each intermediate step. Then by a growing process it can be observed, how the patterns and fractals are built up. In Applet ▷3, the loop in Line 2 of Algorithm 8 is not bounded and the transformations and the seed image σ can be modified in real time. Also, new transformations can be added. This gives a certain level of interactivity in constructing wallpaper groups and hyperbolic IFS.

It can be observed, how recently added strokes, which are drawn to the seed image σ, will be propagated through the entire system. In the example series of Figure 3.1, some very simple strokes have been drawn by the user to specify the seed-image. Upon that, the transformations are added one by one as semi-group generators. After each transformation is added, it takes some small number of the rendering steps in Line 3 of Algorithm 8 until the image on the compact screen becomes constant. This enhances the understanding of the mathematics of patterns.

For instance, in Figure 3.1a, a pattern with crystallographic group p1 (We use the naming convention of IUC) is generated by first drawing a small L-shape and then subsequently adding four semi-group generators, namely two linear independent translations and their inverse translations. The user can observe how the wallpaper pattern grows once the translations are added. Furthermore, it can immediately be seen that a set of four arbitrary chosen translations would generate a chaotic picture; The orbit of a single point would become dense. Therefore also snapping points have been introduced in Applet ▷3, which ease the specification of relations between the semi-group generators.

In Figure 3.1b and Figure 3.1c, the groups p3 and cm were generated. For all these Euclidean transformations the alpha-value of the transformed images has been slightly reduced, which damps the propagation of artifacts that occur due to the discretization of the registers.

In Figures 3.1d and 3.1e, affine contractive mapping were set to render the orbit of simple seed images through a hyperbolic IFS. This rendering technique reveals more information than only showing the limit set, which would, for instance, contain just a single point after a single contraction has been added.

Figure 3.1f shows a rendering where Euclidean and contractive transformations have been combined. The first added transformation is a reflection. The second one is a translation, which is orthogonal to the axis of reflection (otherwise the similarity would spread into two directions as in the third image of Figure 3.1c). The third transformation is a contraction by factor $\frac{1}{2}$ and center on the reflection axis. Slight deviations of these transformations would produce chaos, i.e., the orbit of a single point would become dense in an uncontrolled environment.

3.4.3. Kleinian groups

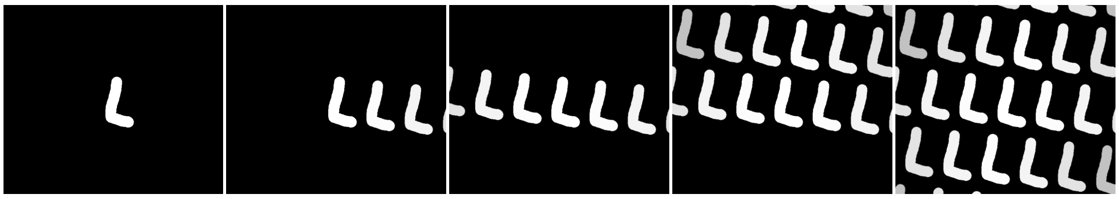
The DPM algorithms that have been introduced made use of only one register with a "parallel domain" so far. The following algorithm utilizes a register that contains a stack of several continuous parallel domains.

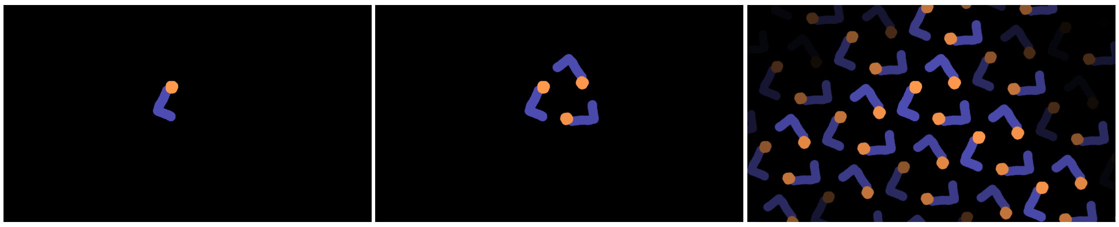
A *Kleinian group* Γ is a discrete group of Möbius transformations. Its *limit set* $\Lambda(\Gamma) \subset \hat{\mathbb{C}}$ is the set of all occurring accumulation points (Maskit, 2012).

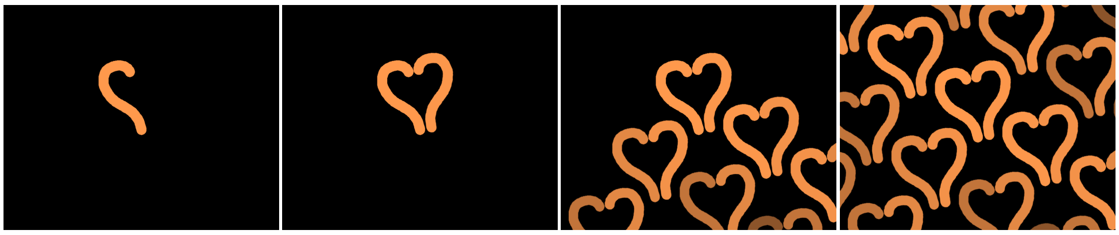
In Montag (2014), we have shown that if the language of geodesics in Γ is regular and $\Lambda(\Gamma) \neq \emptyset$, then the limit set $\Lambda(\Gamma)$ can be approximated in Hausdorff convergence as follows:

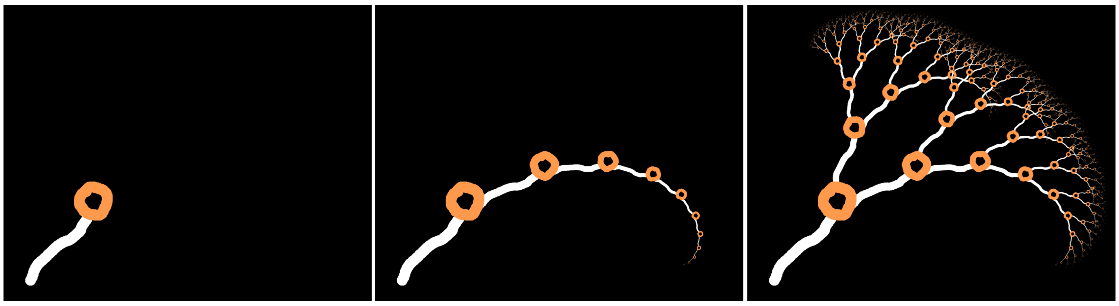
Theorem 23. *(from Montag (2014)) Let Γ be a Kleinian group with $\Lambda(\Gamma) \neq \emptyset$, let $C \subset \hat{\mathbb{C}} \setminus \Lambda(\Gamma)$ be non-empty compact and let Σ be a set of semi-group generators that is closed under inversion. Furthermore, suppose that there is a DFA $A = (Q, \Sigma, \delta, q_0, F)$*[1] *accepting the language $L(A) = \{w \in \Sigma^* \mid w \text{ geodesic in } \Gamma\}^R \cong \Gamma$.*

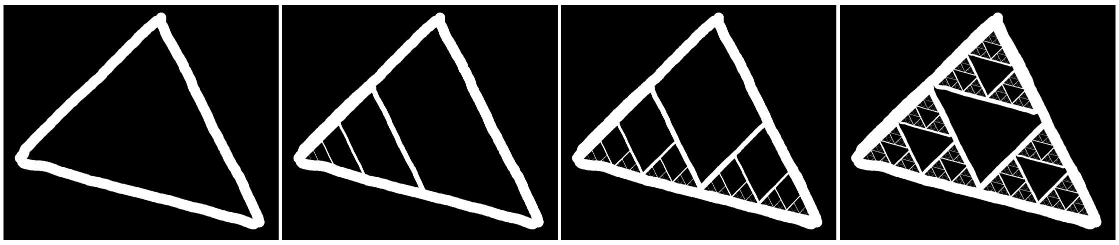
[1] i.e., A is a *deterministic finite automaton* over the finite alphabet Σ. A has a finite set Q of states, the transition function $\delta : Q \times \Sigma \to Q$, the starting state $q_0 \in Q$ and the finite set $F \subset Q$ of accepting states.


(a) Four semi-group generators are added one by one to generate the group p1.


(b) The group p3 is generated through a three fold rotation and a translation.


(c) The group cm is generated through a reflection and two translations.


(d) A two-generator hyperbolic IFS that renders a tree with trunk and branches.


(e) An IFS with a triangle as condensating set and three contractions added one after the other.

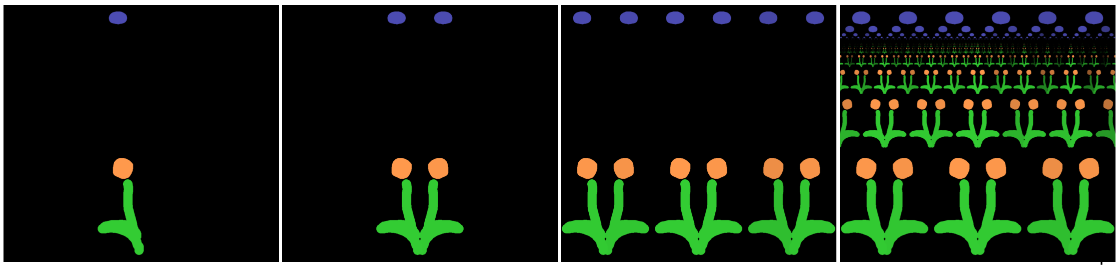

(f) A combination of a carefully chosen reflection, translatation and contraction.

Figure 3.1.: Sequences of screenshots from Applet ▷3. The left-most always contains the seed picture only. Then iteratively transformations are added and the orbit of the images through the generated semi-group is displayed.

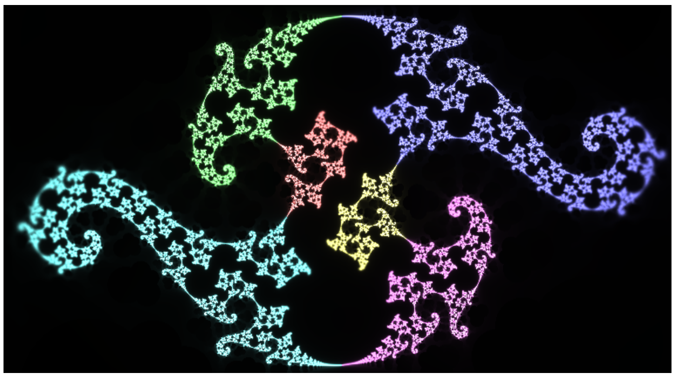


Figure 3.2.: Screenshot of Applet ▷4. The colors indicate the state-coordinate of the register.

Then the sequence $\left(W_q^n C\right)_{q\in Q, n\in\mathbb{N}}$ *of compact sets in* $\hat{\mathbb{C}}$ *which is recursively defined as*

$$W_q^0 C = \begin{cases} C & \text{if } q = q_0 \\ \varnothing & \text{if } q \neq q_0 \end{cases} \qquad W_q^{n+1} C = \bigcup_{\substack{\sigma\in\Sigma,\, p\in Q:\\ \delta(p,\sigma)=q}} \sigma W_p^n C$$

fulfills

$$\lim_{n\to\infty} \bigcup_{q\in F} W_q^n C = \Lambda(G)\,.$$

with respect to the Hausdorff-metric.

Theorem 23 can be translated to the DPM Algorithm 9. This DPM computes a sequence of sets that converge in Hausdorff-metric to the limit set of a Kleinian group. By definition of a DPM, the domains have to be subspaces of some $\mathbb{R}^n$. Here we omitted the technical detail of splitting the parallel domain $\hat{\mathbb{C}}$ into the domains $\mathbb{R}^2$ and the single point $*$.

A drawback of Theorem 23 and Algorithm 9 is that some knowledge of $\Lambda(\Gamma)$ is required in advance in order to choose a compact starting set $C \subset \hat{\mathbb{C}} \setminus \Lambda(\Gamma)$. Furthermore, this algorithm turns out not to be very stable to perturbations. However, in (Montag, 2014), we have shown that both these issues can be solved if instead of transforming sets, measures can be transformed. This in turn gives rise to a very similar DPM where the registers store density functions instead of characteristic functions. Applet ▷4 is a CindyGL based implementation of this variant of the algorithm (Screenshot in Figure 3.2). It renders the limit set of a free group of Möbius transformations chosen by “grandma’s recipe” from (Mumford et al., 2002).

Algorithm 9: A DPM approximating the limit set of a Kleinian group

Input: The characteristic function $\chi_C : \hat{\mathbb{C}} \to \{0,1\}$ of a non-empty compact set $C \subset \hat{\mathbb{C}} \setminus \Lambda(\Gamma)$, the number of iterations $m \in \mathbb{N}$, a Kleinian group Γ with the semi-group generators $\sigma_1, \dots \sigma_n$ and a DFA $A = (Q = \{q_0, \dots, q_{N-1}\}, \Sigma, \delta, q_0, F)$ accepting the language $L(A) = \{w \in \Sigma^* \mid w \text{ geodesic in } \Gamma\}^R \cong \Gamma$.

Output: The characteristic function of $\bigcup_{q \in F} W_q^m C$ approximating the limit set $\Lambda(\Gamma) = \lim_{m\to\infty} \bigcup_{q\in F} W_q^m C$, with W as in Theorem 23.

```
1  compute $\phi[k,x], \phi : [N]_0 \times \hat{\mathbb{C}} \to \{0,1\}$ everywhere as
2      if $k = 0$ then return $\chi_C[x]$
3      else return 0 // $\phi|_{\{k\}\times\hat{\mathbb{C}}}$ is characteristic function of $W^0_{q_k}C$.
4  for $j = 1$ to $m$ do
5      compute $\phi[k,x], \phi : [N]_0 \times \hat{\mathbb{C}} \to \{0,1\}$ everywhere as
6          $v \leftarrow 0$
7          for all $\sigma \in \Sigma$, $q_l \in Q$ with $\delta(q_l, \sigma) = q_k$ do
8              if $\phi[l, \sigma^{-1}x] = 1$ then $v \leftarrow 1$
9          return $v$
       // now, $\phi|_{\{k\}\times\hat{\mathbb{C}}}$ stores characteristic function of $W^j_{q_k}C$.
10 compute result$[x]$, result $: \hat{\mathbb{C}} \to \{0,1\}$ everywhere as
11     $v \leftarrow 0$
12     for $k = 0$ to $N-1$ do
13         if $\phi[k,x] = 1$ then $v \leftarrow 1$
14     return $v$
15 return result // characteristic function of $\bigcup_{q\in F} W_q^m C$
```

3.5. Iterating functions

The algorithm that we are going to develop will be both a useful instrument and as well the foundation for a proof of the fact, that DPMs are inherently faster than BSS.

Given a function $f : D \to D$ and a value $x_0 \in D$, let us iteratively define the sequence $(x)_n \in \mathbb{N}_0$ by $x_{n+1} = f(x_n)$. We assume that $f : D \to D$, given $x \in D$ is computable at unit time on a BSS, i.e. f is $O(1)$-computable. In this section we study, how the value x_n can be computed efficently.

3.5.1. Iterating functions on BSS machines

First, let us study this problem on conventional computers where we count each ring-operation as a unit-cost step. This corresponds to a computation on a BSS machine.

If one wants to compute the value of x_n, one can do this in $\Theta(n)$ operations by calculating the sequence $x_0, x_1 = f(x_0), \ldots x_n = f(x_{n-1})$ step by step. Can it be done faster? Or are there only special cases when it is possible to compute the n-th in $o(n)$ steps, i.e. asymptotically faster than $\Theta(n)$.

Case 1: D is finite

If D is finite, then during the iteration of $f : D \to D$, eventually there must be a cycle, i.e. two values $k < l$ such that $x_k = x_l$ (wlog. let k and l be minimal in having this property). By induction follows that $x_{k+n} = x_{l+n}$ for every $n \in \mathbb{N}_0$. Let $p = l - k$. Again by induction over n follows that for every $n, r \in N_0$ one has $x_{k+n \cdot p+r} = x_{k+r}$. Thus it suffices to pre-compute only the values $x_1 \ldots x_{k+p-1}$, because every index $n \geqslant k + p$ can be written in the form $n = k + l \cdot p + r$ with $r < p$ and thus $x_n = x_{k+l \cdot p+r} = x_{k+r}$, which is pre-computed. The asymptotic speed of computing x_n essentially depends on the speed of computing the modulus $n \mod p$, which however can be done in $\Theta(\log n) \subset o(n)$:

Proposition 24. There is a BSS machine over $\mathbb{R}$ that computes the fractional part

$$f : \mathbb{R}_{\geqslant 0} \to [0, 1), \quad x \mapsto x - \lfloor x \rfloor$$

in $\Theta(\log(x))$.

Proof. Use Algorithm 10.

Both while loops have exactly $\lceil \log_2 x \rceil$ iterations and therefore the entire running time is in $\Theta(\log(\lfloor x \rfloor)) = \Theta(\log(x))$. The first while loop computes $k = 2^{\lceil \log_2(n) \rceil}$ to be the smallest power of two greater than x. In the further steps of the algorithm, k will always remain an integral power of two. The variable x is only modified in Line 5. Therefore,

Algorithm 10: Computation of the fractional part on a BSS machine over $\mathbb{R}$ in $O(\log x)$

```
Input: $x \in \mathbb{R}_{\geqslant 0}$
Output: the fractional part $x \mapsto x - \lfloor x \rfloor \in [0, 1)$
1 $k \leftarrow 1$
2 while $k < x$ do
3     $k \leftarrow k \cdot 2$
4 while $x > 1$ do
5     $k \leftarrow \frac{k}{2}$
6     if $k \leqslant x$ then
7         $x \leftarrow x - k$
8 return $x$
```

the fractional part of the variable x always remains the same. Before executing Line 5, we will always have $k \geqslant x$, because we subtract the powers of two in decreasing order from x. The powers of two that are subtracted correspond to the ones in the binary composition of $\lfloor x \rfloor$. During the second while-loop, every power of two will be subtracted from x. □

Corollary 25. *The modulus* $\mod : \mathbb{N} \times \mathbb{N} \to \mathbb{N}, (n, p) \mapsto n \mod p$ *can be computed in* $\Theta(\log n)$*:*

Proof. Compute

$$n \mod p = p \left(\frac{n}{p} - \lfloor \frac{n}{p} \rfloor \right).$$

Algorithm 10 can compute $\frac{n}{p} - \lfloor \frac{n}{p} \rfloor$ in $\Theta(\log(\frac{n}{p})) = \Theta(\log n)$. All other arithmetic operations have unit cost on a DPM over $\mathbb{R}$. □

Case 2: D is infinite

If D becomes infinite, it is only possible in some cases to compute x_n efficiently on a computer that can do fast-arithmetic over D. One example is exponentiation: If $D = \mathbb{R}$ and $f(x) = a \cdot x$, then $x_n = a^n x_0$. Using fast-exponentation for a^n, the value $x_n = a^n x_0$ can be computed through $\Theta(\log(n))$ arithmetic operations. This example of a linear $f : \mathbb{R} \to \mathbb{R}$ can be also generalized to the k-dimensional case when $f : R^k \to R^k$ is a linear map. Each linear map can be represented by a matrix $A \in R^{k \times k}$, such that $f(x) = Ax$. The general solution is $x_n = A^n x_0$. The matrix A^n can be computed in $\Theta(k^3 \log n)$ arithmetic operations using fast exponentation then the matrix-vector product $A^n x_0$

can be computed in $\Theta(k^2)$. Since k is constant in the given problem, the total asymptotic running time is $\Theta(\log n)$.

One prominent example of this class of linear iteration in a finite-dimensional space is the computation of the Fibonacci numbers. The Fibonacci-numbers are defined as $F_0 = 0$, $F_1 = 1$ and $F_{n+1} = F_n + F_{n-1}$. If we define $x_n = (F_n, F_{n+1}) \in \mathbb{Z}^2$ and $f : \mathbb{Z}^2 \to \mathbb{Z}^2$ as $f(x_n) = f(F_n, F_{n+1}) = (F_{n+1}, F_n + F_{n+1}) = x_{n+1}$ then f is linear and can be represented through the following matrix product

$$f(x) = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} x$$

Thus,

$$F_n = \begin{pmatrix} 1 & 0 \end{pmatrix} x_n = \begin{pmatrix} 1 & 0 \end{pmatrix} f^n(x_0) = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

If fast exponentation by squaring is used, then the n-th Fibonacci number can be computed with $O(\log n)$ arithmetic operations over $\mathbb{Z}$.

However, there are $O(1)$-computable functions $f : D \to D$, for which there is no algorithm (on a BSS machine and hence as well for a Turing machine) to compute x_n in sublinear time. For instance, the "most simple" non-constant, non-linear function $f : \mathbb{R} \to \mathbb{R} : x \mapsto x^2$. As we shall see in the following Lemma, it is impossible to compute $x_n = f^n(x) = x^{2^n}$ in $o(n)$ arithmetic operations on a BSS machine:

Lemma 26. *There is no finite-dimensional BSS machine that can compute $p_n(x) = x^{2^n}$ in sublinear time in n.*

Proof. Assume that there was a finite-dimensional BSS machine that computes $p_n(x)$ in $o(n)$. Choose an $x_0 \in \mathbb{R}$ such that a neighborhood of x_0 has the same state trajectory. This is possible:

Since we assume that the running time is bounded, there is only a finite number of different state trajectories, and the domains with the same state trajectory are semialgebraic sets (Blum et al., 2012). The semialgebraic subsets of $\mathbb{R}$ are the finite unions of intervals. All domains of the same state trajectory form a partition of $\mathbb{R}$, so there must be at least one state-trajectory having a non-empty interior.

Within this neighborhood of x_0, no branching occurs and every register (locally) stores a rational function in x. Let us consider the degree $\deg r = \max\{\deg p, \deg q\}$ for rational functions $r(x) = \frac{p(x)}{q(x)}$. If two rational functions are added, subtracted, multiplied or divided, the degree of the result is bounded by the sum of the degree of the original rational functions. Thus, in the best case, the maximal local degree of all the register can double in one step on a BSS machine. Therefore, the maximum possible degree after $o(n)$ operations is $2^{o(n)}$, which, for sufficiently large n, will be smaller than 2^n, the degree of x^{2^n}. None of the registers could store the desired output. □

Remark 27. The previous result holds also for uniform BSS machines, because the length of the input is fixed (Blum et al., 2012). Also it can be extended to hold for the parallel BSS modell introduced by Blum et al. (2012).

3.5.2. Iterating functions on DPMs in sublinear time

Now let us apply DPMs over the domain D to solve this problem of computing the element x_n of the sequence generated by iterating an $O(1)$-computable function $f : D \to D$. It turns out, that such a DPM can always compute the iterate x_n in time $O(\log n)$. First we demonstrate an algorithm that computes $f^{2^k}(x_0) = x_{2^k}$ in $O(k)$:

Algorithm 11: Fast Iterated Function of f on DPMs over a domain D

```
  Input: $x \in D$, $n \in \mathbb{N}$, $f : D \to D$
  Output: The value $f^{2^n}(x)$ in $O(n)$
1 $g \leftarrow f : D \to D$ // Either by computation or by input
2 repeat $n$ times
3     $g \leftarrow g \circ g$
4 return $g[x]$
```

Instead of the step $g \leftarrow g \circ g$ in Line 3, we also could have written

compute $g[x], g : D \to D$ **everywhere as return** $g[g[x]]$.

After k steps of the repeat-loop, each with cost $O(1)$, the register labeled with g stores the function f^{2^k}. Since the evaluation in Line 3 by definition takes only constant time, $f^{2^n}(x)$ can be computed in $O(n)$.

Algorithm 11 could be considered as an other variant of the *pointer jumping* technique, which is the basis for several parallel algorithms that operate on pointer structures (Casanova et al., 2008). The novelity of our algorithm used in this context is, that the space is possibly non-discrete. It can be used, for instance, to iterate Newtons method faster or to visualize the Julia set very efficiently (An approximation on the GPU will be presented in Example 56).

To compute x_n with n not being a power of two, we can also provide a DPM-algorithm to compute x_n in $O(\log n)$, which is inspired by the fast exponentiation algorithm. Our formal definition of a DPM does not support a straightforward recursion[2], so we will

[2] However, a call-stack could be implemented provided there is a domain $D \in \mathcal{D}$ with $\mathbb{N} \subset D$

provide an implementation that has been designed by considering the binary decomposition of n. Using a variable k, which will be initialized as $2^{\lfloor \log_2(n) \rfloor}$, the greatest power of 2 less or equal to n, we use it to extract the binary decomposition n from the highest to the lowest bit. For each digit we double the number of the applications of the currently built function g. Whenever a bit is 1, we apply f one time more to the built g.

Algorithm 12: Fast Iterated Function of f on DPMs over a domain D

```
Input: $x \in D, n \in \mathbb{N}, f : D \to D$
Output: The value $f^n(x)$ in $O(\log n)$
$k \leftarrow 1$
while $k \leqslant n$ do // compute $k = 2^{\lfloor \log_2 n \rfloor}$
    $k \leftarrow k \cdot 2$
$g \leftarrow \mathrm{id} : D \to D$
while $k \geqslant 1$ do
    /* Go to the lower digit of the binary representation */
    $g \leftarrow g \circ g$
    if $k \leqslant n$ then // The corresponding bit is 1
        $g \leftarrow g \circ f$
        $n \leftarrow n - k$
    $k \leftarrow k/2$
return $g[x]$
```

Altogether, this algorithm will require $O(\log n)$ DPM operations.

If we apply this algorithm to $f(x) = x^2$, we immediately see that DPMs provided with domains large enough, are essentially faster than BSS machines.

Corollary 28. *There are problems that a DPM-machine with $\mathbb{R} \in \mathcal{D}$ can solve faster than any BSS machine.*

Proof. Consider the problem to compute the map $p_n(x) = x^{2^n}$. According to Lemma 26, any BSS machine needs at least linear time in n to compute $p_n(x)$. A DPM machine implementing Algorithm 12 and iterating the map $f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$, requires only $O(\log n)$ steps to compute $f^n(x) = p_n(x)$ and therefore is asymptotically faster than any BSS machine. □

Algorithm 12 has many further applications, for instance the Newton Method can be evaluated even faster. Any finite-dimensional BSS machine can be exponentially accelerated on a DPM with the statespace of the BSS as domain and the computing homomorphism of the BSS as iterated function (more details follow in Section 5.1).

Chapter 4.

DPMs as models for real-world systems

In general, a DPM can be considered as a single-threaded computer that can command an “army of arithmetic units”. A DPM has access to a possibly infinite number of agents that can execute the same instruction given by the single-threaded central machine. The performed tasks are not real Turing-complete programs but an a-priory fixed number of atomic operations. The agents can operate on different data elements. (SIMD)

In this sense, considering DPMs as a large collection of parallel processors (MIMD) is wrong, because the agents are not able to perform their computations based on their program flow/instruction stream.

In this chapter, we will introduce several real-world “computers” that can be modeled via our mathematical DPM model.

4.1. DPMs with finite domains $\mathcal{D}$ as parallel computers

In our definitions, there are very simple parameters to regulate the strength of DPMs: The allowed domains $\mathcal{D}$ and the ring R. Let us assume that $\mathcal{D}$ contains finite domains only and R can be meaningfully represented on a conventional computer (possible at the cost of approximation of real arithmetic through floating point arithmetic). In this case, the DPM can be computed on a conventional computer and it is straight-forward to accelerate it by (finite) parallelization.

Every register, that has now a finite set as domain, can be stored on the computer as an array. The compute-everywhere-as loops, to compute the contents of such registers, are natural candidates for a parallelization on many different architectures because the program that is evaluated on them is the same for each data point and there is no communication between the different threads. Herlihy and Shavit (2011) use the term "embarrassingly parallel" for such expressions.

Such DPMs with finite domains can be considered as instances of the PRAM (Parallel Random Access Machine) model (Casanova et al., 2008). The PRAM model has a shared central memory that can be accessed by various parallel units (PU), which execute synchronously the same algorithm. More precisely, such DPMs almost reflect the PRAM model with CREW (Concurrent Read Exclusive Write): During a given step of an algorithm, arbitrarily many PUs can read the value of a cell simultaneously while at most one PU can write a value to a cell. The PRAM model is built as well on various simplifying assumptions: As it is the case with DPMs, the cost of data communication is ignored, and any memory location can be accessed in unit time. However, its algorithms are closer to a physical realization, and the runtime complexity of algorithms in the PRAM gives a vague approximation of the asymptotic running time of real-world parallel computers.

Let p the number of available parallel processors. If $p \geqslant \max\{|D| \mid D \in \mathcal{D}\}$, then all parallel computations can be made concurrently. The running time of the DPM coincides with the running time of the PRAM.

4.2. A single-threaded computer with access to a GPU

Our DPM model also reflects a computer that has access to the fragment shader based GPU architecture. In Part II of the work we will extensively discuss how certain DPMs with two-dimensional domains can be realized through programs utilizing the fragment shader of the GPU. So we will give only a short overview here.

Shaders can be described as little programs which the GPU can execute massively in parallel. The program is the same for each data point and, in general, these shader programs are not Turing complete. A register with a two-dimensional domain on a DPM can be considered as the idealization of a texture that is stored on the GPU[1]. A DPM computation that assigns values to such a register corresponds to the computation of a fragment shader. The operations and branching of a DPM correspond to the current program state of the CPU.

The programmable shaders of a GPU are not limited for the graphics output of a computer. Programmable shaders can be used for general-purpose computations on the GPU (GPGPU). A shader can be used when the execution of the same program at an independent set of data points is required. Often numerical simulations can be built on such a computational scheme.

[1]Technically, four such $\mathbb{R}$-valued registers give rise to an RGBA-Texture where each channel of red, green, blue and alpha is represented via one register.

DPMs idealize GPUs in three decisive points:

- As the computations on a fragment shader on the GPU run in parallel, and we counted a DPM operation that modifies a register with a non-trivial domain as a single step, the running time of our DPM model corresponds to the running time of an idealized GPU that has a possible infinitely large number of parallel units.
- A DPM can be very powerful if it is specified to allow computations on the domains such as $\mathbb{R}$ or $\mathbb{R}^2$ instead on a finite set of pixels. DPMs assume that the size of an image can be arbitrary and that the textures are infinitely fine sampled. However, this idealization reflects attempts to use higher resolutions and can be considered as a continuous limit of these endeavors.
- DPMs enable real computations instead of floating point computations. DPMs are more like mathematical machines. However, when numerical algorithms are developed often thinking of an idealized model is the first step before transferring an algorithm to a physical computer.

However, studying DPMs in this context still can be justified. To understand possible applications of GPUs, DPMs provide not only tools but also an alternative way of thinking. The technical obstacles are abstracted away, and in the first phase of designing an algorithm that is suitable for a GPU, this can be helpful. An ideal algorithm on an ideal machine gives rise to an approximation algorithm on physical devices.

A philosophy of the same kind is applied to study, for instance, a numerical algorithm such as Newton's method. In the first step consists of the assumption that all the computations are precise. A theory of the speed of convergence in mathematical rigor is developed based on the assumption of real arithmetic. In a later step, the method is implemented on a machine with floating-point arithmetic. Tools such as the condition number will estimate the deviation from the idealized algorithm and the implemented algorithm.

4.3. Optical phenomena for parallel computation

Can optical phenomena be utilized to perform parallel computations? A single patch of an image could represent a single thread and local properties, for instance, having photons or ink in place or some local chemical state, could encode data associated to that thread. In this section, we will give two examples of "optical computers" and model their computation through DPMs.

A strength of the DPMs comes from the ability to perform computations on an infinitely fine resolution. Can this a concept transferred to the physical real-world? There are physical limitations in optical computing such as the Lateral resolution or the information density making too close objects indistinguishable (Aaronson, 2005). However, physical models can approximate DPM algorithms and might be useful in the communication of mathematics, with the same justification as “images” of fractals, having a limited resolution and therefore different from a mathematical fractal, serve their purpose in the communication of mathematics. In education and communication, showing the idea of a concept is often more valuable than the demonstration of a technically precise, but non-observable, instance.

In the following we give two examples of such “optical computers” that can be used in education and entertainment (“edutainment”). Mathematical ideas can be demonstrated by using these “optical computers”.

4.3.1. Computing by copying onto transparencies

Head (2009) solves the NP-complete boolean satisfiability problem (SAT) problem in linear time by copying[2] onto transparent plastic sheets (Of course, the copying machine and the sheets are mathematically idealized). In the following, we want to establish a DPM that performs equivalent operations.

The only operations that are used are copying on transparencies, aligning and overlaying the produced sheets on the photocopier, using *negative* photocopies and masking some areas of a transparent plastic sheet before copying. A black opaque box printed on a transparency indicates the truth value True and a transparent area indicates the truth value False. The logical operation OR can be “implemented” by overlaying and copying several transparencies with such boxes. Negation can be modeled by producing negative xerorx copies on transparent plastic sheets. With negation and the logical OR, de Morgan’s laws allow for the evaluation of arbitrary boolean formula. If these operations are applied to transparent plastic sheets with several transparent or opaque boxes, the operations can be considered to be carried out in *in parallel*.

Given n variables, Head (2009) first generates a truth-table containing 2^n rows by making n copies. (Here a physical limitation is that a very high resolution of the copying machine is required if n is large.). The truth table and its negative is then duplicated linearly often in the length of the SAT expression. By overlaying these transparencies according to the formula, the values of each of the clauses for each row of the truth table can be evaluated. If the values of each of the clauses for each row are negated

[2]also known as xeroxing.

Algorithm 13: A DPM implementation of “Parallel computing by xeroxing on transparencies” (Head, 2009) solving SAT in polynomial time

```
Input: A Boolean formula $\psi$ with $n$ variables, $m$ clauses each containing $k$ clauses
Output: 1 iff $\psi$ is satisfiable. Otherwise 0
// Generate the $n \times 2^n$ truth table containing $2^n$ rows.
table ← 0 : $[2 \cdot n] \times [2^n] \to \mathbb{F}_2$ // 0 is True, 1 is False
for $k = 0$ to $n$ do
    // duplicate the entire generated table below with one masked column
    compute table[$x, y$], table : $[2 \cdot n] \times [2^n] \to \mathbb{F}_2$ everywhere as
        if $y \leqslant 2^k$ then
            return table[$x, y$]
        if $x = k$ then
            return 1 // mask one column
        else
            return table[$x, y - 2^k$] // aligned copy
// generate a table with negated values right of the truth table
compute table[$x, y$], table : $[2 \cdot n] \times [2^n] \to \mathbb{F}_2$ everywhere as
    if $x \leqslant n$ then
        return table[$x, y$]
    else
        return 1 − table[$x - n, y$]
// The evaluation of $\psi$ for each row of the truth table
evaluations ← 1 : $[2^n] \to \mathbb{F}_2$ // 0 is False, 1 is True
for each clause c in $\psi$ do
    cval ← 1 : $[2^n] \to R$ // 0 is True, 1 is False
    for each literal $l$ ($l = x_m$ or $l = \neg x_{m-n}$) in $c$ do
        // overlay cval with column $m$ of the truth table
        compute cval[$y$], cval : $[2^n] \to \mathbb{F}_2$ everywhere as
            return cval[$y$] · table[$m, y$]
    compute evaluations[$y$], evaluations : $[2^n] \to \mathbb{F}_2$ everywhere as
        return evaluations[$y$] · (1 − cval[$y$])
// is there any row of evaluations containing only True (1)?
result ← evaluations
for $k = 0$ to $n$ do
    compute result[$y$], result : $[2^n] \to \mathbb{F}_2$ everywhere as
        return 1 − (1 − result[$y$]) · (1 − result[$y + 2^k$])
        // $= 1 \Leftrightarrow$ result[$y$] $= 1 \vee$ result[$y + 2^k$] $= 1$
return result[0];
```

again and printed on transparencies, the evaluation of the entire formula can be done again by aligning these transparencies. If a row remains transparent (this corresponds to each clause having the Boolean value True), this row in the truth table satisfies the formula.

All the used operations can be modeled via DPM operations over the domain $\mathbb{R}^2$. Transparency (False) can be encoded as 1 and absorbing material (True) as 0. This unusual assignment carries the OR operation by overlaying the transparent plastic sheets over to multiplication on DPMs. Each produced transparency can be considered as a register on the DPM. The process of photocopying corresponds to the re-evaluation of a register. Offsetting a sheet on the copier corresponds to accessing a register at an offset coordinate. If several slides are aligned on top of each other, this can be modeled through accessing other registers at a shifted position and *multiplying* the results.

We give in Algorithm 13 a translation for a DPM of Head (2009)'s Algorithm designed for photocopiers. For simplicity, we do not use idealized sheets, which could be modeled as with $D = [0, w] \times [0, h]$, as a domain and boxes of a tiny precomputed size to encode the values of Boolean variables. Instead, we will use the domains that are finite subsets of $\mathbb{Z} \times \mathbb{Z}$ which makes it easier to access a box at a given coordinate. However, a more close translation of the Algorithm would be possible on a DPM. Furthermore, we will use pseudo-code for reading and interpreting the Boolean formula and avoid the technical process of encoding the Boolean formula (Nethertheless, we could read and store the formula, for instance, in a register having a subset of $\mathbb{Z} \times \mathbb{Z}$ as domain).

The DPM algorithm only used offsets (positioning the sheets), `if` (masking and copying more sheets next to each other), and the operations $\cdot$ (overlaying two transparent sheets) and negation (inversion, which is assumed to be supported by the copier). The asymptotic running time of the DPM is equivalent to the asymptotic number of operations that a user and a copying machine has to perform. A constant time for each copy operation is assumed.

4.3.2. Analog feedback loops

An impressive "optical computer" can be built by pointing a camera at a screen that displays the image that the camera records. After some time, this system will render a tunnel: An image is rendered that shows a screen that displays a screen that displays a screen, and so on. We will call this system a video feedback loop.

Let us assume for the time being that both the screen and the camera have infinite resolution and that we can further put some objects such as mirrors, lenses between the camera and the screen that precisely perform some mathematical deformations. Several DPM algorithms over a rectangular subdomain of $\mathbb{R}^2 \cong \mathbb{C}$ can be simulated analogously with such a feedback loop. More precisely, consider programs of the form of Algorithm 14:

Algorithm 14: A DPM scheme representing analog feedback loops for an $O(1)$-computable function f

```
Input: An initial image $\xi$
Output: A sequence of images generated by the feedback loop
1 while true do
2     compute $\xi[x], \xi : \mathbb{R}^2 \to [0,1]^3$ everywhere as
3         return $f(x, \xi)$
4     display $\xi$
```

If f can be realized optically through a physical setting, then a video feedback loop can compute the program. An example would be to construct an f that creates the overlay of three versions of ξ that each is scaled to half its size. For instance

$$f(x,\xi) = \xi[2x + e^{\frac{0}{3}2\pi i}] + \xi[2x + e^{\frac{1}{3}2\pi i}] + \xi[2x + e^{\frac{2}{3}2\pi i}], \qquad x \in \mathbb{C}.$$

In theory, such a system can be realized by semi-transparent mirrors or by replacing the screen with three idealized beamers that project the captured image three times partially overlayed and scaled to half its size. The described system models an iterated function system and obtain images of the Sierpinski triangle as limit. Note that the scheme of Algorithm 14 can be very powerful depending on the chosen f. For instance, the fast-iteration algorithm presented in Algorithm 11 or the simulation of a cellular automaton as in Algorithm 5 can be considered as instances of this scheme.

Realization of analog feedback loops

We have built up a similar system for a demonstration at the Open Doors Day at Technical University of Munich in 2016[3]. The primary purpose of this project was the communication of mathematics in a playful way.

In order to simulate the results of using physical obstacles such as systems of mirrors, lenses etc. between the camera and the screen, one could simply describe the induced optical effects through mathematical transformations. These transformations can be computed on the GPU using CindyGL before displaying the recorded image on the screen (for details on CindyGL see Chapter 7).

[3]The programs and a demonstration video in Applet ▷5. In order to run these examples, a webcam that points to the screen is required. This can be the combination of an external screen and an internal webcam, or a USB-connected camera or a smartphone as a webcam together with an app like DroidCam. Good results are obtained if all automatic camera adjustments are disabled. On Linux, this can be done with guvcview.

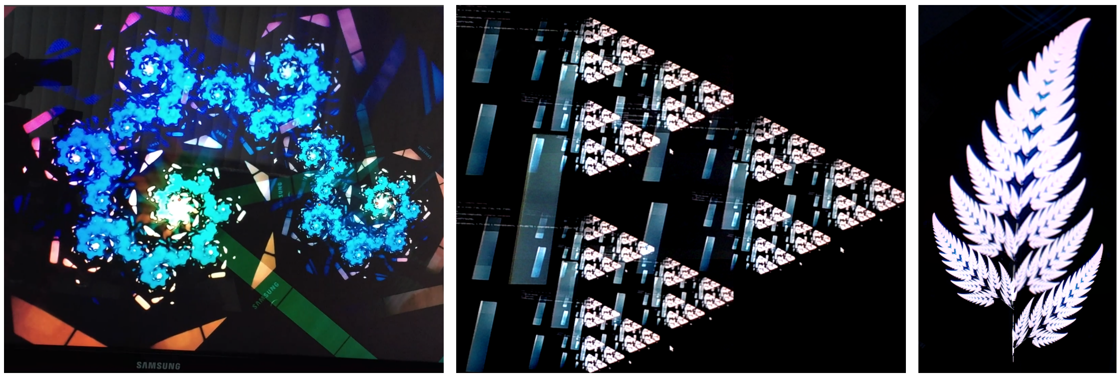


Figure 4.1.: IFS-Fractals rendered through a camera that points to a screen that multiple overlaid copies of the recorded image: (a) A two-generator IFS, (b) Sierpinski's triangle obtained by compositing three copies by itself, (c) Barnsleys farn generated through 4 deformed overlays recordings.

Recording a physical screen induces a perspective deformation. Instead of setting up a fine-adjusted physical system that would simulate various feedback loops, we initialized a non-calibrated system by showing four points on the screen and measured the recorded image of those four points. These four pairs of points define a unique projective transformation that has been used for a geometric calibration: Before displaying the recorded image on the screen, the inverse projection can be applied on the GPU.

This procedure resulted in the advantage that switching between several optical feedback loops could be done very fast and visitors could step between the camera and the screen and therefore interact with the entire system. Several IFS have been realized, such as spirals, the Sierpinski triangle and the Barnsley farn (see Figure 4.1).

Also, the Julia set could be realized by computing $f(z) = z^2$ (see Figure 4.2a).

An elegance of using a screen that displays at position z the color of the recorded position at z^2 ($z \in \mathbb{C}$. de Smit et al. (2012) coined the term the "quadratic camera") is that this system is relatively prone to errors and does not require many adjustments in the relative position of screen and camera. No "geometric calibration" as described about is required to obtain meaningful images in such a feedback loop.

Let us assume that the camera "sees" at position $z \in \mathbb{C}$ the pixel with coordinate

$$d(z) = az + b$$

of the screen, where $a, b \in \mathbb{C}, a \neq 0$. This corresponds to a similarity transformation. We set $c := a \cdot b$. The entire system can be described by a function that describes from

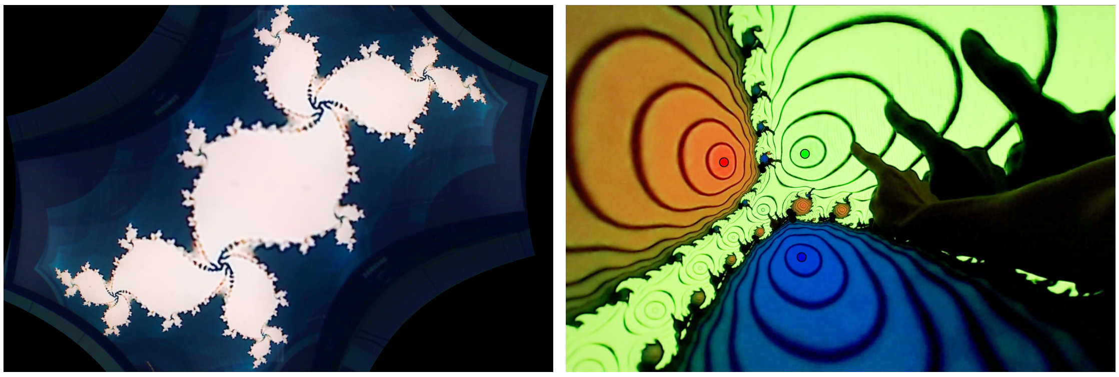


Figure 4.2.: Fractals rendered through a camera that points to a screen that shows a conformal deformation of the recorded image: (a) The Julia via a “quadratic camera”, (b) Newton fractals (with hand) via the deformation of a single Newton step on a polynomial with three roots. The roots have been highlighted by overlaying a red, green and blue circle at its position.

which position of the screen the color of a “pixel” with coordinate $z \in \mathbb{C}$ is obtained. In this case the function can be described as follows:

$$\begin{aligned} d \circ (z \mapsto z^2) &= (z \mapsto az^2 + b) \\ &= (z \mapsto \frac{z}{a}) \circ (z \mapsto z^2 + ab) \circ (z \mapsto az) = \phi^{-1} \circ (z \mapsto z^2 + c) \circ \phi. \end{aligned}$$

The feedback loop therefore is conjugated to the quadratic Julia map $z \mapsto z^2 + c$ via the homeomorphism $\phi(z) = az$.

As an example of an outstanding application of this kind, the system can also realize Newton’s iteration. For this context the aforementioned “projective calibration” is essential. Let $p : \mathbb{C} \to \mathbb{C}$ be a polynomial with derivative p'. If we set $f(z, \xi) = \xi[z - \frac{p(z)}{p'(z)}]$ in Algorithm 14, then a step of iteration of the analog feedback loop corresponds a parallel (backward) Newton iteration on the entire screen. After some time, all points in the basin of attraction of the same root of the polynomial will be displayed with the same color. If the roots are colored artificially (for instance by putting physically colored objects in front of them at the corresponding position at the screen) then the entire basin will get this color after some time. The Newton fractal can be rendered through an analog feedback loop.

Also, the cellular automaton of the Game of Life has been modeled through an analog feedback loop.

After the Open Doors Day, the concept has been demonstrated to several school classes. It turned out to be a good start to point the camera to an image that is already familiar to the audience, for instance, the audience itself. The live deformation that is applied to the recorded image then immediately becomes clear. The moment the camera is pointed at the screen and the feedback loop is enabled, the audience is cognitively challenged. This might be a good time to explain the underlying mathematical structures.

Chapter 5.

First steps in computability and complexity theory of DPMs

In the first part of this Chapter, we will show what can be principally computed on a DPM. We will compare DPMs with other computational models in terms of the *set of their computable functions*. It will turn out, that the set of computable functions of DPMs *equals* to the set of computable functions of uniform BSS machines (Blum et al., 2012), provided that representatives of one model can understand the input and output of the other model. However, the running times might be enhanced with DPMs compared to BSS machines. The limits of DPMs thus are to be found in their run-time-complexity. In the second part of this Chapter, we take the first steps in investigating how the available domains of a DPM affect the running time.

5.1. Relations of DPMs to other computational models

Since the computational models have their own, often very elaborate, definition that tends to appear differently in different sources, we will avoid reformulating one particular definition and omit endeavors to formulate lengthy technical proofs reflecting one exact, but arbitrarily chosen, definition. Specific models of computations have certain computational capabilities that are independent of their precise definition and specify algorithms in clearly readable *pseudo-code*.

We believe that this approach provides a distraction-free understanding of the statements and will not cause any confusions. We assume that the reader of this Section is familiar with basic concepts of Turing machines (Hopcroft, 2008) and uniform BSS machines (Blum et al., 2012). The computational ability of the latter can be thought of as a conventional computer performing computations over arbitrary rings.

5.1.1. Simulation and acceleration of BSS machines and Turing machines through DPMs

In this section, we present a DPM that can simulate every *uniform* BSS machine.

We will give an informal description of uniform BSS machines. For an exact (but much more extensive) definition, we refer to (Blum et al., 2012). For a given ring R, a *uniform* BSS machine can store elements of R on a bi-infinite tape. During the "execution" of the BSS machine, this tape will store only finite (but unbounded) sequences of R. At all the other positions, the tape will store $0 \in R$. BSS machines can be considered as a generalization of Turing machines. If R is chosen as $\mathbb{Z}_2$, then the definition of a uniform BSS machine is equivalent to the definition of a Turing Machine (Blum et al., 2012). Thus, if we show that any BSS machine can be simulated with a DPM, we have also shown that we can simulate Turing Machines with DPMs.

Similar to a DPM, the program flow of a uniform BSS machine is described as a finite connected directed graph consisting of *input*, *branch*, *computation*, *output* and *shift* nodes. All nodes η, except the output and branch nodes, have a unique next node β_η. The branch nodes have two next nodes β^+_η and β^-_η, out of which during the computation one is chosen as next node based on an associated R-entry on the tape.

An inclusion map from the space of all finite R sequences to the tape is associated with the input node. After initialization of the machine, the tape is filled with the inclusion map applied to the input. The output nodes have maps associated that take finite subsequences of the tape.

The computation nodes alter one R-entry on the tape with a fixed position through a polynomial function that has other R-entries with fixed position as input. If R is a field, this function can also be rational. The shift nodes either shift the entire tape to the left or to the right. They change the positions of all entries simultaneously.

The computation of a BSS machine can be simulated though a DPM over the domains $\mathcal{D} = \{*, \mathbb{Z}\}$ as in Algorithm 15; The DPMs simulates "tape" as a function $\tau : \mathbb{Z} \to R$. For simplicity, the simulating DPM reads a function $x_0 : \mathbb{Z} \to R$ representing the tape after processing input and outputs a function $\tau : \mathbb{Z} \to R$ describing the tape upon the termination of the BSS machine.

5.1.2. Acceleration of finite-dimensional BSS machines through DPMs

We want to show that every *finite-dimensional* BSS machine over R can be accelerated using a DPM with a sufficiently large parallel domain over the same ring R.

A *finite-dimensional* BSS machine can be considered as a uniform BSS machine without shift nodes. Instead of a bi-infinite tape, it will have only a finite-dimensional state

Algorithm 15: A DPM with $\mathcal{D} = \{*, \mathbb{Z}\}$ over the ring $\mathbb{Z} \times R$ that simulates every BSS machine over the ring R

Input: A specification of a BSS machine M over the ring R according to (Blum et al., 2012), the tape-state $x_0 : \mathbb{Z} \to R$ of the BSS machine after reading finite-length input

Output: The tape state at termination of M with the given.

```
τ ← x_0 // Copy the input to the tape τ : ℤ → R
η ← β_{η_0} // The unique node after the input node
while η is not an output node do
    if η is a branch node based on τ[i_η] then
        if τ[i_η] = (⩾)0 then // if R is ordered, then ⩾ is used.
            η ← β_η^+
        else
            η ← β_η^-
    else
        if η computes τ[a] ← g(τ[x_1], ..., τ[x_n]) with g : R^n → R polynomial (or
          rational if R is a field) then
            compute τ[x], τ : ℤ → R everywhere as
                if x = a then  return g(τ[x_1], ..., τ[x_n])
                else  return τ[x]
        else if η is a shift node then
            compute τ[x], τ : ℤ → R everywhere as
                if η shifts τ left then  return τ[x + 1]
                else if η shifts τ right then  return τ[x − 1]
        η ← β_η
return τ
```

space R^k. We call such a machine a k-dimensional BSS machine. According to Blum et al. (2012), for every uniform *uniform* BSS machine with fixed size of input and bounded running time, there exists a *finite-dimensional* BSS machine computing the same function with the same time effort as the uniform BSS machine. Alternatively, finite-dimensional BSS machines can be regarded as DPMs with access only to the trivial domain $*$, as we have already proven in Remark 5.

The following Lemma shows a straightforward method for exponentially accelerating these machines by utilizing the fast-iteration approach of Algorithm 11.

Lemma 29. *If M is a k-dimensional BSS machine over R with m nodes, $|R| \geqslant m$, state space R^k and running time $\Theta(t(n))$ for a finite-dimensional input containing a number $n \in \mathbb{N}$, then there is a R-DPM with $\mathcal{D} = \{*, [m] \times R^k\}$ that can simulate M in $O(\log t(n))$.*

Proof. Without loss of generality, let R be a ring that contains $[m] = \{1, \dots, m\}$ as subset. If R does not contain $[m]$ as subset, then because of $|R| \geqslant m$ an inclusion $[m] \hookrightarrow R$ can be considered to identify elements of R with the corresponding numbers.

We modify the k-dimensional BSS machine in such a way that any node is indexed through an integer of $[m]$ and 1 is the starting node.

The modified BSS machine has an associated *computing endomorphism*

$$H : [m] \times R^k \to [m] \times R^k,$$

similar to the one we defined in Definition 7.

$$H(\eta, x) = (\beta_\eta(x), g_\eta(x))$$

means that $\beta_\eta(x) \in [m]$ is the next node and $g_\eta(x) \in R^k$ the register value after one computational step of a running BSS in the state $\eta \in [n]$ and the value $x \in R^k$ stored in its registers. For every termination node $\tau \in \mathcal{T} \subset [m]$ and $x \in R^k$ we set $H(\tau, x) = (\tau, x)$ as a fixed point.

The function H is computable on a DPM in parallel. We define M as the DPM that implements fast-iteration approach of Algorithm 11 for the map $H : [m] \times R^k \to [m] \times R^k$.

Let $x_0 \in R^k$ be the state of the registers after reading the input. Using Algorithm 11, we can compute in $l \leqslant \lceil \log_2 t(n) \rceil$ computational steps of M the iterates H^1, H^2, H^4,... until $H^{(2^l)}(\eta_0, x_0) = (\tau, x_{end})$ for some termination node $\tau \in \mathcal{T}$. We set M to output the value based on $x_{end} \in R^k$. Then M simulates the BSS machine in $O(\log t(n))$ and produces the same output. □

5.1.3. Simulation of DPMs through uniform BSS machines

The reverse statement of the previous Section is valid as well, and we will outline a proof here. This implies the equality of the computational strengths of DPMs and BSS.

We will also show that *computational strength does not increase* with the use of DPMs compared to *uniform BSS machines*.

For a comparison of the computational strength, it is crucial that a machine of one model can understand the input and output of the machine to be simulated of the other model. A DPM can read entire functions as input, whereas an universal BSS machine can take only a vector of ring elements *of finite length* as input. Moreover, the output of a DPM can be an entire function, whereas a BSS machine can produce only a finite-length vector of ring elements as output.

This restriction to finite input and output limits BSS machines compared to DPMs. However, providing the BSS-machine with an *oracle* that allows for reading the input of the DPM at an arbitrary coordinate avoids this technical hurdle. There is a BSS machine that can simulate any DPM over the same ring, provided the BSS-machine has access to such an oracle. The BSS machine does not produce the same output as the DPM machine, which might be an entire function. Instead, the BSS machine computes the output of the DPM point-wise. i.e., the BSS machine takes a register and a coordinate as input and will calculate the value of the register of the DPM at the given coordinate after the DPM has terminated with the given input. With this, a BSS can compute essentially the same as a DPM. If the DPMs are restricted to vectors of finite length as inputs and outputs, then a BSS avoiding the oracle simulates the same behavior as the DPM by successively calculating the entire output and reading the input directly.

In Algorithm 16, a simulating BSS machine is presented in *pseudo-code*, because a technical formalization close to the exact definition would go beyond the scope of this thesis[1]. We silently assumed that R is ordered. If R is not ordered, then all the branching conditions $\leqslant$ and $>$ have to be replaced by $=$ and $\neq$ respectively.

Algorithm 16 simulates the program flow of the DPM M. All potentially occurring types of nodes (*input*, *output*, *branch* and *computation*) are taken into consideration. The algorithm terminates if and only if M terminates.

The stack data structure of cstack and its operations are to be understood as in Cormen et al. (2009). It contains a history of evaluated commands. These commands are the representations of the operations of Definition 4 through a finite number of ring elements.

The function ψ is crucial for this algorithm. Given the current history of performed computations, the register id, and coordinate, it computes the value of the register at the specified coordinate. The recursive function ψ terminates because the finite size

[1] In principle, it should be possible to transform the pseudo code into the exact specification of a BSS machine, just as it is also possible to specify Turing machines with high-level pseudo-code (Hopcroft, 2008).

Algorithm 16: A BSS machine that simulates every DPM

```
   Input: A specification of a DPM $M$ according to Definitions 2 and 4, an oracle
          $I(x_0) : \bigcup_{k=1}^{h}(k, D_k) \to R$, which indicates the states of the DPM after the
          input $x_0 \in I_M$ has been given and the coordinate $(k, c)$, where $k \in \{1, \dots, n\}$
          and $c \in D_k$, specifying the output.
   Output: The value $h_k(c) = \phi_M(x_0)(k, c)$, the output of $M$ at position $(k, c)$ given the
           input $x_0$.
 1 $\eta \leftarrow \beta_{\eta_0}$ // The unique node after the input node
 2 initialize cstack as an empty stack
 3 while $\eta$ is not an output node do
 4     if $\eta$ is a branch node then
 5         $v \leftarrow \psi(\text{cstack}, i_\eta, *)$ // $\psi$ returns the value $h_{i_\eta} \in R$ (Line 11)
 6         if $v \geqslant 0$ then $\eta \leftarrow \beta_\eta^+$ else $\eta \leftarrow \beta_\eta^-$
 7     else if $\eta$ is a computation node then
 8         cstack.push(finite representation of $g_\eta$) // as in Definition 4
 9         $\eta \leftarrow \beta_\eta$ // the unique next node of $\eta$
10 return $\psi(\text{cstack}, k, c)$ // the value $h_k(c)$
11 Function $\psi(\text{cstack}, k, x)$
       Input: The history cstack of previously performed computations, the index $k$
              of a register, the coordinate $x$ within the given register.
       Output: The value $h_k(x)$ after the computations specified in cstack
12     if cstack.empty() then // No commands have been evaluated
13         return $I(x_0)(k, x)$ // read from the input oracle
14     else
15         lastop $\leftarrow$ cstack.top()
16         cstack.pop() // remove last operation from cstack
17         if lastop is of kind $h_k \leftarrow \boldsymbol{\alpha}$ with $\boldsymbol{\alpha} \in R$ then
18             return $\boldsymbol{\alpha}$
19         else if lastop is of kind $h_k \leftarrow \pi_m$ then
20             compute $\pi_m(x)$ directly and return result
21         else if lastop is of kind $h_k \leftarrow h_i \odot h_j$ with $\odot \in \{+, -, \cdot, \div\}$ then
22             return $\psi(\text{cstack}, i, x) \odot \psi(\text{cstack}, j, x)$
23         else if lastop is of kind $h_k \leftarrow \texttt{if}(h_l \geqslant 0, h_i, h_j)$ then
24             if $\psi(\text{cstack}, l, x) \geqslant 0$ then return $\psi(\text{cstack}, i, x)$ else return
                $\psi(\text{cstack}, j, x)$
25         else if lastop is of kind $h_k \leftarrow h_j \circ (h_{i_1}, \dots, h_{i_l})$ then
26             return $\psi(\text{cstack}, j, (\psi(\text{cstack}, i_1, x), \dots, \psi(\text{cstack}, i_l, x)))$;
27         else // $h_k$ has not been modified by lastop
28             return $\psi(\text{cstack}, k, x)$
```

of cstack decreases in Line 16 before every potential recursive call. Induction on the size of cstack yields that $\psi(\text{cstack}, h, k)$ indeed returns the value $h_k(x)$, where h_k is the k-th register of the DPM that has previously executed all the commands in cstack.

Let $T(k)$ be the maximum number of operations required by the BSS machine implementing Algorithm 16 to simulate k steps of any DPM accessing domains of dimension $\leqslant d$. Simulating k steps could be considered as simulating k iterations of the loop in Line 3. All operations aside potential recursive calls of ψ within this loop require a constant number of operations. The major contributions to $T(k)$ origin from calling ψ. The number of recursive self-calls within ψ is bounded and the running time of calling ψ with cstack of length k is less than $T(k)$. In every case the number of self-calls is a constant number, only in Line 26 it is bounded by the maximal occurring dimension +1. Thus, up to some constant terms $\delta > 0$ and $C_d > 0$, we yield

$$T(k+1) \leqslant C_d T(k) + \delta,$$

which results in $T(k) = O(C_d^k)$. In other words, the running time increases in worst case “only” exponentially when simulating a DPM through a BSS machine. This gives rise to the following *lower bound* on running times on DPMs:

Lemma 30. *For any $d \in \mathbb{N}$ it exists a $C_d \in \mathbb{R}_{>0}$ such that the running time of any problem requiring at least $\Omega\left(t(n)\right)$ operations on a BSS machine requires at least $\Omega\left(\log_{C_d} t(n)\right)$ operations on a DPM that has access to domains of dimension $\leqslant d$.*

Proof. We choose C_d according to our previous observation. Suppose there was a DPM with domains with dimensions $\leqslant d$ that solves the given problem in $o\left(\log_{C_d} t(n)\right)$. Then Algorithm 16 could be used to solve the problem using

$$o\left(C_d^{\log_{C_d} t(n)}\right) = o\left(t(n)^{\log_{C_d} C_d}\right) = o\left(t(n)\right)$$

operations on a BSS machine by simulating the DPM. This contradicts the lower bound of the running time of the given problem on a BSS machine. □

Corollary 31. *There are decidable problems that are not solvable in polynomial time on a DPM.*

Proof. Take a decidable problem that requires at least doubly exponential time. For instance, any decision procedure for Presburger arithmetic requires at least doubly exponential time (Fischer et al., 1974) in the length of its input. According to Lemma 30, deciding this problem on a DPM requires at least exponential running time. □

5.2. Domains and complexity

We have seen that any DPM over the ring R with an arbitrary domain can be simulated on a BSS machine over the ring R, which then can be simulated on a DPM with access to a domain containing $\mathbb{Z}$. Thus, the set of computable functions is independent of the allowed set of domains provided that $\mathbb{Z} \hookrightarrow D \in \mathcal{D}$ of the DPMs. However, the complexity of problems, in terms of running time, might change if the allowed domains change.

Here we take first steps in investigating how the available domains of a DPM affect the running time. In the definition of the DPM, we did not fix the ring R and the set of the allowed domains. In this part, we assume that $R = \mathbb{R}$. First we specify, the complexity classes of functions that can be computed on a DPM over $\mathbb{R}$ with a given domain. Note that it is sufficient to specify only the "largest" domain in $\mathcal{D}$, because this larger domain can be used to perform computations on smaller domains as well.

Definition 32. Let $D \subset \mathbb{R}^n$ for some $n \in \mathbb{N}$. We specify

$$\begin{aligned} \mathrm{P}_{\mathrm{DPM}(D)} = \{f : \mathbb{N} \times A \to B \mid & f \text{ is a function with } A \subset \mathbb{R}^k, B \subset \mathbb{R}^l \text{ for some } k, l \in \mathbb{N} \\ & \text{that can be computed in } \textit{polynomial time} \text{ in one} \\ & \text{input parameter } n \text{ by a } \mathbb{R}\text{-DPM with the} \\ & \text{domains } \mathcal{D} = \{*, D\}\}. \end{aligned}$$

Oberservation 33. Increasing the size of the available domains does not increase the asymptotic running time of a problem, because computations on a domain can be always simulated by performing these operations on a bigger domain. Therefore we yield:

$$\mathrm{P}_{\mathrm{DPM}(*)} \subseteq \mathrm{P}_{\mathrm{DPM}(\mathbb{R})} \subseteq \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)} \subseteq \dots$$

An interesting question is whether these subsets are proper.

A function $f : \mathbb{N} \times \mathbb{R} \to \mathbb{R}$, that we have already studied in terms of runtime complexity, is the map $p_n(x) = x^{2^n}$. Using the degree, we could argue in Lemma 26 that any finite-dimensional BSS machine over $\mathbb{R}$ needs at least linear time to compute this map, while a DPM machine implementing Algorithm 12 iterating the map $f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$, requires only $O(\log n)$ time. As we have argued in Remark 5, finite-dimensional BSS coincide with DPMs with $\mathcal{D} = \{*\}$. If we apply these results to the map $\textsc{RealItSquare} : \mathbb{N} \times \mathbb{R} \to \mathbb{R}$ defined as

$$\textsc{RealItSquare}_n(x) := p_{2^n}(x) = x^{2^{2^n}}$$

we obtain the following corrollary, that states that the first inclusion in Oberservation 33 is proper:

Lemma 34.

$$\mathrm{P}_{\mathrm{DPM}(*)} \subsetneq \mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$$

Proof. We have $\textsc{RealItSquare} \in \mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$, because $\textsc{RealItSquare}_n(x) = p_{2^n}(x)$ can be computed in $O(\log 2^n) = O(n)$ on a DPM over the domain $\mathbb{R}$ using Algorithm 11 for the map $f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$. However, computing $\textsc{RealItSquare}_n = p_{2^n}$ on a DPM over the domains $\mathcal{D} = \{*\}$ or a finite-dimensional BSS machine requires at least exponential time in n according to Lemma 26, hence $\textsc{RealItSquare} \notin \mathrm{P}_{\mathrm{DPM}(*)}$. □

What about the other inclusions in the chain of Oberservation 33?

Conjecture 35. For every $n \in \mathbb{N}$, the inclusion

$$\mathrm{P}_{\mathrm{DPM}(\mathbb{R}^n)} \subsetneq \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^{n+1})},$$

is proper, i.e, there is a problem that essentially requires $n + 1$ instead of n dimensions of the parallel domain on a DPM to be “solved” efficiently.

Unfortunately we can not solve this conjecture, however let us do some considerations.

Let us begin to study the problem for $n = 1$, i.e. which of $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} \subsetneq \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ or $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} = \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ holds? A strategy to prove $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} \subsetneq \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ is to find a problem that can be solved in polynomial time on a DPM with $\mathbb{R}^2 \in \mathcal{D}$, but cannot be solved in polynomial time on a DPM with access to the domains $\mathcal{D} = \{*, \mathbb{R}\}$. Candidates are iterations that essentially require a two-dimensional domain to be carried out efficiently.

As a candidate, let us consider the problem of computing iterates of the first non-trivial or linear map over $\mathbb{C} \simeq \mathbb{R}^2$, the complex squaring function $z \mapsto z^2$. The function decoding its 2^n-th iterate is the following.

$$\begin{aligned} \textsc{ComplexItSquare} : \mathbb{N} \times \mathbb{C} &\to \mathbb{C} \\ (n, z) &\mapsto z^{2^{2^n}} \end{aligned}$$

Via the identification $\mathbb{C} \simeq \mathbb{R}^2$, $(x, y) \mapsto x + i \cdot y$, the function $\textsc{ComplexItSquare}_n$ can be computed on a DPM over the domains $\mathcal{D} = \{*, \mathbb{R}^2\}$ by iterating the complex squaring function $\mathbb{C} \to \mathbb{C}, z \mapsto z^2$ in $O(n)$ using Algorithm 11. Thus,

$$\textsc{ComplexItSquare} \in \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}.$$

Is $\textsc{ComplexItSquare}$ a candidate to seperate $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$ from $\mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$?

Question 36. Does

$$\textsc{ComplexItSquare} \in \mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$$

hold? I.e., is there an algorithm for DPMs that uses the domains $\mathcal{D} = \{*, \mathbb{R}\}$ that takes $n \in \mathbb{N}$, $x, y \in \mathbb{R}$ as input and computes the values $\mathrm{Re}((x+iy)^{2^{2^n}})$ and $\mathrm{Im}((x+iy)^{2^{2^n}})$ in polynomial time in n?

If the answer to Question 36 is **No**, then $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} \subsetneq \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ is proper and Conjecture 35 holds for $n = 1$. If the answer is **Yes**, there is an interesting algorithm that solves the given problem fast, but still does not answer Conjecture 35.

Unfortunately, it will turn out that the answer is **Yes** (see the following section). Consequently, an algorithm for computing $\textsc{ComplexItSquare}$ on a DPM over the domains $\mathcal{D} = \{*, \mathbb{R}\}$ is constructed.

5.2.1. Iterating complex squaring with a DPM with real domain

The aim is to iterate the complex squaring function $\mathbb{C} \to \mathbb{C}$, $z \mapsto z^2$ on a DPM with the domains $\mathcal{D} = \{*, \mathbb{R}\}$ efficiently. More specifically, from the real and imaginary part of z, the real and imaginary part of $z^{2^{2^n}}$ is to be computed in polynomial time with a DPM that has only access to the domains $\mathcal{D} = \{*, \mathbb{R}\}$.

The core idea of the presented algorithm is that complex multiplication can be carried over to essential one-dimensional problems that are independent from each other: Let $z_1, z_2 \in \mathbb{C} \setminus \{0\}$, then

$$z_1 \cdot z_2 = \underbrace{\left(\frac{z_1}{|z_1|} \cdot \frac{z_1}{|z_1|}\right)}_{\text{a computation on } S^1 \subset \mathbb{C}} \cdot \underbrace{(|z_1| \cdot |z_2|)}_{\text{a computation on } \mathbb{R}_{\geqslant 0} \subset \mathbb{C}} .$$

Both multiplications take place on an one-dimensional $\mathbb{R}$-manifold. With $S^1 \subset \mathbb{C}$ we denote all complex numbers with the norm 1. Using the commutivity of multiplication, this idea can be adopted to the following approach for $z \in \mathbb{C} \setminus \{0\}$:

$$\textsc{ComplexItSquare}_n(z) = z^{2^{2^n}} = \underbrace{\left(\frac{z}{|z|}\right)^{2^{2^n}}}_{\text{a computation on } S^1 \subset \mathbb{C}} \cdot \underbrace{|z|^{2^{2^n}}}_{\text{a computation on } \mathbb{R}_{\geqslant 0} \subset \mathbb{C}}$$

One problem is that the value $|z| = |x+iy| = \sqrt{x^2+y^2}$ cannot be computed on a DPM since we did not include $\sqrt{\cdot} : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ in the set of the allowed functions. However, for this special problem, we can use a trick to avoid this computation: Instead of squaring z iteratively (2^n times), we will square z^2 iteratively ($2^n - 1$ times) and use

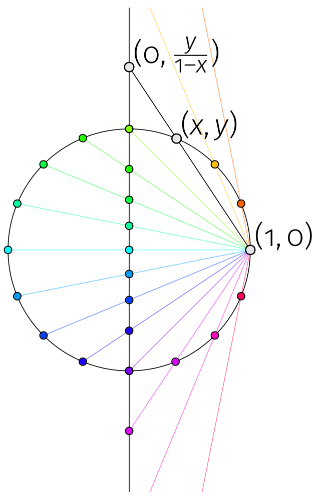


Figure 5.1.: Image of the chosen stereographic projection in $\mathbb{R}^2 \simeq \mathbb{C}$.

the fact that the norm $|z^2| = |z|^2 = |(x + i \cdot y)|^2 = x^2 + y^2$ is computable. In particular, we compute each of the terms of

$$\textsc{ComplexItSquare}_n(z) = z^{2^{2^n}} = (z^2)^{2^{2^n-1}}$$
$$= \underbrace{\left(\frac{z^2}{|z^2|}\right)^{2^{2^n-1}}}_{\text{a computation on } S^1 \subset \mathbb{C}} \cdot \underbrace{|z^2|^{2^{2^n-1}}}_{\text{a computation on } \mathbb{R}_{\geqslant 0} \subset \mathbb{C}}.$$

The real scalar $|z^2|^{2^{2^n-1}} = (x^2 + y^2)^{2^{2^n-1}}$ can be computed by iterating the map $f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$. A DPM over the domains $\mathcal{D} = \{*, \mathbb{R}\}$ using Algorithm 12 can yield the desired value $f^{2^n-1}(|z^2|)$ in $O(\log 2^n - 1) = O(n)$ steps.

Let us tackle the problem to compute iterated squares restricted to S^1 efficiently on a DPM over the domains $\mathcal{D} = \{*, \mathbb{R}\}$. We will map S^1 to the real line via a "suitable-choosen" stereographic projection. The complex number $1 \in \mathbb{C}^2 \cap S^1$ is a fixed point of the squaring operation. This fixed point is a suitable center for the stereographic projection onto $i\mathbb{R} \simeq \mathbb{R}$.

Definition 37. Let $\hat{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. We define the stereographic projection with center 1 as follows (compare Figure 5.1):

$$\sigma : S^1 \to \hat{\mathbb{R}}$$
$$x + iy \mapsto \begin{cases} \infty & \text{if } x + iy = 1 \\ \frac{y}{1-x} & \text{otherwise} \end{cases}$$

$\sigma\colon S^1 \to \hat{\mathbb{R}}$ is bijective and has the following inverse:

$$\begin{aligned}\sigma^{-1}\colon \hat{\mathbb{R}} &\to S^1\\ r &\mapsto \begin{cases} 1 & \text{if } r = \infty \\ \frac{(r^2-1)+i(2r)}{1+r^2} & \text{if } r \in \mathbb{R}\end{cases}\end{aligned}$$

Furthermore, let $N = 2^{2^n-1}$ and $\mathrm{sq}_{S^1}\colon S^1 \to S^1$, $x + i\,y \mapsto (x^2 - y^2) + i\,(2xy)$ describing the complex square function $z \mapsto z^2$ restricted to S^1.

We want to compute $\mathrm{sq}_{S^1}^N$ efficiently on a $\mathbb{R}$-DPM and want to reduce the number of used real parameters. Let us define $\mathrm{sq}_{\hat{\mathbb{R}}} = \sigma \circ \mathrm{sq}_{S^1} \circ \sigma^{-1}$ as the complex square function pulled back through the stereographic projection to $\mathbb{R}$ and one expectional point. We can use this conjugated function to compute $\mathrm{sq}_{S^1}^N$ as follows:

$$\mathrm{sq}_{S^1}^N = (\sigma^{-1} \circ \mathrm{sq}_{\hat{\mathbb{R}}} \circ \sigma)^N = \sigma^{-1} \circ \mathrm{sq}_{\hat{\mathbb{R}}}^N \circ \sigma$$

How can we compute $\mathrm{sq}_{\hat{\mathbb{R}}}^N : \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ efficiently?

Evaluation of $\mathrm{sq}_{\hat{\mathbb{R}}} = \sigma \circ \mathrm{sq}_{S^1} \circ \sigma^{-1}$ gives the following scheme for computation: If $r \in \{\infty, 0\}$, then $\mathrm{sq}_{\hat{\mathbb{R}}}(r) = \infty$, which corresponds to the fact that $1^2 = (-1)^2 = 1$. Otherwise, if $r \in \mathbb{R} \setminus \{0\}$, then

$$\mathrm{sq}_{\hat{\mathbb{R}}}(r) = \sigma(\mathrm{sq}_{S^1}(\sigma^{-1}(r))) = \sigma\left(\frac{(r^2-1)^2 - 4r^2}{(r^2+1)^2} + i\frac{4r\ (r^2-1)}{(r^2+1)^2}\right) = \frac{r^2-1}{2r}$$

We cannot include ∞ into the domain of a DPM. So we define

$$\begin{aligned}\mathrm{sq}_{\mathbb{R}} : \mathbb{R} &\to \mathbb{R}\\ r &\mapsto \begin{cases} \frac{r^2-1}{2r} & \text{if } r \neq 0 \\ 0 & \text{if } r = 0\end{cases}\end{aligned}$$

as a real-part-version of $\mathrm{sq}_{\hat{\mathbb{R}}} : \hat{\mathbb{R}} \to \hat{\mathbb{R}}$. Its iterations are efficiently computable in parallel on a DPM with domain $\mathbb{R}$. At the same time, we can keep track of the points that eventually became ∞ during the iteration of $\mathrm{sq}_{\hat{\mathbb{R}}}$. In Algorithm 17, those points $r \in \mathbb{R}$ that eventually attain the fixed-point ∞ of $\mathrm{sq}_{\hat{\mathbb{R}}}$ will be marked by $\textsc{isReal}[r] = 0$.

Now, we are ready to formulate the algorithm that computes $\textsc{ComplexItSquare}_n(z)$ on entire $\mathbb{R}^2$ on a DPM over the domains $\mathcal{D} = \{*, \mathbb{R}\}$ in polynomial time (see Algorithm 18). Therefore, $\textsc{ComplexItSquare} \in \mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$ and the answer to Question 36 is **Yes**. The iterated squaring-functions contains a structure that has been exploited.

Iterating a more complicated function on $\mathbb{C}$ such as $z \mapsto z^2 + c$ with $c \in \mathbb{C}^\times$ is perhaps a better candidate to separate $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$ from $\mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$? Let $\textsc{ComplexItJulia} : \mathbb{N} \times \mathbb{C} \times \mathbb{C} \to \mathbb{C}$

be the problem to compute the iterated map

$$\textsc{ComplexItJulia}_n(z, c) = \left(\xi \mapsto \xi^2 + c\right)^{2^n}(z)$$

Again, $\textsc{ComplexItJulia}_n(z, c)$ can be computed on a DPM with $\mathcal{D} = \{*, \mathbb{R}^2\}$ in $O(n)$ using Algorithm 11, thus $\textsc{ComplexItSquare} \in \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$.

Question 38. Does

$$\textsc{ComplexItJulia} \in \mathrm{P}_{\mathrm{DPM}(\mathbb{R})}$$

hold? If the answer is **No**, then $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} \subsetneq \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ is proper.

A vague intuition to trust in an inherent difficulty in iterating $z \mapsto z^2 + c$ comes from the fact that exist $c \in \mathbb{C}$ for which the limit-set has non-integral Hausdorff dimension. Having a non-integral Hausdorff dimension is the essential ingredient in the proof that the limit set for such $c \in \mathbb{C}$ cannot be decided by BSS machines Blum et al. (2012).

No proof of this property could be found. However, no evidence of $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} = \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ could be found. The question whether the inclusion $\mathrm{P}_{\mathrm{DPM}(\mathbb{R})} \subset \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^2)}$ is proper is still to be studied. Another interesting question could be whether

Question 39. Does $\mathrm{P}_{\mathrm{DPM}(\mathbb{R}^{n+1})} = \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^n)}$ imply that $\mathrm{P}_{\mathrm{DPM}(\mathbb{R}^{n+2})} = \mathrm{P}_{\mathrm{DPM}(\mathbb{R}^{n+1})}$?

If this implication would hold, then the entire tower of complexity classes would become stationary if two of classes turn out to be equal.

Algorithm 17: Fast iterative squaring of complex numbers on S^1 with a DPM over $\mathbb{R}$ and $\mathcal{D} = \{*, \mathbb{R}\}$

```
Input: $(x, y) \in S^1$ and $n \in \mathbb{N}$
Output: The real and imaginary part of $(x + iy)^{2^{2^n}-1}$ in $O(n)$
compute $\mathrm{sq}_{\mathbb{R}}[r], \mathrm{sq}_{\mathbb{R}} : \mathbb{R} \to \mathbb{R}$ everywhere as
    if $r = 0$ then return 0 else return $\frac{r^2-1}{2r}$
compute $\textit{isReal}[r], \textit{isReal} : \mathbb{R} \to \{0, 1\}$ everywhere as
    if $r = 0$ then return 0 else return 1
$g \leftarrow \mathrm{sq}_{\mathbb{R}}$
repeat $n$ times
    compute $\textit{isReal}[r], \textit{isReal} : \mathbb{R} \to \{0, 1\}$ everywhere as
        if $\textit{isReal}[r] \wedge \textit{isReal}[g[g[\mathrm{sq}_{\mathbb{R}}[r]]]]$ then return 1 else return 0
    $g \leftarrow g \circ g \circ \mathrm{sq}_{\mathbb{R}}$
    // After $k$ iterations, $g$ stores the function $\mathrm{sq}_{\mathbb{R}}^{2^{2^k}-1}$
// $g$ now stores the function $\mathrm{sq}_{\mathbb{R}}^{N}$
if $x = 1$ then
    return $(1, 0)$ // Then $y = 0$ and $(x + iy)^N = 1$
else
    $r \leftarrow \frac{y}{1-x}$ // $r = \sigma(x, y)$
    if $\textit{isReal}[r]$ then
        // return $\sigma^{-1}(\mathrm{sq}_{\mathbb{R}}^{N}(r))$
        $r \leftarrow g[r]$
        $x \leftarrow \frac{r^2-1}{1+r^2}$
        $y \leftarrow \frac{2r}{1+r^2}$
        return$(x, y)$
    else
        // The fixed-point $\infty$ occured while iterating $\mathrm{sq}_{\hat{\mathbb{R}}}$
        // We return $\sigma^{-1}(\infty) = 1$
        return $(1, 0)$
```

Algorithm 18: An algorithm computing ComplexItSquare in $O(n)$ on a DPM over the domains $\mathcal{D} = \{*, \mathbb{R}\}$

```
Input: $(x, y) \in \mathbb{R}^2$ that encode the complex number $z = x + iy$ and $n \in \mathbb{N}$
Output: The real and imaginary part of $z^{2^{2^n}}$
1 $\alpha \leftarrow x^2 + y^2$
2 if $\alpha = 0$ then
3     return $(0, 0)$ // $0^{2^{2^n}} = 0$
4 $\tilde{x} \leftarrow x^2 - y^2$
5 $\tilde{y} \leftarrow 2xy$
  // $\tilde{x} + i\tilde{y} = z^2$ and $|z^2| = \alpha$
6 phase $\leftarrow$ compute $(\frac{\tilde{x}}{\alpha} + i\frac{\tilde{y}}{\alpha})^{2^{2^n - 1}}$ according to Algorithm 17
7 norm $\leftarrow$ compute $\alpha^{2^{2^n - 1}}$ according to Algorithm 12
8 return $(\text{phase}_1 \cdot \text{norm}, \text{phase}_2 \cdot \text{norm})$
```

Part II.

Implementation: Pixel shaders as special DPMs

Chapter 6.

Approximating DPM algorithms on a GPU

As described in Section 4.2, a DPM with $\mathcal{D} = \{*, D\}$, and $D \subset \mathbb{R}^2$ bounded, can be considered as an idealized computer with the ability to run GPU shader programs. In fact, using only a bounded domain $D \subset \mathbb{R}^2$ poses no theoretical restriction compared to a DPM with $\mathcal{D} = \{*, \mathbb{R}^2\}$. Any DPM algorithm requiring $\mathcal{D} = \{*, \mathbb{R}^2\}$, i.e. utilizing the entire domain $\mathbb{R}^2$, can be reduced to an equivalent DPM with $\mathcal{D} = \{*, [-1,1]^2\}$, using only a domain with a bounded extent. Instead of storing a function $f : \mathbb{R}^2 \to \mathbb{R}$, two functions $g_1, g_2 : [-1,1]^2 \to \mathbb{R}$ could be stored, where $g_1 := f|_{[-1,1]^2}$ stores the local part of f and

$$g_2(x,y) := \begin{cases} f(\frac{x}{x^2+y^2}, \frac{y}{x^2+y^2}) & \text{if } x^2+y^2 \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

stores the remote part of f. The original $f : \mathbb{R}^2 \to \mathbb{R}$ can be reconstructed from $g_1, g_2 : [-1,1]^2 \to \mathbb{R}$ via rational functions and branching:

$$f(x,y) = \begin{cases} g_1(x,y) & \text{if } x^2+y^2 \leqslant 1 \\ g_2(\frac{x}{x^2+y^2}, \frac{y}{x^2+y^2}) & \text{otherwise} \end{cases}$$

These assignments can be computed in any involved $\langle$*parallel-statement*$\rangle$. So, without loss of generality, we can assume that $\mathcal{D} = \{*, [-1,1]^2\}$.

If $\mathcal{D} = \{*, D\}$ with bounded $D \subset \mathbb{R}^2$, registers on the DPM storing a function $h : D \to \mathbb{R}^k$ will correspond to textures stored on the GPU. Any **compute everywhere**-statement can be translated into a shader program. The simplicity of $\langle$*parallel-statement*$\rangle$ suits us at this point, as shader programs are generally not Turing complete. The running time of the execution of every $\langle$*parallel-statement*$\rangle$ does not depend on the value of the coordinates.

However, DPMs idealize computers equipped with GPUs in the following ways. GPUs have

Floating-point arithmetic. A computer and a GPU cannot represent arbitrary real numbers. However, floating-point arithmetic can be used to approximate the real arithmetic of a DPM.

Textures. Technically, a two-dimensional texture consists of a finite number of pixels that are aligned in a rectangular shape. Each pixel stores four values: red, green, blue, and alpha. It is most common to store each of these values as an 8-bit integer between 0 and 255. However, modern GPUs also support 32-bit floating-point values for each of the values, which is sufficient for most of our purposes. We want to encode functions $f : D \to \mathbb{R}$ with bounded $D \subset \mathbb{R}^2$ of our idealized model as floating-point textures. This is not possible without a loss of information. The process of function encoding has two steps:

Rasterization in the domain. We approximate a function $f : D \to \mathbb{R}$ by a finitely sampled function $\hat{f} : D^\varepsilon \to \mathbb{R}$ where $D^\varepsilon = \varepsilon\mathbb{Z}^2 \cap D$ and $\hat{f} = f|_{D^\varepsilon}$. Values $f(x)$ with $x \notin D^\varepsilon$ are estimated by resampling $\hat{f}$. Smaller values of $\varepsilon > 0$ correspond to higher resolutions. Not every DPM algorithm can be approximated with textures of a sufficiently high resolution, but several algorithms can. We will discuss this issue in the following section Section 6.2.

Use floating-point values in the co-domain. Instead of storing real numbers for each pixel, floating-point values that approximate the real numbers are used.

Only finite parallelism A GPU can run only a finite number of computations concurrently. In the DPM model, we assumed the possibility of infinite parallelism. After rasterization, parallelism is only required for finite domains D^ε; It suffices to run one thread for each data point of the textures. Therefore only a finite number of concurrent computations is required. The time needed by the computer with the GPU depends on the number of available parallel units. Unless the number of the execution units is not higher than the number of pixels of the used textures, the running time depends on the number of the execution units as well. We will assume that the ratio of the maximal number of pixels and the number of execution units is sufficiently small (both are constant numbers). Running a **compute everywhere** statement then can be considered as a constant time operation. If further time delays in memory access and communication between CPU and GPU are ignored, then the asymptotic running time of the DPM vaguely reflects the time of a real computer with a GPU that has a sufficient number of parallel units.

We will analyze the approximation through resampling/interpolation in the rest of this chapter. Then in Chapter 7, we will demonstrate a concept, how such programs utilizing the shader based technology of the GPU can be created via scripting. This concept makes it possible to realize a subset of DPM algorithms on the GPU very quickly. We have created a sample implementation *CindyGL*, which will also be demonstrated in Section 7.3. In the last part of the thesis (Part III), we will utilize *CindyGL* to obtain several applets that run on the GPU from various DPM algorithms.

6.1. Multi-linear interpolation

In this section, we will introduce *multilinear interpolation* of an arbitrary dimension from a functional point of view. In the two-dimensional case, the *bilinear interpolation* is built in within standard shader models for GPUs.

We will prove the property that Lipschitz continuity on the samples is carried over to the resampled function on the entire domain. These properties are needed in proofs of the following section. No other occurrences in the literature of a proof of the preservation of Lipschitz continuity through multilinear interpolation are known to the author, so we give an exact definition and proof here.

Definition 40. Let $\hat{f} : \mathbb{Z}^k \to \mathbb{R}$ be a discrete function. In the following, we define a *multilinear resampling operator* $\alpha_k : (\mathbb{Z}^k \to \mathbb{R}) \to (\mathbb{R}^k \to \mathbb{R})$ recursively.

For $k = 1$, we define $\alpha_1 : (\mathbb{Z} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$ through its evaluation at $\hat{f} : \mathbb{Z} \to \mathbb{R}$ and $x \in \mathbb{R}$ as follows, where $x = \lfloor x \rfloor + \mathrm{fract}(x)$:

$$\alpha_1(\hat{f})(x) = \alpha_1(\hat{f})(\lfloor x \rfloor + \mathrm{fract}(x)) = (1 - \mathrm{fract}(x)) \cdot \hat{f}(\lfloor x \rfloor) + \mathrm{fract}(x) \cdot \hat{f}(\lfloor x \rfloor + 1)$$

For $k > 1$, we define $\alpha_k : (\mathbb{Z}^k \to \mathbb{R}) \to (\mathbb{R}^k \to \mathbb{R})$ recursively. Given $\hat{f} : \mathbb{Z}^k \to \mathbb{R}$, $\tilde{x} \in \mathbb{R}^{k-1}$ and $x \in \mathbb{R}$ we set:

$$\alpha_k(\hat{f})(\tilde{x}, x) = \alpha_1(\hat{f}_{\tilde{x}})(x)$$

where $\hat{f}_{\tilde{x}} : \mathbb{Z} \to \mathbb{R}$ is obtained by resampling each of the "slices" $\hat{f} \circ i_s : \mathbb{Z}^{k-1} \to \mathbb{R}$ with $i_s : \mathbb{Z}^{k-1} \to \mathbb{Z}^k$, $x \mapsto (x, s)$ recursively:

$$\hat{f}_{\tilde{x}}(s) = \alpha_{k-1}(\hat{f} \circ i_s)(\tilde{x}).$$

By induction on k, it can be directly seen that

- $\alpha_k(\hat{f})$ extends the domain of $\hat{f}$, i.e. $\alpha_k(\hat{f})|_{\mathbb{Z}^k} = \hat{f}$

- The values of $\alpha_k(\hat{f})$ remain in the same range as the values of $\hat{f}$. In particular, $\|\alpha_k(\hat{f})\|_\infty = \|\hat{f}\|_\infty$.
- The operator α_k is linear, i.e. $\alpha_k(\lambda\hat{f} + \mu\hat{g}) = \lambda\alpha_k(\hat{f}) + \mu\alpha_k(\hat{g})$.

We will now prove through evaluation that Lipschitz-continuity of $\hat{f} : \mathbb{Z}^k \to \mathbb{R}$ is carried over to $\alpha_k(\hat{f}) : \mathbb{R}^k \to \mathbb{R}$ and give a bound on the Lipschitz-constant: $\mathrm{Lip}(\alpha_k(\hat{f}))$ can be bounded by $\mathrm{Lip}(\hat{f})$ up to a factor depending on the dimension k.

Lemma 41. *Let $\hat{f} : \mathbb{Z}^k \to \mathbb{R}$ be Lipschitz. Then also $\alpha_k(\hat{f}) : \mathbb{R}^k \to \mathbb{R}$ is Lipschitz. Furthermore,*

$$\mathrm{Lip}_{\max}(\alpha_k(\hat{f})) = \mathrm{Lip}_{\max}(\hat{f}),$$

where $\mathrm{Lip}_{\max}(g)$ *denotes the corresponing Lipschitz-constants with respect to the* maximum-norm *of a function $g : D \to \mathbb{R}$, i.e. $|g(x) - g(y)| \leqslant \mathrm{Lip}_{\max}(g)\|x - y\|_{\max}$ for any $x, y \in D \subset \mathbb{R}^k$.*

Proof. Since $\alpha_k(\hat{f})|_{\mathbb{Z}^k} = \hat{f}$, clearly $\mathrm{Lip}_{\max}(\alpha_k(\hat{f})) \geqslant \mathrm{Lip}_{\max}(\hat{f})$. We will prove the property $\mathrm{Lip}_{\max}(\alpha_k(\hat{f})) \leqslant \mathrm{Lip}_{\max}(\hat{f})$ by induction over k.

For $k = 1$ and $\hat{f} : \mathbb{Z} \to \mathbb{R}$ and $x < y \leqslant \lfloor x \rfloor + 1$, we have

$$|\alpha_1(\hat{f})(x) - \alpha_1(\hat{f})(y)| = |(\mathrm{fract}(x) - \mathrm{fract}(y)) \cdot (\hat{f}(\lfloor x \rfloor + 1) - \hat{f}(\lfloor x \rfloor))| \leqslant |x - y| \cdot \mathrm{Lip}_{\max}(\hat{f})$$

For $x < \lfloor x \rfloor + 1 \leqslant \lfloor y \rfloor \leqslant y$, we have

$$\begin{aligned} &|\alpha_1(\hat{f})(x) - \alpha_1(\hat{f})(y)| \\ &\qquad \leqslant |\alpha_1(\hat{f})(x) - \hat{f}(\lfloor x \rfloor + 1)| + |\hat{f}(\lfloor x \rfloor + 1) - \hat{f}(\lfloor y \rfloor)| + |\alpha_1(\hat{f})(\lfloor y \rfloor) - \alpha_1(\hat{f})(y)| \\ &\qquad \leqslant \mathrm{Lip}_{\max}(\hat{f}) \cdot |x - (\lfloor x \rfloor + 1)| + \mathrm{Lip}_{\max}(\hat{f}) \cdot |\lfloor x \rfloor + 1 - \lfloor y \rfloor| + \mathrm{Lip}_{\max}(\hat{f}) \cdot |\lfloor y \rfloor - y| \\ &\qquad \leqslant \mathrm{Lip}_{\max}(\hat{f}) \cdot |x - y| \end{aligned}$$

For arbitrary $x, y \in \mathbb{R}$ either $x = y$ or by possible swap of x and y, the same relation can be reduced to the two cases above.

Now let $k > 1$, $\hat{f} : \mathbb{Z}^k \to \mathbb{R}$ and $\tilde{x}, \tilde{y} \in \mathbb{R}^{k-1}$ and $x, y \in \mathbb{R}$. Then

$$\begin{aligned} |\alpha_k(\hat{f})(\tilde{x}, x) - \alpha_k(\hat{f})(\tilde{y}, y)| &= |\alpha_1(\hat{f}_{\tilde{x}})(x) - \alpha_1(\hat{f}_{\tilde{y}})(y)| \\ &\leqslant |\alpha_1(\hat{f}_{\tilde{x}})(x) - \alpha_1(\hat{f}_{\tilde{y}})(x)| + |\alpha_1(\hat{f}_{\tilde{y}})(x) - \alpha_1(\hat{f}_{\tilde{y}})(y)| \\ &\leqslant |\alpha_1(\hat{f}_{\tilde{x}} - \hat{f}_{\tilde{y}})(x)| + |\alpha_1(\hat{f}_{\tilde{y}})(x) - \alpha_1(\hat{f}_{\tilde{y}})(y)| \end{aligned} \tag{6.1}$$

The first summand of Equation (6.1) can be bounded by using the induction hypothesis for $k-1$ and the fact that $\hat{f} \circ i_s$ is Lipschitz-continious: With this we can bound $\|\hat{f}_{\tilde{x}} - \hat{f}_{\tilde{y}}\|_\infty$:

$$\begin{aligned} |(\hat{f}_{\tilde{x}} - \hat{f}_{\tilde{y}})(s)| &= |\alpha_{k-1}(\hat{f} \circ i_s)(\tilde{x}) - \alpha_{k-1}(\hat{f} \circ i_s)(\tilde{y})| \\ &\leqslant \|\tilde{x} - \tilde{y}\|_{\max} \cdot \mathrm{Lip}_{\max}(\hat{f} \circ i_s) \leqslant \|\tilde{x} - \tilde{y}\|_{\max} \cdot \mathrm{Lip}_{\max}(\hat{f}) \end{aligned}$$

In order to bound the second summand of Equation (6.1), we prove for $\hat{f}_{\tilde{y}} : \mathbb{Z} \to \mathbb{R}$ that $\mathrm{Lip}_{\max}(\hat{f}_{\tilde{y}}) \leqslant \mathrm{Lip}_{\max}(\hat{f})$:

$$|\hat{f}_{\tilde{y}}(s_1) - \hat{f}_{\tilde{y}}(s_2)| = |\alpha_{k-1}(\hat{f} \circ i_{s_1})(\tilde{y}) - \alpha_{k-1}(\hat{f} \circ i_{s_2})(\tilde{y})| = |\alpha_{k-1}(\hat{f} \circ i_{s_1} - \hat{f} \circ i_{s_2})(\tilde{y})|$$
$$\leqslant \|\hat{f} \circ i_{s_1} - \hat{f} \circ i_{s_2}\|_\infty \leqslant \mathrm{Lip}_{\max}(\hat{f})|s_1 - s_2|.$$

With the base case $k = 1$, it follows that $|\alpha_1(\hat{f}_{\tilde{y}})(x) - \alpha_1(\hat{f}_{\tilde{y}})(y)| \leqslant \mathrm{Lip}_{\max}(\hat{f})|x - y|$.

Alltogether, Equation (6.1) can be simplified to

$$|\alpha_k(\hat{f})(\tilde{x}, x) - \alpha_k(\hat{f})(\tilde{y}, y)| \leqslant |\alpha_1(\hat{f}_{\tilde{x}} - \hat{f}_{\tilde{y}})(x)| + |\alpha_1(\hat{f}_{\tilde{y}})(x) - \alpha_1(\hat{f}_{\tilde{y}})(y)|$$
$$\leqslant \|\hat{f}_{\tilde{x}} - \hat{f}_{\tilde{y}}\|_\infty + \mathrm{Lip}_{\max}(\hat{f})|x - y| \leqslant \mathrm{Lip}_{\max}(\hat{f})\,(\|\tilde{x} - \tilde{y}\|_{\max} + |x - y|)$$
$$\leqslant \mathrm{Lip}_{\max}(\hat{f})\|(\tilde{x}, x) - (\tilde{y}, y)\|_{\max},$$

which finishes the proof of $\mathrm{Lip}_{\max}(\alpha_k(\hat{f})) = \mathrm{Lip}_{\max}(\hat{f})$. Due the equivalence of norms in finite-dimensional spaces, this also implies that $\alpha_k(\hat{f})$ is Lipschitz in the classical (2-norm) sense with a Lipschitz-constant that equals to the Lipschitz-constant of $\hat{f}$ up to a factor depending on the dimension. □

We also want to *resample* functions $\hat{f} : D^\varepsilon \to \mathbb{R}^n$ where $D^\varepsilon := \varepsilon\mathbb{Z}^k \cap D$ for $D \subset \mathbb{R}^k$.

Definition 42. For a *sampling interval* $\varepsilon > 0$, we want to define a *multilinear resampling operator* $\alpha_k^\varepsilon : (\varepsilon\mathbb{Z}^k \to \mathbb{R}) \to (\mathbb{R}^k \to \mathbb{R})$. We utilize that $\hat{f}_\varepsilon : \varepsilon\mathbb{Z}^k \to \mathbb{R}$ can be rescaled to $\hat{f}_\varepsilon \circ (x \mapsto \varepsilon \cdot x) : \mathbb{Z}^k \to \mathbb{R}$ and reuse Definition 40 to define $\alpha_k^\varepsilon : (\varepsilon\mathbb{Z}^k \to \mathbb{R}) \to (\mathbb{R}^k \to \mathbb{R})$ as follows:

$$\alpha_k^\varepsilon(\hat{f}_\varepsilon)(x) := \alpha_k(\hat{f}_\varepsilon \circ (x \mapsto \varepsilon \cdot x))(\frac{1}{\varepsilon}x)$$

Let $D \subset \mathbb{R}^k$ be arbitrary and $D^\varepsilon := \varepsilon\mathbb{Z}^k \cap D$. We further define a resampling operator $\alpha_{k,D}^\varepsilon : (D^\varepsilon \to \mathbb{R}) \to (D \to \mathbb{R})$ via

$$\alpha_{k,D}^\varepsilon(\hat{f}) = \alpha_k^\varepsilon(\bar{f})|_D,$$

where $\bar{f} : \varepsilon\mathbb{Z}^k \to \mathbb{R}$ extends $\hat{f} : D_\varepsilon \to \mathbb{R}$. An exact definition of how to extend the domain is not of importance for the succeeding proofs. A suitable extension that preserves Lipschitz continuity (adapted from McShane (1934)) is setting

$$\bar{f}(x) = \inf_{y \in D^\varepsilon}\{\hat{f}(y) + \mathrm{Lip}_{\max}(\hat{f})\|x - y\|_{\max}\} \quad \text{for any } x \in \varepsilon\mathbb{Z}^k.$$

Then clearly $\bar{f}|_{D^\varepsilon} = \hat{f}$ and $\mathrm{Lip}_{\max}(\bar{f}) = \mathrm{Lip}_{\max}(\hat{f})$.

In the future, we will leave out the indices for the resampling operator and simply write $\alpha : (D^\varepsilon \to \mathbb{R}) \to (D \to \mathbb{R})$, since the used function is deducible from the domains.

For this extended definition, the preservation of Lipschitz continuity still holds:

Corollary 43. *Let $\hat{f} : D_\varepsilon \to \mathbb{R}$ be Lipschitz. Then also $\alpha^\varepsilon_{k,D}(\hat{f}) : D \to \mathbb{R}$ is Lipschitz and from Lemma 41, we can immediately deduce that*

$$\begin{aligned}\mathrm{Lip}_{\max}(\alpha^\varepsilon_{k,D}(\hat{f})) &= \mathrm{Lip}_{\max}(\alpha^\varepsilon_k(\bar{f})) = \mathrm{Lip}_{\max}(\alpha_k(\bar{f} \circ (x \mapsto \varepsilon x)) \circ (x \mapsto \frac{1}{\varepsilon}x)) \\ &= \frac{1}{\varepsilon} \cdot \mathrm{Lip}_{\max}(\bar{f} \circ (x \mapsto \varepsilon x)) = \frac{1}{\varepsilon} \cdot \varepsilon \cdot \mathrm{Lip}_{\max}(\bar{f}) = \mathrm{Lip}_{\max}(\bar{f}) = \mathrm{Lip}_{\max}(\hat{f})\end{aligned}$$

6.2. Discretization of continuous domains

Let us assume a DPM M over the ring $\mathbb{R}$ with arbitrary domains is given. In this section, we want to define the concept of an *approximating DPM* M^ε, which no longer requires continuous domains. M^ε will store only *sampled functions* with *sampling interval* $\varepsilon > 0$ in its registers. ε is a parameter that controls the coarseness of the approximation. Afterwards, we are going to discuss when and how the computation of the original DPM M can be approximated through a sequence of DPMs $(M^\varepsilon_n)_{n\in\mathbb{N}}$ for $\lim_{n\to\infty} \varepsilon_n \to 0$.

Let us first start with the definitions for function sampling and resampling.

Definition 44 (sampling interval, sample points, sampled function, resampling operator). Let $f : D \to \mathbb{R}$ with $D \subset \mathbb{R}^k$ be a function. For a *sampling interval* $\varepsilon > 0$ we define the *sample points* as $D^\varepsilon := \varepsilon\mathbb{Z}^k \cap D$ and call the discrete map $\hat{f} = f|_{D^\varepsilon}$ the *sampled function*. Furthermore, we call the map $\alpha : (D^\varepsilon \to \mathbb{R}) \to (D \to \mathbb{R})$ from Definition 42 a *resampling operator*.

The *Bilinear interpolation* is a prominent example of such a *resampling operator* in 2D. This interpolation method is built in within common shader models for GPUs.

Now we are ready to define the concept of an *approximating DPM*.

Definition 45 (approximating DPM). Let M be a DPM over the domains $\mathcal{D} = \{D_1, \ldots, D_n\}$ over the ring $\mathbb{R}$ with $D_i \subset \mathbb{R}^k$. For $\varepsilon > 0$, we call a machine M^ε with domains $\mathcal{D}^\varepsilon = \{D^\varepsilon_1, \ldots, D^\varepsilon_n\}$ an *ε–approximating DPM* with *resampling functions* $\alpha_i : (D^\varepsilon_i \to \mathbb{R}) \to (D_i \to \mathbb{R})$ of M if

- The registers of M^ε store discrete functions.
- The state space of M^ε is

  $$\mathcal{S}^\varepsilon_M = \bigtimes_{i\in[n]} \{f : D^\varepsilon_i \to R \mid f \text{ discrete function}\}.$$

- M^ε has the same *input space* $\mathcal{I}_M$ and *output space* $\mathcal{O}_M$ as M.

- M^ε has the same directed graph as M for its program flow. The associated computation maps are however adopted to the corresponding discrete functions:
  - The unique *input node* of M^ε has the input map $I^\varepsilon : \mathcal{I}_M \to \mathcal{S}^\varepsilon_M$ of M^ε, which is the sampled version of the input map $I : \mathcal{I}_M \to \mathcal{S}_M$ of M, i.e. $I^\varepsilon = I|_{\mathcal{S}^\varepsilon_M}$.
  - Each *output node* η of M with output map $O_\eta : \mathcal{S}_M \to \mathcal{O}_M$ induces an output node η^ε with the output map $O^\varepsilon_\eta = O_\eta \circ \alpha : \mathcal{S}^\varepsilon_M \to \mathcal{O}_M$ where $\alpha : \mathcal{S}^\varepsilon_M \to \mathcal{S}_M$ is the composition of all resampling functions α_i applied to each of the components of $\mathcal{S}^\varepsilon_M$.
  - The *branch nodes* of M are directly transferred to branch nodes of M^ε.

    Each computation node η of M has an associated computation node η and a map $g_\eta : \mathcal{S}_M \to \mathcal{S}_M$ out of `constant`, `project`, `copy`, `add`, `subtract`, `multiply`, `divide`, `ifelse`, and `compose`. If the computation node η of M was different from `compose`, then the corresponding node η' of M^ε gets assigned almost the same map $g_{\eta'} : \mathcal{S}^\varepsilon_M \to \mathcal{S}^\varepsilon_M$, which is only restricted to a potential smaller sampled domain. As in Definition 4, each of these maps is performed pointwise.

    The map `compose` $: \mathcal{S}_M \to \mathcal{S}_M$, which can be used by M, has been defined in Definition 4 as

    $$\texttt{compose}(\{i_1, \ldots, i_l\}, j, k) : \mathcal{S}_M \to \mathcal{S}_M, h \mapsto h[h_k \leftarrow h_j \circ (h_{i_1}, \ldots, h_{i_l})],$$

    where

    $$h_j \circ (h_{i_1}, \ldots, h_{i_l}) : D_{i_1} \times \cdots \times D_{i_l} \to \mathbb{R},$$
    $$(x_1, \ldots, x_l) \mapsto \begin{cases} h_j(h_{i_1}(x_1), \ldots, h_{i_l}(x_l)) & \text{if } (h_{i_1}(x_1), \ldots, h_{i_l}(x_l)) \in D_j \\ 0 & \text{otherwise} \end{cases}.$$

    The `compose`-operator should not be directly transferred to M^ε, because the domains D^ε_j potentially got substantially smaller and most accessed values might not coincide with a sampled point. Instead of `compose`, the map

    $$\texttt{resample}_\alpha(\{i_1, \ldots, i_l\}, j, k) : \mathcal{S}^\varepsilon_M \to \mathcal{S}^\varepsilon_M,$$
    $$h \mapsto h[h_k \leftarrow \alpha_j(h_j) \circ (h_{i_1}, \ldots, h_{i_l})]$$

    is used in M^ε for the resampling operator $\alpha_j : (D^\varepsilon_j \to \mathbb{R}) \to (D_j \to \mathbb{R})$. This avoids accesses on non-defined parts of the register $h_j : D^\varepsilon_j \to \mathbb{R}$; instead the values are obtained by from a resampled function $\alpha_j(h_j) : D_j \to \mathbb{R}$.

The evaluation of an approximating DPM and the input-output map $\phi^\varepsilon_M : \mathcal{I}_M \to \mathcal{O}_M$ is defined in the same manner as it has been done for common DPMs (see Definition 8).

6.3. A convergence theorem for approximating DPMs

It is easy to see whenever there are no nodes η with $g_\eta = \texttt{compose}$ in the program-flow of a DPM M (which means that even single values are not extracted from a register), then any approximating DPM performs the same computation as M; the component-wise operations are only performed on a smaller set. The output, if it is a function, is just re-sampled.

What about DPM that have nodes computing $g_\eta = \texttt{compose}$? Note that nodes of this kind are also needed if a value is just read from a non-trivial register at a given position (The model was defined in a way that the value $f[x]$ for $x : * \to \mathbb{R}$ and $f : \mathbb{R} \to \mathbb{R}$ is computed via $f \circ x$). Does

$$\lim_{\varepsilon \to 0} \phi_M^\varepsilon(x) = \phi_M(x)$$

for every DPM M and $x \in \Omega_M$ hold? Can we approximate every DPM operating on potential continuous domains arbitrary well through an approximating DPM that perform computations on discrete domains if we chose the sampling width ε only sufficiently small?

The answer is in general negative, as the following example shows. This section aims to extract reasons that in many cases forbid the convergence of approximating DPMs to the original DPM. Later in Theorem 54, we will show that the absence of these reasons is already sufficient to obtain convergence.

Example 46. The following DPM M with $\mathcal{D} = \{*, \mathbb{R}\}$ does not fulfill

$$\lim_{\varepsilon \to 0} \phi_M^\varepsilon(x) = \phi_M(x)$$

for every $x \in \Omega_M$, if M^ε uses linear interpolation for resampling:

```
   Input: $x \in \mathbb{R}$
   Output: 1 if $x^2 = 2$, otherwise 0
1  compute $f[x], f : \mathbb{R} \to \mathbb{R}$ everywhere as
2      return $x^2$
3  if $f[x] = 2$ then return 1 else return 0
```

Let M^ε be an approximating DPM over the domain $\mathcal{D} = \{*, \varepsilon\mathbb{Z}\}$ that uses *linear interpolation* for resampling. Clearly, we have $f[x] = x^2$ after execution of Line 2 and finally obtain $\phi_M(\sqrt{2}) = 1$. However, for a sequence $\varepsilon_n \to 0$ with $\varepsilon_n \in \mathbb{Q}$, we yield $\phi_M^{\varepsilon_n}(\sqrt{2}) = 0$ for every $\varepsilon_n > 0$ because all stored sampling points will attain rational values. The value $f[\sqrt{2}]$ will be approximated through $\alpha(f^\varepsilon)(\sqrt{2})$, i.e. through interpolation between two samples $(a, a^2), (b, b^2) \in \mathbb{Q}^2$ with $a < \sqrt{2} < b$. With $\lambda = \frac{\sqrt{2}-a}{b-a} \in (0,1)$ we yield:

$$\alpha(f^\varepsilon)(\sqrt{2}) = (1-\lambda)a^2 + \lambda b^2 = \left(1 - \frac{\sqrt{2}-a}{b-a}\right)a^2 + \frac{\sqrt{2}-a}{b-a}b^2 = \sqrt{2}(a+b) - ab,$$

which for $a, b \in \mathbb{Q}_{>0}$ is irrational and, in particular, $\alpha(f^{\varepsilon_n})(\sqrt{2}) \neq 2$. Thus M^{ε_n} will always traverse the else-branch for any input. In general, the convergence $\lim_{\varepsilon\to 0} \phi^\varepsilon_M(x) = \phi_M(x)$ can not hold for every $x \in \Omega_M$.

Here one might suspect that the reason for the non-converge was the fact that the branching for $f[x] = 2$ works only for $x = \sqrt{2}$ and not for any neighborhood in the state space of M. The concept of a *stable computation path* should prevent such cases.

Definition 47 (stable computation path). For a DPM M with halting-set Ω_M we say that the input $x \in \Omega_M$ *induces a stable computation path*, if sufficiently small perturbations in the state trajectory $x^0, x^1, \dots, x^T \in \mathcal{S}_M$ in those entries that are influenced by resampled values lead to the same *computation path* $\eta^0, \eta^1, \dots, \eta^T \in V$ of x (see Definition 8), i.e. there exists $\varepsilon > 0$ such that for every $k \in [T-1]$ the computation endomorphism $H : V \times \mathcal{S}_M \to V \times \mathcal{S}_M$ fulfills $\pi_V(H(\eta^k, \tilde{x}^k)) = \eta^{k+1}$ for every $\tilde{x}^k \in \mathcal{S}_M$ such that $\|\tilde{x}^k - x^k\|_\infty < \varepsilon$ and $\tilde{x}^k = x^k_i$ for all registers that are not influenced by resampling. This is equivalent to the following statement: "Sharp branching conditions are avoided on unsafe values during the computation".

However, alone the property of x to have a stable computation path does not suffice to yield convergence through approximating DPMs: Even if the computational path for every $x \in \Omega_M$ is the same, then still the convergence $\lim_{\varepsilon\to 0} \phi^\varepsilon_M(x) = \phi_M(x)$ might not hold as it can be seen in the following example where the branching of the program flow has been shifted to the register-inherent branching within a computation node:

Example 48. The following example has the same computational path for every input:

```
  Input: $x \in \mathbb{R}$
  Output: 1 if $x^2 = 2$, otherwise 0
1 compute $f[x], f : \mathbb{R} \to \mathbb{R}$ everywhere as
2     return $x^2$
3 compute $\chi[x], \chi : \mathbb{R} \to \mathbb{R}$ everywhere as
4     if $x = 2$ then return 1 else return 0
5 return $\chi[f[x]]$
```

With the same argumentation as in Example 46, the given DPM can not be approximated with a linear resampling function:

$$M_\phi(\sqrt{2}) = 1 \neq 0 = M^\varepsilon_\phi(\sqrt{2}) \quad \text{for } \varepsilon \in \mathbb{Q}$$

So, aside having a stable computation path, also the data-parallel if-else functions should be smooth. A restricted set of branching functions are the max and min-functions, which are a safe alternative to define continuous function piece-wise:

Definition 49. We say that a DPM M shows *soft parallel branching*, if every occurring node computing $\texttt{ifelse}(l,i,j,k) : \mathcal{S}_M \to \mathcal{S}_M$, i.e. the command $h_k \leftarrow \text{if}(h_l \geqslant 0, h_i, h_j)$, decodes either the component-wise max or min-function of two-registers. In other words, we demand that any node of M computing $\texttt{ifelse}(l,i,j,k)$ fulfills at running time either $h_l = h_i - h_j$ or $h_l = h_j - h_i$.

We will see in Theorem 54, that *soft parallel branching* and a *stable computation path* already implies convergence of approximating DPMs provided that only compact domains are used.

In the process of proving the convergence, a difficulty arises from the `compose` operator. The pointwise convergence of two sequences of functions does not imply even point-wise convergence of the composition of the sequences, even if all involved functions are continuous. Already the simpler problem of plugging a converging sequence $(x_n)_{n\in\mathbb{N}}$ in a converging family of continuous functions $f_n : \mathbb{R} \to \mathbb{R}$ with a pointwise continuous limit $f(x) = \lim_{n\to\infty} f_n(x)$ does not imply $\lim_{n\to\infty} f_n(x_n) = f(x)$. Consider for instance the sequence of continuous functions $f_n(x) := \max\{0, 1 - |nx - 1|\}$, a flat line with a peak of height 1 at position $\frac{1}{n}$. For every $x \in \mathbb{R}$, it can be proved that $\lim_{n\to\infty} f_n(x) = 0 =: f(x)$ (the peak moves past all points $x > 0$). However, for $x_n = \frac{1}{n}$ we have

$$\lim_{n\to\infty} f_n(x_n) = 1 \neq 0 = f(0) = f(\lim_{n\to\infty} x_n)$$

Nevertheless, for *uniform convergence* instead of pointwise convergence, we have the following property:

Lemma 50. *Let $f_n : D \to \mathbb{R}$ be a sequence of functions converging* uniformly *to a continuous function $f : D \to \mathbb{R}$. Furthermore let a converging sequence $x_n \to x$ in D be given. Then*

$$\lim_{n\to\infty} f_n(x_n) = f(x)$$

Proof. $\lim_{n\to\infty}|f_n(x_n) - f(x)| \leqslant \lim_{n\to\infty}|f_n(x_n) - f(x_n)| + \lim_{n\to\infty}|f(x_n) - f(x)| = 0 + 0$. The first convergence holds due to the uniform convergence of $f_n \to f$ and the second follows from the continuity of f. □

A direct proof of uniform convergence of a sequence of functions is often more difficult than a proof of pointwise convergence. The following Lemma gives a criterion when pointwise convergence can be upgraded to uniform convergence:

Lemma 51. *Let $D \subset \mathbb{R}^k$ be compact and $S \subset D$ dense. A sequence $(f_n : D \to \mathbb{R})_{n=0}^{\infty}$ of L-Lipschitz functions converges* uniformly *to a continuous $f : D \to \mathbb{R}$ if $f_n|_S$ converges* pointwise *to $f|_S$.*

Proof. Clearly, all the Lipschitz functions f_n are continuous. We will show that the sequence $(f_n)_{n=0}^{\infty}$ is Cauchy in the complete space $C(D, \mathbb{R})$.

Let $\varepsilon > 0$. We set $\delta = \frac{\varepsilon}{3L}$. The family $\{B_\delta(s) \mid s \in S\}$ covers D and due to the compactness of D there is a finite subcover $B_\delta(s_1), \ldots, B_\delta(s_N)$ of D with $s_1, \ldots, s_N \in S$. Since $\lim_{n\to\infty} f_n(s_k) = f(s_k)$ for every $k \in [N]$, there is a $n \in \mathbb{N}$ such that

$$|f_n(s_k) - f_m(s_k)| < \frac{\varepsilon}{3} \quad \text{for every } m > n \text{ and } k \in [N].$$

For any $x \in D$, there is some $k \in [N]$ with $x \in B_\delta(s_k)$ and we yield for $m > n$

$$|f_n(x) - f_m(x)| \leqslant |f_n(x) - f_n(s_k)| + |f_n(s_k) - f_m(s_k)| + |f_m(s_k) - f_m(x)| \leqslant L\delta + \frac{\varepsilon}{3} + L\delta = \varepsilon.$$

Let $\tilde{f} \in C(D, \mathbb{R})$ be the limit of the Cauchy sequence $(f_n)_{n=0}^{\infty}$. The pointwise convergence of $f_n|_S$ to $f|_S$ implies $\tilde{f}|_S = f|_S$. Since S is dense in D, the continuous functions $\tilde{f}$ and f coincide. $\square$

The restriction on a compact D is not a big problem for us because we are aiming to use rasterization in order to use textures of limited extent. Lemma 51 invites to prove pointwise convergence.

A notion of pointwise convergence of sampled functions is useful. To make pointwise convergence for the well-defined discrete functions we need to ensure that the once added sampling points remain existent for the further refinements.

Definition 52 (refining, point wise convergence of refining sequence). We call a sequence of discrete sampled functions $f_n : D_n \to \mathbb{R}$ *refining* if $D_m \supset D_n$ for $m > n$ and $\bigcup_{n\in\mathbb{N}} D_n$ is dense in D, i.e. for every $\varepsilon > 0$ there is a $n \in \mathbb{N}$ such that $D \subset D_n + B_\varepsilon$.

A canonical example for a refining sequence is given with the domains $D_n = \frac{1}{2^n}\mathbb{Z}^k \cap D$ and the sequence of discrete functions $f_n : D_n \to \mathbb{R}$.

For a refining sequence, we say that the sequence of discrete functions $f_n : D_n \to \mathbb{R}$ *converges pointwise to a function $f : D \to \mathbb{R}$* when for every $m \in \mathbb{N}$ and $x \in D_m$ the sequence $(f_n(x))_{n=m}^{\infty}$ converges to $f(x)$.

Under certain conditions, pointwise convergence of a refining sequence of discrete functions can be upgraded to uniform convergence of the resampled functions.

Lemma 53. *Let $D \subset \mathbb{R}^k$ be compact with $D = \overline{\operatorname{Int} D}$ (equivalently, D is the closure of a bounded open set in $\mathbb{R}^k$). Let $(f_n : D_n \to \mathbb{R})_{n=0}^{\infty}$ with $D_n = d_n\mathbb{Z}^k \cap D$ be a refining sequence* *of sampled L-Lipschitz functions that* converges pointwise *to a continuous function* $f : D \to \mathbb{R}$. *Then for the sequence of resampling functions* $\alpha_n : (D_n \to \mathbb{R}) \to (D \to \mathbb{R})$, *the sequence of resampled functions* $(\alpha_n(f_n))_{n=0}^{\infty}$ converges uniformly *to* f *on* D.

Proof. According to Corollary 43, for every $n \in [N]$, $\alpha_n(f_n)$ has the same Lipschitz-constant $C \cdot L$ for some fixed $C \in \mathbb{R}$ and for every $s \in D_n$ we have $\alpha_n(f_n)(s) = f_n(s)$. Therefore, on the set $S = \bigcup_{n=0}^{\infty} D_n$, the $(C \cdot L)$-Lipschitz sequence $(\alpha_n(f_n))_{n=0}^{\infty}$ converges pointwise to f. The set $S \cap \operatorname{Int} D$ is dense in the open set $\operatorname{Int} D \subset \mathbb{R}^k$. Thus S is also dense in $D = \overline{\operatorname{Int} D}$. By Lemma 51, $(\alpha_n(f_n))_{n=0}^{\infty}$ converges uniformly to f. ☐

With these properties in mind, we can formulate and prove the following Theorem that gives us a criterion when a DPM can be approximated through a sequence of sampling DPMs.

Theorem 54. *Let* M *be a DPM with* compact *domains only and* $x \in \Omega_M$ *a Lipschitz input that induces a* stable computation path, *shows* soft parallel branching *and* avoids division by zero.

Furthermore, let $(\varepsilon_n)_{n=0}^{\infty}$ *be a* refining *sequence with associated sequence of the approximating DPMs* M^{ε_n} *with arbitray resampling functions* α.

Then

- *for sufficiently small* ε_n, *the computational path of* M^{ε_n} *eventually coincides with the computational path of* M.
- *the state trajectory* $x^0, x^1, \ldots, x^T$ *of* M *then can be uniformly approximated with* $x^0_{\varepsilon_n}, x^1_{\varepsilon_n}, \ldots, x^T_{\varepsilon_n}$, *i.e. for any* $k \in [T] \cup \{0\}$ *the sequence of functions* $\alpha(x^k_{\varepsilon_n})$ *converges uniformly to* x^k *with* $n \to \infty$.
- *In particular,* $\phi_M^{\varepsilon_n}(x) : \mathcal{O}_M \to \mathbb{R}$ *converges uniformly to* $\phi_M(x)$.

In order to prove Theorem 54, we need another rather technical Lemma that can be used to show that such a DPM with compact domains stores during its execution Lipschitz functions with bounded Lipschitz constants only.

Lemma 55 (preservation of Lipschitz continuity). *Let* $f, g : D \to \mathbb{R}$ *and* $r : \tilde{D} \to D$, *be Lipschitz and* $D \subset \mathbb{R}^n$, $\tilde{D} \subset \mathbb{R}^m$.

Then $f + g, f - g, \max(f, g), \min(f, g) : D \to \mathbb{R}$, *and* $f \circ r : \tilde{D} \to \mathbb{R}$, *are Lipschitz.*

If we assume that $|f(x)|$ *and* $|g(x)|$ *are bounded, then* $f \cdot g : D \to \mathbb{R}$ *is Lipschitz. If we further assume that* $|g(x)|$ *is bounded from below, then also* $\frac{f}{g} : D \to \mathbb{R}$ *is Lipschitz.*

In particular, the Lipschitz-constants can be bounded as follows:

(a) $\operatorname{Lip}(f + g) \leqslant \operatorname{Lip}(f) + \operatorname{Lip}(g)$ *and* $\operatorname{Lip}(f - g) \leqslant \operatorname{Lip}(f) + \operatorname{Lip}(g)$.

(b) $\mathrm{Lip}(\min(f,g)) \leqslant \max\{\mathrm{Lip}(f), \mathrm{Lip}(g)\}$ *and* $\mathrm{Lip}(\max(f,g)) \leqslant \max\{\mathrm{Lip}(f), \mathrm{Lip}(g)\}$.

(c) $\mathrm{Lip}(f \circ r) \leqslant \mathrm{Lip}(f) \cdot \mathrm{Lip}(r)$.

(d) $\mathrm{Lip}(f \cdot g) \leqslant \bar{F} \cdot \mathrm{Lip}(g) + \bar{G} \cdot \mathrm{Lip}(f)$ *if* $|f(x)| \leqslant \bar{F}$ *and* $|g(x)| \leqslant \bar{G}$.

(e) $\mathrm{Lip}(\frac{f}{g}) \leqslant \frac{\bar{F}\mathrm{Lip}(g) - \bar{G}\mathrm{Lip}(f)}{\underline{G}^2}$ *if* $|f(x)| \leqslant \bar{F}$ *and* $0 < \underline{G} \leqslant |g(x)| \leqslant \bar{G}$ *for every* $x \in D$.

Proof. (a) $|(f \pm g)(x) - (f \pm g)(y)| \leqslant |f(x) - f(y)| + |g(x) - g(y)| \leqslant (\mathrm{Lip}(f) + \mathrm{Lip}(g))\|x - y\|$.

(b) It suffices to show $\mathrm{Lip}(\max(f,g)) \leqslant \max\{\mathrm{Lip}(f), \mathrm{Lip}(g)\}$, because together with

$$\min(f,g) = -\max(-f,-g)$$

and

$$\mathrm{Lip}(-h) = \mathrm{Lip}(h)$$

we yield from that

$$\mathrm{Lip}(\min(f,g)) = \mathrm{Lip}(-\max(-f,-g)) \leqslant \max\{\mathrm{Lip}(f), \mathrm{Lip}(g)\}.$$

If both $f(x) \geqslant g(x)$ and $f(y) \geqslant g(y)$ (or $f(x) \leqslant g(x)$ and $f(y) \leqslant g(y)$), the term $|\max(f,g)(x) - \max(f,g)(y)|$ is equal to $|f(x) - f(y)|$ (or $|g(x) - g(y)|$) and thus can be bounded by $\max\{\mathrm{Lip}(f), \mathrm{Lip}(g)\}\|x - y\|$. Otherwise, let without loss of generality, $f(x) > g(x)$ and $f(y) < g(y)$. With a trick of McShane (1934), we extend $f, g : D \to \mathbb{R}$ to $\bar{f}, \bar{g} : \mathbb{R}^n \to \mathbb{R}$ with preservation of the Lipschitz constants via $\bar{f}(x) = \inf_{y \in D}\{f(y) + \mathrm{Lip}(f)\|x - y\|\}$ and $\bar{g}(x) = \inf_{y \in D}\{g(y) + \mathrm{Lip}(g)\|x - y\|\}$. We have $(\bar{f} - \bar{g})(x) > 0$ and $(\bar{f} - \bar{g})(y) < 0$. Then by the intermediate value theorem, there exists a $z \in [x, y] \subset \mathbb{R}^n$ with $(\bar{f} - \bar{g})(z) = 0$, i.e. $\bar{f}(z) = \bar{g}(z)$. Then we yield,

$$\begin{aligned} |\max(f,g)(x) - \max(f,g)(y)| &= |f(x) - g(y)| = |\bar{f}(x) - \bar{f}(z)| + |\bar{g}(z) + \bar{g}(y)| \\ &\leqslant \mathrm{Lip}(\bar{f})\|x - z\| + \mathrm{Lip}(\bar{g})\|z - x\| \leqslant \max\{\mathrm{Lip}(f), \mathrm{Lip}(g)\}\|x - y\|. \end{aligned}$$

(c) $|(f \circ r)(x) - (f \circ r)(y)| = |f(r(x)) - f(r(y))| \leqslant \mathrm{Lip}(f)\|r(x) - r(y)\| \leqslant \mathrm{Lip}(f)\mathrm{Lip}(r)\|x - y\|$.

(d) $|(fg)(x) - (fg)(y)| \leqslant |f(x)g(x) - f(x)g(y)| + |f(x)g(y) - f(y)g(y)| \leqslant \bar{F}|g(x) - g(y)| + \bar{G}|f(x) - f(y)| \leqslant (\bar{F} \cdot \mathrm{Lip}(g) + \bar{G} \cdot \mathrm{Lip}(f))\|x - y\|$.

(e) $|\frac{f}{g}(x) - \frac{f}{g}(x)| \leqslant \frac{|f(x)g(y) - f(x)g(x)| + |f(x)g(x) - f(y)g(x)|}{|g(x)g(y)|} \leqslant \frac{\bar{F}\mathrm{Lip}(g) - \bar{G}\mathrm{Lip}(f)}{\underline{G}^2}\|x - y\|$.

□

Now we have all the ingredients to prove Theorem 54:

Proof of Theorem 54. By induction over k, we show that any $k \in \{0, 1, \ldots, T\}$ fulfills the following **Induction Hypothesis:**

(a) The traversed computation path $(\eta^l_{\varepsilon_n})^k_{l=0}$ of M^{ε_n} eventually coincides with the computation path $(\eta^l)^k_{l=0}$ of M, given the input x, i.e. there exits $n_k \in \mathbb{N}$ such that for any $n \geqslant n_k$ we have $(\eta^l_{\varepsilon_n})^k_{l=0} = (\eta^l)^k_{l=0}$.

(b) The sequence $(x^k_{\varepsilon_n} \in \mathcal{S}^{\varepsilon_n}_M)^\infty_{n=n_k}$ of state trajectories consists of L_k-Lipschitz functions for some fixed $L_k \in \mathbb{R}$.

(c) For each register $h : D \to \mathbb{R}$ stored in $x^k \in \mathcal{S}_M$, which is approximated through $h^\varepsilon_n : D^{\varepsilon_n} \to \mathbb{R}$ in M^{ε_n}, the sequence $(h^k_{\varepsilon_n})_{n=n_k}$ converges pointwise to h.

(d) $\alpha(x^k_{\varepsilon_n})^\infty_{n=n_k}$ converges uniformly to x^k with $n \to \infty$.

$x^k_{\varepsilon_n}$ is composed of a finite number of registers. Hence, (d) always follows from (b) and (c) through the application of Lemma 53. So, we will show (a), (b), (c) only.

Base case $k = 0$**:** (a) holds already for $n_k = 0$ since $\eta^0 = \eta^0_{\varepsilon_n}$ is the starting node. (b) and (c) hold since the input $x \in \Omega_M$ is assumed to be Lipschitz and the input map $I^{\varepsilon_n} : \mathcal{I}_M \to \mathcal{S}^{\varepsilon_n}_M$ does nothing then sampling the input end embedding it into $\mathcal{S}^{\varepsilon_n}_M$.

Induction step $k > 0$**:** Let $k \in [T]$. We assume that the Induction Hypothesis above is fulfilled for $k-1$. By (a), $\eta^{k-1} = \eta^{k-1}_{\varepsilon_n}$ for any $n \geqslant n_k$. (d) gives us uniform convergence of $\alpha(x^k_{\varepsilon_n})$ to x^k. Branching is made on a pointwise evaluation of $\alpha(x^k_{\varepsilon_n})$ and by assumption the computation path is stable, thus $\eta^k_{\varepsilon_n} = \pi_V(H^{\varepsilon_n}(\eta^{k-1}_{\varepsilon_n}, \alpha(x^k_{\varepsilon_n})) = \pi_V(H(\eta^{k-1}, \alpha(x^k_{\varepsilon_n})) = \eta^k$ will hold for a sufficiently large $n_k \geqslant n_{k-1}$. (a) is proved for k. We are still missing proofs for (b) and (c).

In step k, only one register h is modified by a map g_{η^k}. It suffices to show (b) and (c) for the registers $h^{\varepsilon_n} : D^{\varepsilon_n} \to \mathbb{R}$. i.e., we want to show that (b) $(h^{\varepsilon_n})_{n=n_k}\infty$ consists of Lipschitz functions with bounded Lipschitz-constant and (c) that for every $s \in S := \bigcup^\infty_{n=n_k} D^{\varepsilon_n}$ the convergence $\lim_{n\to\infty} h^{\varepsilon_n}(s) = h(s)$ holds.

There are several different cases for g_{η^k}.

g_{η^k} **is `constant` or `project`:** Let the "command" be $h \leftarrow \alpha$ or $h \leftarrow \pi_k(x)$. Then the sampling points of h^{ε_n} can be computed directly and h^{ε_n} coincides with h on D^{ε_n}. Clearly, every such h^{ε_n} is constant or 1-Lipschitz.

g_{η^k} **is copy:** Let the command be $h \leftarrow g$. On the approximating DPM $h^\varepsilon_n \leftarrow g^\varepsilon_n$ is computed. By induction, the properties (b) and (c) for k are obtained from (b) and (c) for $k-1$.

g_{η^k} is **add**, **subtract**, **multiply**, **divide** or **ifelse**: Let us assume that the register $h^{\varepsilon_n} : D_n \to \mathbb{R}$ is recomputed in step k based on the registers $f^{\varepsilon_n} : D_n \to \mathbb{R}$ and $g^{\varepsilon_n} : D_n \to \mathbb{R}$.

If the command is to compute $h \leftarrow f \odot g$ with $\odot \in \{+, -, \cdot, /\}$ or $h \leftarrow \operatorname{op}(f, g)$ with $\operatorname{op} \in \{\min, \max\}$, then for a fixed sample point $s \in S = \bigcup_{n=n_k}^{\infty} D_n$, the convergences

$$\begin{aligned}\lim_{n\to\infty} h^{\varepsilon_n}(s) &= \lim_{n\to\infty} \left(f^{\varepsilon_n}(s) \odot g^{\varepsilon_n}(s)\right)\\ &= \lim_{n\to\infty} f^{\varepsilon_n}(s) \odot \lim_{n\to\infty} g^{\varepsilon_n}(s) = f(s) \odot g(s) = h(s)\end{aligned}$$

or, equivalently,

$$\begin{aligned}\lim_{n\to\infty} h^{\varepsilon_n}(s) &= \lim_{n\to\infty} \operatorname{op}\left(f^{\varepsilon_n}(s), g^{\varepsilon_n}(s)\right)\\ &= \operatorname{op}\left(\lim_{n\to\infty} f^{\varepsilon_n}(s), \lim_{n\to\infty} g^{\varepsilon_n}(s)\right) = \operatorname{op}\left(f(s), g(s)\right) = h(s)\end{aligned}$$

hold due to the continuity $+, -, \cdot, /$, min and max and Induction Hypothesis (c).

The family $(h^{\varepsilon_n})_{n=n_k}^{\infty}$ will have a shared Lipschitz constant due to the application of Lemma 55. Note that the functions f^{ε_n} and g^{ε_n} are bounded because of the uniform convergence of $\alpha(f^{\varepsilon_n})$ to f and $\alpha(g^{\varepsilon_n})$ to g, which both are continuous functions on a compact domain. In case of the division $h \leftarrow \frac{f}{g}$, also $|\alpha(g^{\varepsilon_n})|$ will eventually be bounded from below due to the avoidance of the division by zero.

g_{η^k} is **composite**: Let the command be $h \leftarrow f \circ (g_1, \dots, g_l)$. On the approximating DPMs M^{ε_n}, we compute the values $\alpha(f^{\varepsilon_n}) \circ (g_1^{\varepsilon_n}, \dots, g_l^{\varepsilon_n})$ on the sample points $D_f^{\varepsilon_n}$ only. First, we want to show (c), i.e. pointwise convergence on the sample points. Let $s \in D_h^{\varepsilon_m}$ be a sampling point. For $n \geqslant m$ we define $y_n^s = (g_1^{\varepsilon_n}(s), \dots, g_l^{\varepsilon_n}(s))$. By Induction Hypothesis (c), there is a limit point $y^s \in \mathbb{R}^l$ with

$$\lim_{n\to\infty} y_n^s = (g_1(s), \dots, g_l(s)) = y^s$$

Also by Induction Hypothesis (d), the resampled functions $\alpha(f^{\varepsilon_n})$ converge *uniformly* to f. Thus we can apply Lemma 50: We yield the following pointwise convergence on sample points:

$$\begin{aligned}\lim_{n\to\infty} h^{\varepsilon_n}(s) = \lim_{n\to\infty} \left(\alpha(f^{\varepsilon_n}) \circ (g_1^{\varepsilon_n}, \dots, g_l^{\varepsilon_n})\right)(s) &= \lim_{n\to\infty} \alpha(f^{\varepsilon_n})(y_n^s)\\ &= f(y^s) = h(s).\end{aligned}$$

Why does (b) hold? The families $(f^{\varepsilon_n})_{n=n_k}^{\infty}$ and $(g_1^{\varepsilon_n}, \dots, g_l^{\varepsilon_n})_{n=n_k}^{\infty}$ have a bounded Lipschitz constants by Induction Hypothesis (b). According to the Corollary 43, $(\alpha(f^{\varepsilon_n}))_{n=n_k}^{\infty}$ has a bounded Lipschitz constant and thus Lemma 55 gives a common Lipschitz constant for the family

$$(h^{\varepsilon_n})_{n=n_k}^{\infty} = (\alpha(f^{\varepsilon_n}) \circ (g_1^{\varepsilon_n}, \dots, g_l^{\varepsilon_n}))_{n=n_k}^{\infty}.$$

With this all cases for computations g_{η^k} are covered, and the induction is finished.

□

Let us give an example of an application of Theorem 54.

Example 56. (fast approximations of the filled-in Julia set) Let $f : \mathbb{C} \to \mathbb{C}$ be a continuous complex function with a *bailout radius* R, i.e. $|z| > R$ implies $|f(z)| > |z|$. For instance, for the Julia-map $f(z) = z^2 + c$, the value $R = \frac{1}{2} + \sqrt{\frac{1}{4} + |c|}$ is a valid bailout radius, because for $|z| > R$ we have

$$\begin{aligned}|f(z)| - |z| = |z^2 + c| - |z| \geqslant |z|^2 - |z| - |c| \\ = \left(|z| - \left(\frac{1}{2} + \sqrt{\frac{1}{4} + |c|}\right)\right) \cdot \left(|z| - \left(\frac{1}{2} - \sqrt{\frac{1}{4} + |c|}\right)\right) > 0.\end{aligned}$$

Every $z \in \mathbb{C}$ with $|f^m(z)| > R$ for some $m \in \mathbb{N}$ fulfills $\lim_{n\to\infty}|f^n(z)| = \infty$, because if the sequence $|f^n(z)|_{(n\in\mathbb{N})}$ was bounded by some $\hat{R} > R$, then there would be some accumulation point $a = \lim_{k\to\infty} f^{n_k}(z)$ in the compact disc $\{z \in \mathbb{C} \mid R \leqslant |z| \leqslant \hat{R}\}$. This would contradict the bailout property of f, because

$$|a| < |f(a)| = |f(\lim_{k\to\infty} f^{n_k}(z))| = |\lim_{k\to\infty} f^{n_k+1}(z)| \leqslant |\lim_{k\to\infty} f^{n_{k+1}}(z)| = |a|$$

According to that, the set of all points with $\lim_{n\to\infty} f^n(z) \neq \infty$ consists of those points with an orbit staying within the bailout radius. This set is called the *filled-in Julia set*. Thus $f^{-m}(D_R(0))$ where $D_R(0) = \{z \in \mathbb{C} \mid |z| \leqslant R\}$ for large m approximates the filled-in Julia set. In fact, the sequence $f^{-m}(D_R(0))$ converges with respect to the Hausdorff-Metric to the filled-in Julia set (Montag, 2014).

We want to compute the set $f^{-2^n}(D_R(0))$ efficiently through an *on the GPU approximable* DPM over the domains $\mathcal{D} = \{*, [\tilde{R}, \tilde{R}]^2\}$ with $\tilde{R} > R$. The set $f^{-2^n}(D(0, R))$ contains all points $z \in \mathbb{C}$ whose iteratives $z, f(z), f^2(z), f^3(z), \dots, f^{-2^n}(z)$ never leave the bailout-radius R.

In order to compute $f^{-2^n}(D(0, R))$, it suffices to study the modified function

$$\begin{aligned}\tilde{f} : [-\tilde{R}, \tilde{R}]^2 &\to [-\tilde{R}, \tilde{R}]^2 \\ z &\mapsto \texttt{clamp}(f(z)),\end{aligned}$$

where

$$\begin{aligned}\texttt{clamp} : \mathbb{C} &\to [-\tilde{R}, \tilde{R}]^2 \\ x + iy &\mapsto \min(\tilde{R}, \max(-\tilde{R}, x)) + i \min(\tilde{R}, \max(-\tilde{R}, y)).\end{aligned}$$

We identify $[-\tilde{R}, \tilde{R}]^2$ with the corresponding rectangular domain in $\mathbb{C}$. The functions f and $\tilde{f}$ coincide on the domain $f^{-1}([-\tilde{R}, \tilde{R}]^2) \cap [-\tilde{R}, \tilde{R}]^2$ and we have the bailout property $z \notin D_R(0) \Rightarrow g(z) \notin D_R(0)$ for both $g \in \{f, \tilde{f}\}$. It is equivalent to $g(z) \in D_R(0) \Rightarrow z \in D_R(0)$ and $g^{-1}D_R(0) \subset D_R(0)$. By iteratively applying g^{-1} on both sides of $g^{-1}D_R(0) \subset D_R(0)$ we yield the chain,

$$[-\tilde{R}, \tilde{R}]^2 \supset D_R(0) \supset g^{-1}D_R(0) \supset g^{-2}D_R(0) \supset \dots$$

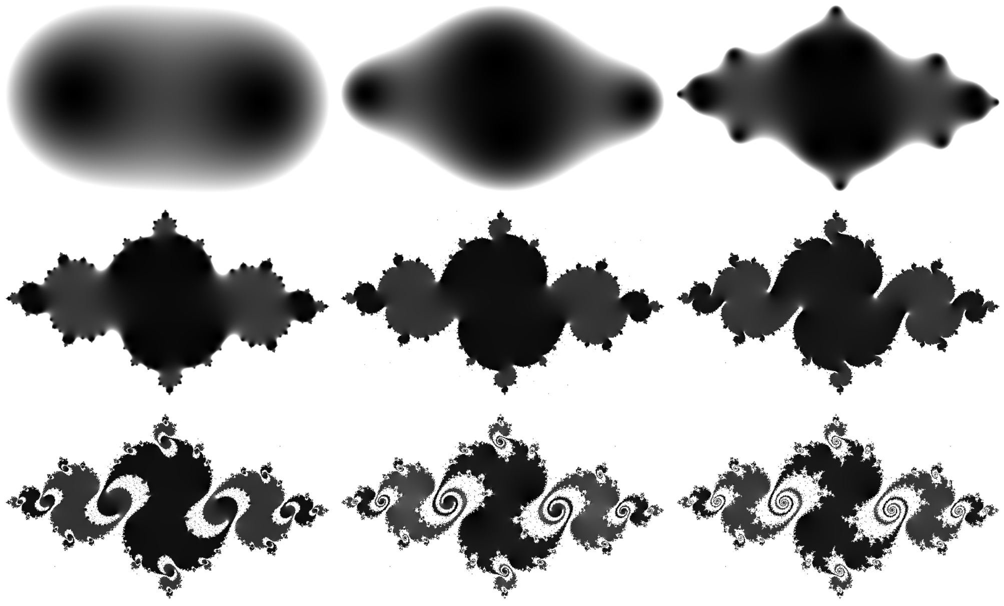
From this follows

$$f^{-2^n} D_R(0) = \tilde{f}^{-2^n} D_R(0).$$

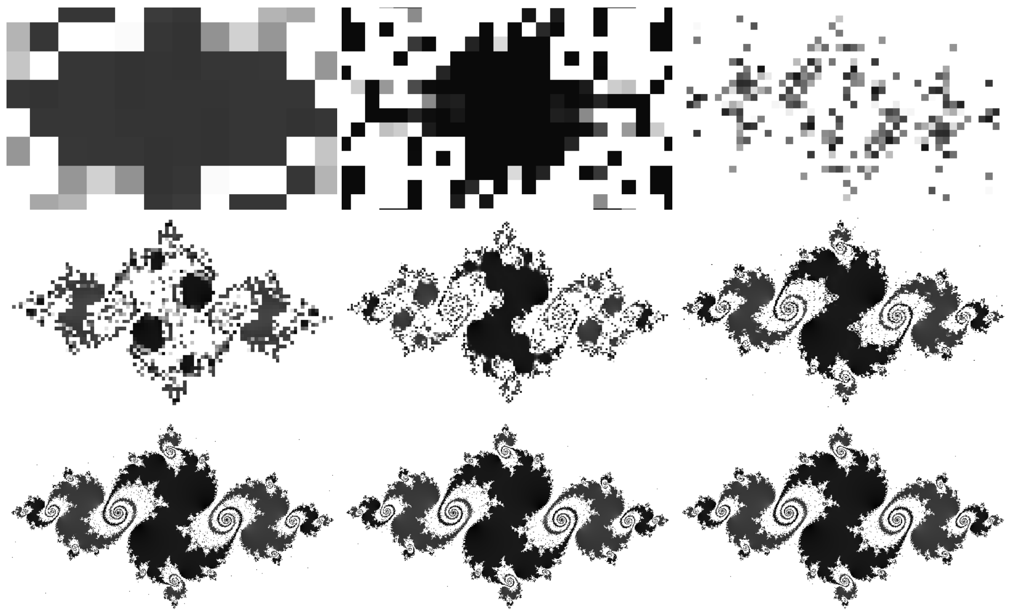
The $z \in \mathbb{C}$ contained in $\tilde{f}^{-2^n}(D_R(0))$ are those $z \in [-\tilde{R}, \tilde{R}]^2$ that have $\tilde{f}^{2^n}(z) \in D_R(0)$. The iterates of $\tilde{f} : [-\tilde{R}, \tilde{R}]^2 \to [-\tilde{R}, \tilde{R}]^2$ can be computed with the idea of Algorithm 11. We yield Algorithm 19 that computes $R^2 - |\tilde{f}^{2^n}(z)|^2$ for every $z \in [-\tilde{R}, \tilde{R}]^2$. This value is non-negative iff $z \in \tilde{f}^{-2^n} D_R(0) = f^{-2^n} D_R(0)$.

Algorithm 19: An approximable DPM with $\mathcal{D} = \{*, [-\tilde{R}, \tilde{R}]^2\}$ iterating exponential steps of the Julia map $f(z) = z^2 + c$

```
Input: an integer $n \in \mathbb{N}$, a parameter $c \in \mathbb{C}$ and a bailout radius $R \in \mathbb{R}_{<\tilde{R}}$ such
       that $|z| > R \Rightarrow |f(z)| > |z|$
Output: A continous function $\chi : [-R, R]^2 \to \mathbb{R}$ with
        $\chi(z) > 0 \Leftrightarrow f^{2^n}(z) \in D(R, 0) \Leftrightarrow z \in f^{-2^n}(D(R, 0))$
Function clamp $(x)$
    return $\min(\tilde{R}, \max(-\tilde{R}, x))$
compute $g[x, y], g : [-\tilde{R}, \tilde{R}]^2 \to \mathbb{R}^2$ everywhere as
    $z \leftarrow (x + i \cdot y)^2 + c$
    return $(\texttt{clamp}(\mathrm{Re}(z)), \texttt{clamp}(\mathrm{Im}(z)))$ // now $g = \tilde{f}$
repeat $n$ times
    $g \leftarrow g \circ g$ // after $k$ iterations, $g = \tilde{f}^{2^k}$
compute $\chi[x, y], g : [-\tilde{R}, \tilde{R}]^2 \to \mathbb{R}$ everywhere as
    return $R^2 - \|g(x, y)\|^2$
return $\chi$
```



(a) The sequence of the images χ for $n \in \{0, \dots, 8\}$ visualizing the dynamic of f^{2^n}. Internally, textures of size 4096 × 4096 were used for the computation of the images.



(b) Approximations for $n = 8$ using shader programs with increasing internal texture sizes: 16 × 16, 32 × 32, 64 × 64, 128 × 128, 256 × 256, 512 × 512, 1024 × 1024, 2048 × 2048 and 4096 × 4096. According to Theorem 54, the corresponding infinite sequence of images converges uniformly to the ideal limit image.

Figure 6.1.: Approximations of $\chi(z) = R^2 - \|f^{2^n}(z)\|^2$ with $f(z) = z^2 - 0.76 + i \cdot 0.07$ according to Algorithm 19. Non-positive values of χ are colored in white and positive values are colored in gray. The higher the value, the darker the color.

Theorem 54 can be applied for Algorithm 19. Thus, for a fixed input $n \in \mathbb{N}$ and $c \in \mathbb{C}$, a sequence of realizable approximating DPM using rasterized images produces a sequence of outputs converging uniformly to the idealized output.

Applet ▷6 contains a fragment shader based implementation of Algorithm 19. Internally, textures of finite resolution store the intermediate data and data points are accessed via bi-linear interpolation. Screenshots of the first steps of its iteration are displayed within Figure 6.1a. The computation of f^{2^n} requires $O(n)$ parallel operations and thus is exponentially faster than the conventional approach where the iterates of the Julia-map are computed for each pixel independently.

However, for precision reasons, a sufficiently high resolution is essential for these computations. According to Theorem 54, uniform convergence for fixed input to the output function of the idealized program can be achieved by iteratively doubling the internal resolution. This relationship is depicted in Figure 6.1b. Observe the artefacts outside of the filled in Julia sets in lower resolutions. These artefacts arise from interpolating coordinates out of the non-convex set $\mathbb{C} \setminus D(R, 0)$ to a coordinate in $D(R, 0)$. The artefacts vanish by choosing higher resolutions because then the corresponding values in general lie closer to each other and thus the convex combination of the values stays more likely within the domain $\mathbb{C} \setminus D(R, 0)$.

Chapter 7.

Programming GPUs

In this chapter it will be discussed how GPUs can be programmed in a similar way as a DPM. It relies on the work published by Montag and Richter-Gebert (2018).

Here we present a concept to embed shader programming seamlessly within a high-level (scripting) programming environment. The high-level programming environment should reflect a programmable DPM as presented in Section 2.2. Code that modifies registers with trivial domain only, can be computed straightforwardly on the CPU. What should happen with **compute everywhere**-statements? These parts of the high-level programming language are translated into shader programming language for the GPU. With the use of only one programming language, the development of complex (mathematical) visualizations and fast-prototyping of general-purpose computations on graphics processing units (GPGPU) applications is eased. We address the challenge of the automatic translation of a high-level programming language to a shader language of the GPU.

The presented approach can be used in mathematical visualization software, such as dynamic geometry software, and we have implemented a sample system. We have built a sample implementation for the dynamic geometry software CindyJS. This enables an interplay with geometry and GPU computations.

To maintain platform independence and the possibility to use our technology on modern devices, we focus on a realization through WebGL.

7.1. Introduction

Graphics processing units (GPUs) open up many new possibilities. However, extensive programming expertise and additional development effort are required to program them. In the first part of this chapter, we propose a concept to overcome this hurdle by seamlessly embedding shader programming language for the GPU within a high-level (scripting) programming language for rapid software prototyping. The concept includes the formal process of a symbolic transcompilation from a high-level language to a GPU shading language.

Richter-Gebert and Kortenkamp (2010) demonstrated that dynamic geometry software systems (DGS) equipped with a programming environment could be used advantageously for many mathematical scenarios. In the second part of this chapter, we will show how such an environment can be extended to utilize the GPU by the formal methods developed in the first part of the project. It provides the mathematician with a relatively easy-to-use tool for the following tasks, among many others: finding certain mathematical conjectures experimentally, the rapid prototyping of mathematical algorithms in parallel, building several visualizations of, for example, algebraic surfaces, fractals, fluid simulations and particle simulation within the framework of a classical DGS. This method is available inside the browser and on modern devices such as tablets and mobile phones. Additionally, the field of mathematics education can greatly benefit from the seamless integration of GPU programming into DGS.

7.1.1. Technical background

Computations on modern devices can, in general, be executed either on the CPU (central processing unit) or GPU (graphics processing unit). While CPUs were designed for sequential general-purpose computations, GPUs were initially invented to accelerate the graphics output of the computer. In order to obtain a high degree of flexibility in designing the graphical operations, programmable shader units were introduced within the GPU rendering pipeline.

Shaders can be considered as little programs which the GPU can execute massively in parallel. According to Owens et al. (2007), a typical GPU rendering pipeline consists of a vertex and fragment shader (among possibly other shaders). Conventionally, the vertex shader performs operations on the vertices of a three-dimensional mesh, while the fragment shader – sometimes also called as pixel shader – computes the color for every single pixel. Soon, it was discovered that these programmable shaders open the door for general-purpose computations on the GPU (GPGPU), that were conventionally performed on the CPU. In addition to the generation of images, a shader can also be used whenever the execution of the same program at an independent set of data points is required. Often numerical simulations can be built on such a computational scheme.

With this approach, the gap between CPU and GPU, with regard to *the technical capabilities*, gradually starts closing and a comprehensive set of traditional CPU-targeted tasks can be executed on the GPU.

For many visualizations and scientific computations, the use of *parallel architectures* (such as the GPU) is essential, because nowadays parallel architectures outperform the computational power of a *single-threaded* CPU by several magnitudes. In the future, one might expect that these differences will become even more crucial because the scale of single-threaded computations is stagnating due to power density issues (Sutter, 2005), whereas the parallel architecture of GPUs can still benefit from exponential growth in its number of cores, Nickolls and Dally (2010).

A more modern approach for GPGPU programming is to use NVIDIA CUDA C (Nvidia, 2017) or OpenCL C/C++ (Munshi et al., 2011) instead of graphics shaders. CUDA is a framework provided by NVIDIA that bypasses the use of standard shaders to enable general-purpose parallel computation on the GPU by directly accessing the GPU. However, it is limited to NVIDIA hardware, and CUDA is not available on browsers. The Khronos Group introduced OpenCL as an alternative that targets a broader range of hardware for GPGPU computation.

Both of the techniques provide programmable kernels that run close to the hardware for the computations. Nevertheless, CUDA and OpenCL are not available on every platform. In particular, these techniques are not available within browsers. The endeavors to make OpenCL available on the browser through WebCL have declined: for instance, Mozilla (announced by Vukicevic (2014)) decided not to implement WebCL in favor of WebGL compute shaders.

This chapter is limited to *shader*-based approaches, although many results could be easily transferred for target architectures such as CUDA or OpenGL. The motivation for our particular focus on the shader-based approach is *WebGL* (Marrin, 2011), which can run shaders on recent web browsers without additional plugins and has extensive *cross-platform compatibility*. WebGL, in particular, can be used for demonstration purposes within a webpage and it is suitable for many modern devices as a target platform (Evans et al., 2014).

7.1.2. The gap in programming concepts between CPU and GPU

Even tough tasks that were initially done on the CPU can now be accelerated on the GPU, there is still a significant gap in *programming concepts* between CPU and GPU. This clear distinction may be desired if the aim is to create most-efficient software that exploits the maximum possible computational power tailored for the available hardware architecture. The standard application programming interfaces (APIs) such as OpenGL, DirectX (and also OpenCL C/C++ and CUDA C), are made for programmers who want to make very conscious hardware decisions. However, if a programmer wants to benefit from the advantages of the GPU within an abstract scripting language (for instance for rapid software prototyping), then the development efforts should also be minimized. For scientific computing, the time-saving advantages of the GPUs are often not used because the gain in performance time often does not justify the additional development effort and time (Klöckner et al., 2012).

In our opinion, there are several difficulties in integrating shader programs (and also kernels for CUDA or OpenCL) within other programs that keep programmers from using the GPU:

Separation. The shaders are separated from the main program. In many APIs, a shader program has to be provided through an external file or a string containing the shader source, and thereon the source is compiled with a special compiler for the GPU. This separation in code increases the complexity of a program.

Lines of code. A lot of technical boilerplate code (i.e. code that is duplicated verbatim for various applications with only slight modifications) has to be written to create even simple applications.

Own language. Shaders are written in a programming language that has been designed for graphics purposes and will be compiled for the shaders for the GPU. Even to map other non-graphical computations to the GPU, the programmer has to become acquainted with this programming language. Furthermore, there is a semantic discrepancy when shaders that are written in a compiled language are used within a scripting language. A scripting language is, in contrast, designed to be interpreted at running time.

Platform-dependence. The use of shaders poses a limitation in the number of platforms on which the program will run. This often demands additional programming to make the software run on a sufficiently large set of different hardware.

Standardized APIs like OpenGL have overcome the last limitation. With OpenGL, shader code can be compiled for GPUs of different vendors. Often, these shaders are compiled at run-time on a target machine. WebGL (Marrin, 2011) utilizes this mechanism to use OpenGL within different browsers.

To our knowledge, for most scripting languages, there are only APIs available that require the programmer to write the shader programs in a particular low-level language that has been designed for the GPU. (For example, if OpenGL is accessed within JavaScript through WebGL, the programmer still has to provide shader codes written in the OpenGL Shading Language (*GLSL*)).

7.1.3. Our objectives for a high-level language with GPU support

The project aims to develop a concept that makes it easier to embed GPU shaders in another high-level programming language (referred to as *host language* in the following). Our main aims are:

Smooth integration. The code for the GPU shaders should be smoothly integrated within the host language with only a minimum of additional language constructs. Ideally, the programmer should not need to indicate (or even be aware of) whether he or she wants to use the graphics card. In particular, the programmer, who does not program close to the shader, does not have to indicate the splitting point between CPU and GPU code. With this approach, a large class of user-defined functions should be usable both on CPU and GPU, and the amount of written code would be drastically reduced.

Efficiency. At the same time, the performance at running time should remain comparable with applications that traditionally use graphics shaders.

Portability & Stability. Ideally, the programs should run on many platforms. For a visualization framework with a broad outreach, we propose using WebGL.

Versatile applications. Our concept is intended to provide a tool for both real-time rendering of images and basic GPGPU programming.

Compilation of untyped language. The scripting language will be transcompiled to a shader language at running time. (The shader language can then be compiled to the vendor-specific binary code for the CPU.) As the host language is assumed to be a dynamically typed scripting language, the programmer should not decide upon GPU-specific types to be chosen. In our approach, the types are *inferred automatically*. The types should be as weak as possible in order to make the program run fast. In fact, this requirement poses strong formal constraints on the software architecture of the transpiler. Major parts of this chapter are dedicated to this issue.

Mathematically-oriented user. The language should fit the requirements of a mathematically-oriented user. For example, variables for numbers should be able to store complex-valued numbers, if needed. Vectors of arbitrary length, basic linear algebra operations for matrices and vectors should be part of the language as well.

In Section 7.1.4, we present an overview of related work, then in Section 7.2, we introduce the primary process of the transcompilation. In Section 7.3, we provide a brief outline of our sample implementation *CindyGL* and a set of examples. *CindyGL* is a JavaScript-based implementation of the concepts introduced in this paper. *CindyGL* can transcompile the scripting language *CindyScript* to *GLSL* and thus can be used to create widely portable visualizations and GPGPU programs that can be run within the browser.

7.1.4. Related work

In dynamic geometry software, often an expression is evaluated for many data points. An aim addressed by approach is the challenge to render images, where each pixel on the screen gets assigned a color by a given function. Liste (2014) suggests a technique to render curves in Geogebra via a so-called "sweeping-line" approach on the CPU, which turns out to be very slow even on recent hardware. Also the dynamic geometry software Cinderella 2 includes a `colorplot`-command to produce similar plots (Richter-Gebert and Kortenkamp, 2012). However, all these approaches are very slow due to computations on the CPU, and complex images cannot be rendered in real-time. To our best knowledge, no dynamic geometry software can utilize the GPU for such tasks that would require a seamless embedding of GPU programming within a high-level environment.

However, some (non-DGS) projects aim for seamless integration of CUDA or OpenCL within another scripting programming language. *Copperhead* (Catanzaro et al., 2011) is probably the most remarkable related project. Copperhead can translate a subset of the scripting language Python to CUDA at running time. The types are modeled in a minimal Hindley-Milner type system and inferred for the GPU. Hence Copperhead enables GPGPU programming within a high-level host language, without requiring the programmer to provide separate code in a low-level language designed for the GPU.

Another approach in integrating GPU accelerated code in another programming language is the introduction of a new datatype for (large) arrays that is permanently stored on the GPU. For example, *Accelerator for C#* (Tarditi et al., 2006) implements a new **`parallel array`** datatype which one can use for element-wise operations, reductions, affine transformations and linear algebra on the GPU. In a similar spirit, in MATLAB, **`gpuArray`** objects were introduced with release R2010b. The **`gpuArray`** objects represent matrices that are stored on the GPU, and several operators are overloaded for objects of the **`gpuArray`** class. For a more extensive description on GPU programming in MATLAB, we refer to Reese and Zaranek (2012). Furthermore, a subset of MATLAB code can be executed on the GPU through the **`arrayfun`** function if it is applied to a **`gpuArray`**. However, the function within **`arrayfun`** cannot access variables from the workspace. Matlab utilizes CUDA for its GPU computations. In addition, Klöckner et al. (2012) introduce a class named **gpuarray** in *PyCUDA* that supports abstractions for many component-wise operations. More complicated functions require the programming of kernels in CUDA C.

Nevertheless, a substantial difference compared to our project is that CUDA is used to access the GPU. In contrast, we use WebGL and focus more on the mathematically-oriented user. Because of our choice of WebGL, running GPGPU code might be slower than approaches that are based on the CUDA architecture, which has been designed for the GPGPU purpose, but we win in terms of compatibility on every machine that provides a browser with WebGL support and can use a pipeline that has been designed for real-time visualizations.

7.2. Concept

Before going into the details of the symbolic process of type inference, we will describe the general setup that clarifies the roles between author, scripting language, interpreter/compiler, GPU and CPU usage.

We recommend generating the GPU code at run-time, as has been introduced by Klöckner et al. (2012). The generation of GPU code at run-time increases the portability of the software, and the scheme goes hand in hand with the design idea of a typical scripting language. Furthermore, this provides the possibility for the self-adaption of a program according to the given input data.

The aim is to enable efficient GPU computations without posing difficulties in the development. Therefore most of the preparation for running binary code on the GPU will be done on the machine of the user. Since the results of the GPU compilation will be cached, the penalties in running time will mostly only occur during the first execution of code that is suitable for parallelization.

We propose the following concept, which is illustrated in Fig. 7.1: The author of the content will directly program his or her idea through a single scripting language, regardless whether he or she is aiming for a GPU or CPU computation. When the end user executes the scripting code, the interpreter will parse the scripting code by standard techniques as described by Levine (2009) to obtain an abstract syntax tree (referred to as AST in the following). Instead of instantaneously executing the entire AST, the interpreter computes a split of the AST such that those parts that are suitable for parallelization are separated from those parts that are more appropriate for execution on the CPU. On its first execution, the interpreter compiles the code suitable for the GPU to a shading language of the GPU and then uses the host's GPU compiler to compile

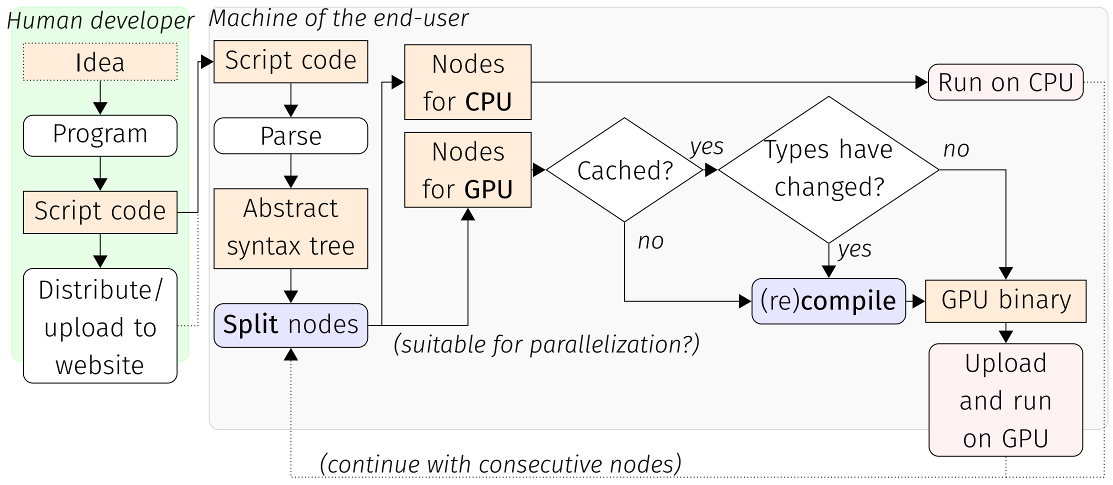


Figure 7.1.: Our proposed scheme, which extends the concept from Klöckner et al. (2012), for a seamless embedding of shader code within the environment of a scripting language. For simplicity, data transfers between GPU and CPU are omitted in this figure.

this shading code to GPU binary. For the subsequent calls of the same code, the compilation is skipped. Only if the types of the contained variables have changed because of different parameters (e.g. if a real-valued variable becomes complex valued) a recompilation is enforced. The GPU binary is uploaded to the GPU and executed on the GPU.

More details about the splitting, type detection, and transcompilation follow in Sections 7.2.1, 7.2.2 and 7.2.3, respectively. The lazy synchronization of data between GPU and CPU will be addressed in Section 7.2.4.

7.2.1. Detection of parts for parallelization and splitting the code

We assume that the host scripting language has a set of specific operations that are suitable for parallel computations on the GPU without any modifications. These can be instructions that belong to the *single instruction, multiple data* (SIMD) class, i.e., very similar instructions that are to be executed on a large set of data points. In our abstract definition of a DPM this corresponds to the **compute everywhere**-statement. Also, the `map` operator of a functional language (if it is applied to a function without side effects and a sufficiently large array) corresponds to this scheme. The `map` operator applies a given function to each element of an array and returns the results as an array of the same dimensions. Another operation, which was implemented in our sample realization *CindyGL*, is the semantically similar `colorplot` command. The command generates a texture, i.e. a large 2-dimensional array of pixels, where the color of each pixel is computed by a given function taking the pixel coordinate as an input.

Within this function, user-defined functions should also be callable. However, it can not be expected that a transcompilation can work for all functions. For instance, the *GLSL* shading language 1.0 (Simpson and Kessenich, 2009) does not allow any recursion.

An alternative language construct that is also suitable for our concept is the introduction of a special array type that is permanently stored on the GPU. This is familiar from the parallel array in Accelerator for C# (Tarditi et al., 2006), `gpuarray` in PyCUDA (Klöckner et al., 2012), and `gpuArray` in Matlab (Reese and Zaranek, 2012). The point-wise operations on this array are suitable for parallelization.

All these constructs have at least one *running variable* that takes a different value for each call of the function. Within `map`, a function is applied to this variable, or the pixel coordinate within `colorplot` becomes the running variable. Those terms that are invariant on the running variables should be computed only once instead of being massively parallelized. To determine them, we build a directed *dependency graph* $G = (V, E)$ that contains all variables and all expressions that appear within the function as nodes. Without loss of generality, we assume that there is only one running variable. Otherwise, we can assume the running variables are stored together in a single array. Let v_0 denote the running variable. There is a directed edge $(a, b) \in E$ iff the variable/term a depends on b. This might be because a is a variable and is assigned to b or a is an expression that contains b and hence its value is dependent on b (i.e., b is a child of a in the AST). Furthermore, all variables that are modified within a conditional loop are dependent on the condition of the loop (e.g., the boolean variable within an if-clause, or the number of repetitions of a loop).

Now a set $D \subset V$ of terms that depend on the running variable is computed: $D \subset V$ contains precisely those nodes that can be reached in G from $v_0 \in V$ and can be determined by a depth-first search in G. The node $r \in V$ corresponding to the result term either is contained in D or is not contained in D. If $r \notin D$, then the computation is independent on the running variable and thus always takes the same value and can be computed once on the CPU. If $r \in D$, all terms in D will be marked for a transcompilation to the GPU.

All nodes in $v \in V \setminus D$ that have an immediate successor in D (i.e., there is an $a \in D$ such that $(v, a) \in E$) will be marked as uniform variables and form a set U.

Let $\mathcal{E} = D \cup U$ denote the set of relevant expressions. The following transformations will be only applied to $\mathcal{E}$. The values of the terms in U will be computed once on the

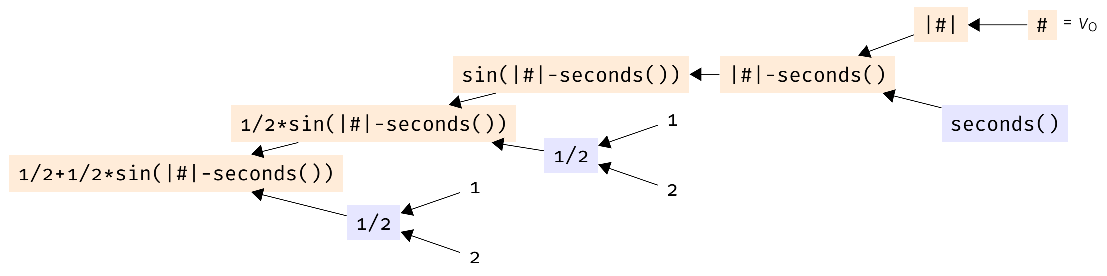


Figure 7.2.: Dependency graph G for expression `1/2+1/2*sin(|#|-seconds())`. Nodes in D are highlighted in orange, nodes in U are highlighted in blue.

CPU and used as input parameter for the GPU program.

This enables us to obtain an almost optimal split of code in CPU and GPU computations.

Example 57. Consider the expression `1/2+1/2*sin(|#|-seconds())` [1]. Let `#` be the running variable v_O. The expression generates a dependency graph G shown in Fig. 7.2. The three terms `seconds()`, and twice `1/2` are the uniform expressions U, i.e. they are independent from v_O = `#` and are computed once on the CPU because they attain their value independently from v_O.

7.2.2. Type detection

A scripting language usually does not have a static type system. A variable can store any object; a function can return various values of different types. This noncommittal typing makes the fast-prototyping programming process easy, but the programs will be less efficient due to run-time type checking. However, for the transcompilation to the GPU variables, terms, arguments and return values of functions should take a static type, since the shader languages, trimmed for efficiency, only support static types.

We impose the following three requirements on the type system:

No type annotations in the host language. The programmer should not have to give type annotations. Omitting the type annotations is essential so that the GPU codes fit seamlessly into the scripting code of the host language. Possible up-casting of types should be done automatically.

No polymorphic types. Because of the restrictions of the shading languages, which should run as efficiently as possible, only monotypes should be allowed. That

[1]This is an expression to generate the colorplot of a centered sinusoidal wave, see Example 62.

means that a general type signature should not be inferred for any user-defined function. Instead, whenever a user-defined function is used, a single signature of monotypes is determined. Functions will be monomorphized.

Minimal typing. The types should be as weak as possible in order to make the program run fast. For example, if the transcompiler can prove that a variable remains to be an integer, then the integer data type on the GPU should be used instead of a floating point number or a complex number.

Thus, the transcompiler has to automatically assign static types to the variables and terms of a dynamically typed language. As is the case for Copperhead, this requires well-typed programs. A counter-example of a well-typed program is `if(booleanexp, 12, [0])`, which returns `12` if `booleanexp` is `True` and `[0]` otherwise. This program will cause an error during the transcompilation since the types of the if and else branch cannot be unified to a single type.

We define the set of all types T recursively:

- $\bot, \top,$ `boolean`, `int`, `real`, `complex` $\in T$
- $\forall \tau \in T \setminus \{\bot, \top\}, \forall n \in \mathbb{N} : \mathrm{list}(n, \tau) \in T$

$\bot$ will correspond to the unset type and $\top$ to the error type, which will be used if no type could be determined. All other types can be modeled in a shader language. For instance, the type list(4, list(4, `real`)) corresponds to a 4×4 matrix and can be modeled in *GLSL* through the `mat4` type. Matrices of higher dimensions can be modeled with corresponding `structs` in *GLSL*. For dynamic geometry software, it is also suitable to add types such as `point`, `line`, and `circle`.

Furthermore, we equip T with a reflexive, antisymmetric, and transitive *subtype* relation $\sqsubseteq$ to make $(T, \sqsubseteq)$ a lattice, which is recursively generated as follows:

- $\forall \tau \in T : \bot \sqsubseteq \tau \sqsubseteq \top$
- `boolean` $\sqsubseteq$ `int` $\sqsubseteq$ `real` $\sqsubseteq$ `complex`
- $\forall \sigma, \tau \in T \setminus \{\bot, \top\}, \forall n \in \mathbb{N} : \sigma \sqsubseteq \tau \Rightarrow \mathrm{list}(n, \sigma) \sqsubseteq \mathrm{list}(n, \tau)$

A subtype relation $\sigma \sqsubseteq \tau$ means that every value that can be represented by type σ can also be represented by a type τ. Furthermore, each relation $\sigma \sqsubseteq \tau$ implies that there is an injective inclusion function from the set of values in σ to the set of values in τ. With $\sigma \sqcup \tau$ we denote the least-upper-bound of σ and τ.

Example 58. Typical values for $\sqcup : T \times T \to T$ are:

$$\begin{aligned} \texttt{int} \sqcup \texttt{real} &= \texttt{real} \\ \bot \sqcup \texttt{complex} &= \texttt{complex} \\ \text{list}(5, \texttt{complex}) \sqcup \text{list}(5, \texttt{real}) &= \text{list}(5, \texttt{complex}) \\ \text{list}(2, \texttt{real}) \sqcup \text{list}(3, \texttt{real}) &= \top \end{aligned}$$

Proposition 59. $\sqcup : T \times T \to T$ is computable.

Proof. $\sigma \sqcup \tau$ for $\sigma, \tau \in T$ can be computed recursively. We can handle the cases $\{\sigma,\tau\} \cap \{\bot,\top\} \neq \emptyset$ via $\sigma \sqcup \bot = \sigma$, $\sigma \sqcup \top = \top$ (and the symmetric equations). If $\{\sigma,\tau\} \subset \{\texttt{boolean}, \texttt{int}, \texttt{real}, \texttt{complex}\}$ then $\sigma \sqcup \tau$ is the maximum of σ and τ in the sequence $\texttt{boolean} \sqsubseteq \texttt{int} \sqsubseteq \texttt{real} \sqsubseteq \texttt{complex}$. In all other cases, σ or τ is a list. Wlog. let $\sigma = \text{list}(n, \alpha)$ for some type $\alpha \in T$. The term $\text{list}(n, \alpha) \sqcup \tau$ can be recursively evaluated as follows

$$\text{list}(n,\alpha) \sqcup \tau = \begin{cases} \text{list}(n, \alpha \sqcup \beta) & \text{if } \tau = \text{list}(n,\beta) \\ \top & \text{otherwise} \end{cases}$$

The recursion terminates because α and β contain fewer list terms than σ and τ.

$\square$

A variable that takes values of both type σ and τ should be of type $\mu := \sigma \sqcup \tau$. Then $\sigma, \tau \sqsubseteq \mu$ and μ is minimal in this property.

Our primary aim is to determine the type of all variables and terms in $\mathcal{E}$. That is challenging because the program poses several, possibly circular, dependencies between the types of terms. First, we will model collections of types through product lattices. Then we will elaborate on how to model the dependencies between the different variables and terms as a mathematical condition on the product lattice. In the last step, we will show how a tailored fixed-point algorithm can be used to compute a minimal typing that fulfills the required conditions.

For any $n \in \mathbb{N}$, T^n becomes a lattice by defining the product order $(\sigma_1, \ldots, \sigma_n) \sqsubseteq (\tau_1, \ldots, \tau_n)$ iff $\sigma_i \sqsubseteq \tau_i$ for every $i \in \{1, \ldots, n\}$. It can be easily seen that the height of the lattice T, and therefore the height of the lattice T^n, is finite. In particular they fulfill the *ascending chain condition* (ACC), i.e. every increasing chain eventually becomes stationary.

A function *fun* of arity n (i.e. *fun* takes n arguments) can have multiple *signatures*. In the transcompilation of *fun*, it is suitable to choose the implementation of *fun* that has a signature as weak as possible, but as strong as necessary. Therefore, for every n-adic function *fun*, we equip the transcompiler with a function

$$\textsc{minSign}_{fun} : T^n \to T^n \times T$$

that, applied to the types of the provided arguments, returns a signature, which we consider as a tuple of argument types and a return type. If

$$\textsc{minSign}_{fun}(\tau_1, \ldots, \tau_n) = ((\alpha_1, \ldots, \alpha_n), \beta)$$

we demand $(\tau_1, \ldots, \tau_n) \sqsubseteq (\alpha_1, \ldots, \alpha_n)$ and $(\alpha_1, \ldots, \alpha_n) \to \beta$ is the *minimal* signature of an available applicable GPU implementation of *fun* for arguments of type $(\tau_1, \ldots, \tau_n)$, i.e. there is an implementation of *fun* that takes arguments of types $(\alpha_1, \ldots, \alpha_n)$ and returns a value of type β. By *minimal*, we mean that there is no other implementation with a signature $(\alpha_1^*, \ldots, \alpha_n^*) \to \beta^*$ such that $(\tau_1, \ldots, \tau_n) \sqsubseteq (\alpha_1^*, \ldots, \alpha_n^*) \sqsubset (\alpha_1, \ldots, \alpha_n)$. The map $\tau \mapsto \textsc{minSign}_{fun}(\tau)_2$ is monotone.

Example 60. Typical values for $\textsc{minSign}$ for the addition function $+$ and the square root $\sqrt{\cdot}$ can be

$$\begin{aligned}
\textsc{minSign}_{+}(\texttt{int}, \texttt{int}) &= ((\texttt{int}, \texttt{int}), \texttt{int}) \\
\textsc{minSign}_{+}(\texttt{complex}, \texttt{int}) &= ((\texttt{complex}, \texttt{complex}), \texttt{complex}) \\
\textsc{minSign}_{\sqrt{\cdot}}(\texttt{int}) &= ((\texttt{real}), \texttt{complex})
\end{aligned}$$

For the first two values we assume that GPU implementations for $+$ for the data types `int`, `real`, and `complex` exist. This is mathematically motivated by the different domains for the functions $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $+ : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ and $\sqrt{\cdot} : \mathbb{R} \to \mathbb{C}$. Whenever those functions are applied to values of a domain that can be embedded within another domain (i.e. $\mathbb{Z} \hookrightarrow \mathbb{R}$), then the generalized function on a wider domain can be used. This means that the evaluation of $+$ on two different arguments, the implementation for the lowest type that is a super-type of both its arguments should be chosen. If one of the two arguments is a real subtype of the other, it will automatically be upcasted before the evaluation.

The third example describes the (complex) square root. It is reasonable to provide an implementation that computes the complex square root of real numbers. If one wants to calculate the square root of an `int`, then it is suitable to choose the same function. It would also be possible to introduce additional datatypes for positive numbers, which can be used to ensure that the square root remains real.

Now, our aim is to compute a *minimal* type assignment $\Gamma \in T^{\mathcal{E}}$ that assigns every relevant expression in $\mathcal{E}$ ($\mathcal{E}$ is defined in Section 7.2.1 as the union of the dependent term/variables and the uniform variables) to a type of T such that the assigned types are *compatible* to the program in the following sense:

- $\Gamma_s \sqsubseteq \Gamma_t$ if t is a variable and there is an assignment $t := s$ for a term s.
- $\textsc{minSign}_{fun}(\Gamma_{a_1}, \ldots, \Gamma_{a_n})_2 \sqsubseteq \Gamma_t$ if t is a term that consists of the function *fun* applied to the arguments $a_1, \ldots, a_n$, i.e. $t = fun(a_1, \ldots, a_n)$.
- $\tau_0 \sqsubseteq \Gamma_{v_0}$ where v_0 is the running variable and τ_0 its type (i.e. `int` for scenarios if `map` is applied to an integer list and list(2, `real`) if v_0 is a pixel coordinate).
- $\tau_u \sqsubseteq \Gamma_u$ if $u \in U$, i.e. u is a uniform expression (this includes constant expressions), u will be computed by the CPU, and is of type τ_u.

Whenever a user-defined function with some arguments is called, then a "virtual assignment" for each of the argument variables is added, in order to make it possible to compute specific types of the arguments and establish a type assignment that also involves user-defined functions.

To compute a minimal compatible type assignment $\Gamma \in T^{\mathcal{E}}$, we will utilize the Kleene fixed-point theorem (Stoltenberg-Hansen et al., 1994): Let $\bot \in T^{\mathcal{E}}$ (i.e. $\bot = (\bot, \ldots, \bot)$ – no type is set yet), then we can iteratively apply a monotone function $F : T^{\mathcal{E}} \to T^{\mathcal{E}}$ on $\bot$. Since $T^{\mathcal{E}}$ fulfills the ACC, the ascending chain

$$\bot \sqsubseteq F(\bot) \sqsubseteq F(F(\bot)) \sqsubseteq \ldots$$

eventually becomes stationary at a fixed point of F and according to the Kleene fixed-point theorem, this stationary fixed point is the unique *minimal* fixed point of F.

In order to compute a minimal $\Gamma \in T^{\mathcal{E}}$ that fulfills the *compatibility* requirements listed above, we choose the following monotone $F : T^{\mathcal{E}} \to T^{\mathcal{E}}$:

$$F(\Gamma)_t = \begin{cases} \Gamma_t \sqcup \bigsqcup_{i=1}^{n} \Gamma_{s_i} & \text{if } t \text{ is a variable and} \\ & \text{there are the assignments } t := s_1, \ldots, t := s_n \\ \textsc{minSign}_{fun}(\Gamma_{a_1}, \ldots, \Gamma_{a_n})_2 & \text{if } t = fun(a_1, \ldots, a_n) \\ \Gamma_t \sqcup \tau_0 & \text{if } t = v_0 \text{ is the iteration variable} \\ \Gamma_t \sqcup \tau_u & \text{if } t = u \text{ is a uniform variable/term} \end{cases}$$

The function F is component-wise monotone and hence also monotone in the product lattice. The component-wise monotonicity follows either by using the least upper bound of the input parameter or by the monotonicity of $\textsc{minSign}$.

Any fixed point of F fulfills the *compatibility* requirements listed above for Γ. Hence, we can compute a *minimal* type assignment $\Gamma \in T^{\mathcal{E}}$ that fulfills the *compatibility* requirements by iteratively applying F to $\bot$ until it becomes stationary and we take this fixed point of F as Γ. This corresponds to a start with unset types and updating the types to the lowest suitable type whenever there is a place where the currently set types are not sufficiently expressible.

Example 61. Consider the example code

```
a = -2;
b = sqrt(a);
a = b + 1;
```

It contains the terms $\mathcal{E} = \{$`a`, `-2`, `b`, `sqrt(a)`, `b+1`, `1`$\}$. Using the presented fixed-point algorithm, we will determine the types of all the terms. We will start with a type assignment $\bot \in T^{\mathcal{E}}$ and iteratively apply F, which uses MINSIGN from Example 60:

	$\bot$	$F(\bot)$	$F^2(\bot)$	$F^3(\bot)$	$F^4(\bot)$	$F^5(\bot)$	$F^6(\bot)$	$F^7(\bot)$
`a`	$\bot$	$\bot$	`int`	`int`	`int`	`int`	`complex`	`complex`
`b`	$\bot$	$\bot$	$\bot$	$\bot$	`complex`	`complex`	`complex`	`complex`
`sqrt(a)`	$\bot$	$\bot$	$\bot$	`complex`	`complex`	`complex`	`complex`	`complex`
`b+1`	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	`complex`	`complex`	`complex`
`-2`	$\bot$	`int`	`int`	`int`	`int`	`int`	`int`	`int`
`1`	$\bot$	`int`	`int`	`int`	`int`	`int`	`int`	`int`

Since $F^6(\bot) = F^7(\bot)$, this value gives a valid and minimal typing for the code.

7.2.3. Transcompilation

Once a typing $\Gamma \in T^{\mathcal{E}}$ is computed, the transcompilation to GPU shader code is straightforward. If the program was not well typed, i.e. there is an expression $t \in \mathcal{E}$ with $\Gamma_t = \top$, then the computations will be evaluated on the CPU. Otherwise (and we hope in most cases), we apply the following scheme:

An expression $t = fun(a_1, \dots, a_n) \in D$ can be converted to shader code by carrying out the following three steps:

(1) First, translate the arguments $a_1, \dots, a_n$ recursively to shader code.

(2) Then up-cast each of the translated arguments a_i, that has type Γ_{a_i}, to the type

$$\left(\text{MINSIGN}_{fun}(\Gamma_{a_1}, \dots, \Gamma_{a_n})_1\right)_i$$

employing the subtype-embedding function. Note that by definition $(\Gamma_{a_1}, \dots, \Gamma_{a_n}) \sqsubseteq \text{MINSIGN}_{fun}(\Gamma_{a_1}, \dots, \Gamma_{a_n})_1$.

(3) Lastly, embed the implementation in the shader language of *fun* that corresponds to the signature $\text{MINSIGN}_{fun}(\Gamma_{a_1}, \ldots, \Gamma_{a_n})$ into the header of the generated shader code (provided it has not already been done) and return the application of this function to the up-casted translated arguments as translated shader code for $fun(a_1, \ldots, a_n)$.

For each uniform variable/term, a new unique variable name will be generated and used in the program.

After this translation to shader code, the driver-dependent GPU compiler will be used to compile and link the generated shader code.

According to Marrin (2011), WebGL 1.0 allows only a fixed number of loop repetitions and fixed-length arrays. However, often the number of loop repetitions or the length of a list is based on input parameters. In this case of a changing number of repetitions, a re-compilation becomes necessary after the corresponding variables have changed their value. If the constant values are encoded as separate types, this re-compilation can be enforced without significant effort. A change of these constants would correspond to a type change, and, according to our scheme introduced in Section 7.2, this would trigger a re-compilation.

7.2.4. Lazy storage of data

A suitable way to store the outcomes of shader programs is to write to textures. Reading this data is possible by texture lookups. According to Gregg and Hazelwood (2011), the memory transfer between CPU and GPU has severe effects on the running time of an application and thus has to be minimized. Therefore, we store GPU-generated data exclusively on the GPU as long as necessary, and the data is only visible for shader programs that run on the GPU. Only if there are read accesses by the part of the code that runs on the CPU, data will be transferred.

Within a single shader program, the preference is often to write data to a target (texture) that is also used for reading. That is problematic for many GPU-APIs. For instance, the WebGL API specifies that the occurrence of operations that both write to and read from the same texture, creating a feedback loop, will generate an error, see (Marrin, 2011, 6.26). For these feedback loops, a ping-pong approach can be used: If both read and write accesses on a texture object are detected, the texture will be stored twice: one texture for reading and another target texture for writing. After the execution of a shading program that writes to the corresponding texture, the two textures will be swapped. For following reading accesses, data will be read from the generated texture.

7.3. Example implementation: *CindyGL*

In (Montag and Richter-Gebert, 2016), we demonstrated an implementation of our proposed concepts. We developed a plug-in called *CindyGL* for *CindyJS*. *CindyJS* (von Gagern, Kortenkamp, Richter-Gebert, and Strobel, 2016) aims to be an open source web-compatible porting of the dynamic geometry software *Cinderella* (Richter-Gebert and Kortenkamp, 2000) [2]. *CindyGL* can translate the *Cinderella* inherent untyped scripting language *CindyScript* to *GLSL* and it enables a smooth integration of dynamic geometry, CPU, and GPU programming.

As *CindyJS* runs in a web environment, *CindyGL* can utilize WebGL to execute the compiled *GLSL*-Code on the GPU. This capability leads to easy portability because WebGL-capable browsers are widespread. By September 2019, almost all[3] of the browsers used on desktops, smartphones and tablets supported WebGL 1. Visualizations that use *CindyGL* can be quickly distributed because no additional software usually has to be installed to access WebGL-based contents. Also, the visualizations will be comfortably accessible on new devices that provide a WebGL-capable browser. If no acceleration is available through the GPU, as a fallback solution, the smooth language integration can be utilized to execute all the code on the CPU.

The comparable simplicity to program GPU accelerated applications through *CindyGL* has led to several implementations. Examples of visualizations that have been generated through *CindyGL* can be found online[4] and tested on almost every browser. In the following sections, we will present a short introduction to *CindyGL*[5] and briefly demonstrate some *CindyGL*-applications.

7.3.1. Usage of *CindyGL*

CindyGL implements a command called `colorplot`. The `colorplot` command assigns a color to each pixel of the screen or a given area according to a given function. The given function is usually dependent on the pixel coordinate `#`. `#` is a 2-component vector. Alternatively, if another free variable is detected, it can also become the running variable. If both the variables `x` and `y` are free, then `colorplot` will interpret `x` and `y` as the coordinates of a pixel. If the free variable `z` is used, then the coordinate will be interpreted as a complex number with `z = x+i*y`.

If the function within the `colorplot` statement attains real numbers as values, then

[2]The entire project is open source and available at `https://github.com/CindyJS/CindyJS`
[3]According to `https://webglstats.com/`, 98% in September 2019
[4]For example, in our web-gallery `https://cindyjs.org/gallery/cindygl/`
[5]A more elaborate tutorial is available online at `https://cindyjs.org/docs/cindygltutorial/`

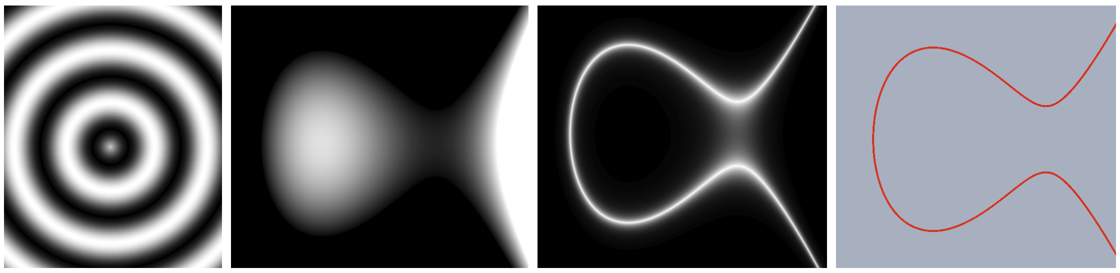


Figure 7.3.: (a) Circular sinusoidal wave generated by the `colorplot` command. (b), (c), (d): Visualizations of the elliptic curve $y^2 = x^3 - x + \frac{1}{2}$ as zero set of $f(x,y) = x^3 - x + \frac{1}{2} - y^2$: (b) `colorplot(f(P))` making the values of $f(P) \in [0,1]$ distinguishable as grayscale value (values outside the range are either black or white), (c) `colorplot(exp(-10*|f(P)|))` highlighting the zero set and (d) by an approach that detects points close to the curve based on the intermediate value theorem.

its color will be interpreted as a grayscale value where 0 is black, and 1 is white. If the result is a three-component vector, then the vector will be interpreted as an RGB value. If there is also a fourth component, the value of this component will be interpreted as an alpha value (transparency).

Example 62. The following *CindyScript* code is an example to render a circular sinusoidal wave through a `colorplot`:

```
colorplot( // assigns to each pixel with coordinate # a color
 1/2+1/2*sin(|#|-seconds()) // an animated centered sinusidial wave
);         // seconds() is the current time in seconds.
```

Once *CindyJS* executes the code, a circular wave as in Fig. 7.3a becomes visible. By executing the code many times within a second, an animation is created (see Applet ▷7). Since the computation is accelerated on the GPU, real-time rendering is possible on almost every modern device.

CindyGL, as an implementation of the concepts described in Section 7.2, compiles the suitable parts of the code which is passed to `colorplot` to *GLSL* and then to WebGL-binary the first time the command is called on a given expression. For consecutive calls, the compiled program is re-used as long as the occurring types have not changed. If the types have changed in the meantime, a recompilation is forced.

In the following sections, we will present a selection of various use case scenarios of *CindyGL*.

7.3.2. Implicit curves and sets of locus within dynamic geometry software

We assume that we are in the Euclidean plane and $f : \mathbb{R}^2 \to \mathbb{R}$ is a smooth function. Instead of a plot of f, often the variety $V(f) := f^{-1}(\{0\})$ is of special interest. In this section, we assume that the gradient of f does not vanish on $V(f)$, which makes $V(f)$ (locally) a curve by the implicit function theorem. The `colorplot` command gives a very simple pipeline to visualize these curves. It is still very efficient compared to conventional approaches (for no known parametrizations) because the GPU performs the computations in parallel.

An example of implicit curves are elliptic curves. They can be defined by the equation

$$y^2 = x^3 + ax + b$$

where a and b are real numbers such that $\Delta = -16(4a^3 + 27b^2) \neq 0$.

We define an implicit f as follows

```
f(P) := (
  x = P.x; y = P.y;
  x^3 + a*x + b - y^2 // last line: return value
);
```

It is clear that $f(P) = 0$ iff P is in the elliptic curve. The evaluation of `colorplot(f(P))` renders an image as in Fig. 7.3b (P is a free variable and therefore will be detected as a varying pixel coordinate). The elliptic curve is located where black just becomes gray. How can we visualize the points P in the plane such that $f(P)$ becomes zero more apparently? In general, running a test to determine whether $f(P)$ becomes zero for a finite set of pixel coordinates P obtained by rasterization is problematic because of numeric issues and the fact that it is improbable that P will hit a zero of f, even if the corresponding (rectangular) pixel contains zeros of f because if f is a non-constant algebraic function, the zero set has Lebesgue measure 0. Liste (2014), who utilizes the "sweeping-line" GeoGebra on the CPU, suggests a rendering approach, which can be modeled in CindyGL via the command `colorplot(exp(-10*|f(P)|))` (see Fig. 7.3c). All of Liste's applications can be easily transferred to the modern GPU-based `colorplot` via *CindyGL* and work in real-time on this architecture.

A straightforward approach to determine a subset of pixels that contain a part of the implicit curve is the evaluation of the values of f at each corner of every pixel. If these values can be separated by zero, then according to the intermediate value theorem, the curve must "enter" the pixel, and the pixel can be marked as one that contains the curve. This approach can be implemented in CindyGL via

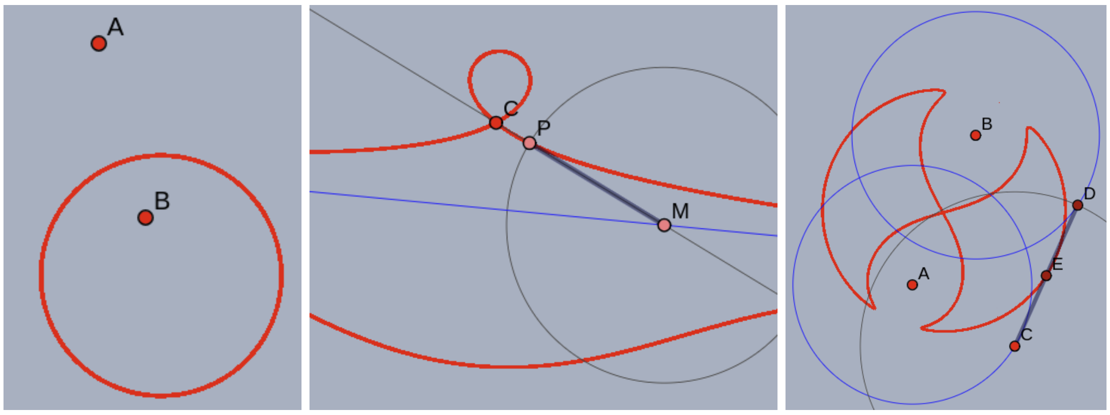


Figure 7.4.: (red) Loci computed on the GPU via the `colorplot` command: (a) All points P such that $\frac{|P-A|}{|P-B|} = 2$ (b) The conchoid of Nicomedes, i.e. the set of all points P that are obtained by moving M in the construction. The second branch of the conchoid can be reached if M is moved through infinity (c) Watt's curve, which describes the set of all points E such that E is the midpoint of C and D, which are two points on distinct circles and have a constant distance $|C-D|$.

```
tinysquare = [[-1,-1],[-1,1],[1,-1],[1,1]]/100; // the corners of a pixel
colorplot(
  // evaluate f at the corners of pixel with center P
  values = apply(tinysquare, delta, f(P+delta));
  if(min(values) <= 0 & 0 <= max(values),
     [1,0,0,1], // the signs swap => plot red with full alpha
     [0,0,0,0]  // all signs are same => plot nothing (transparent)
  );
);
```

If the gradient of f does not vanish at $V(f)$ (for the elliptic curves this is guaranteed by $\Delta \neq 0$) and if the curve has a sufficiently low curvature, then all pixels containing a substantial part of the curve are detected by this method, and a meaningful image can be acquired. Replacing `tinysquare` with a bigger n-gon, provides a straightforward approach to render the curve less precise, but bolder (see Fig. 7.3d).

These implicit curves are of particular interest in dynamic geometry software if they describe the locus of points by geometric terms. As Botana and Abánades (2014) point out, the computation of loci can be a tool for *computer-aided discovery* of mathematical properties. Hölzl (2001) argues that visually accurate sets of locus even without a direct proof can do excellent service in finding evidence: A proof of a hardly doubted property made visible through a locus set can be postponed until a suitable context has been found where it can be embedded.

As a very simple example, given two distinct points A and B in the Euclidean plane and a ratio $r \in \mathbb{R}_{>0}$, a user of DGS could be interested in the set of all points P such that the ratio $\frac{|P-A|}{|P-B|}$ takes the value r. The set of those points can be visualized by the approach above by exchanging f with

```
f(P) := |P-A|/|P-B|-r;
```

The rendered curve, which is the zero set of f, are all the points fulfilling $\frac{|P-A|}{|P-B|} = r$ (Fig. 7.4a shows an image for $r = 2$). When rendering this curve, it instantly becomes visible that those lines of an equal ratio of distances are circles, or if $r = 1$, the perpendicular bisector (line) of A and B. Certainly, this observation of an image does not give a formal proof. However, when this property is needed, it calls to find a proof of such properties.

A more complex locus is the conchoid of Nicomedes. Let a be a line, and C be a point. Furthermore, let C_0 be a circle with its center M on a and a fixed radius. The conchoid is the set of all points P, that can be obtained as the intersection of C_0 and the line connecting M and C if M is moved along a. This condition can also be written as an implicit formula for the point P by building a backward construction of M based on a point P. The point M can be constructed as the intersection of a and the line connecting C and P. P is contained in one of the two branches of the conchoid if P lies on a circle with M as the midpoint and the radius C_0. The geometric construction and the check give rise to the following definition of f, that in a sense reverses the construction sequence:

```
f(P) := (
  l = join(C, P); // the line connecting C and P
  M = meet(l, a); // the intersection of l and a
  |P.xy-M.xy|-C0.radius // return value; =0 ⇔ P is in the locus
);
```

The zero set of f, which is the locus set of the conchoid, can be rendered by the approach above (omitting potential singular points). The results can be seen in Fig. 7.4b or in Applet ▷8. If the geometric construction, which is built within the DGS, is modified, then the GPU computes the new locus in real-time. Note that also types for geometric primitives, such as points and lines, and basic geometric operations have been implemented as a data-type in CindyGL. This enables the transcompilation of such functions f to GPU code and thus enables a stronger interplay of dynamic geometry software and GPU computations.

We were also able to compute the locus of Watt's curve on the GPU (see Fig. 7.4c and Applet ▷9) utilizing a similar geometric "backward construction". The construction involves the intersection of circles and the reflections around a point, which again has been transcompiled to the GPU by our approach and the images of the locus are rendered in real-time. The generalization and mechanical application of this technique of using a backward construction to yield an implicit formula could lead to further research.

7.3.3. Feedback loops and GPGPU applications in *CindyGL*

A strength of `colorplot` is the ability to plot to a texture besides drawing on the screen. This rendered texture can be read in consecutive calls of `colorplot` with the `imagergb` command. The possibility to read and write texture data enables the creation of feedback loops on the GPU and opens the door for many GPGPU computations.

If a user reads and writes to the same texture with some deformations in between, a video feedback loop can be simulated by pointing a camera at a screen that displays the image that the camera records. This technique presents an interesting approach to rendering fractals:

Example 63. Fractals that utilize an escape time algorithm can be rendered through a feedback loop system via `colorplot`. For instance, the filled Julia set for a function $z \mapsto z^2+c$ can be approximated by running the following *CindyScript* source code several times (see Applet ▷10):

```
colorplot("julia", // plot to texture "julia":
 if(|z|<2,         // if |z| < 2, take the color from texture
  imagergb("julia", z^2+c) // "julia" at position z^2 + c
   + (0.01, 0.02, 0.03),   // and make it slightly brighter
  (0, 0, 0)        // otherwise: display black.
 )
);
```

The color of a pixel becomes brighter if it takes more iterations to leave the escape radius 2 from the coordinate of the pixel. After about 50 iterations the fractal in Fig.7.5a can be seen.

The use of *CindyGL* is not limited to graphical visualizations. Computationally demanding numerical schemes are often good candidates for the GPU as well. Data can be stored on textures, and parallelized computations on the data can be triggered by executing a `colorplot` that reads from these textures via `imagergb`. As an example for GPGPU programming, we used *CindyGL* to simulate the interactions of n bodies (see Fig. 7.5b or Applet ▷11). The maximum number of simulated particles that allows

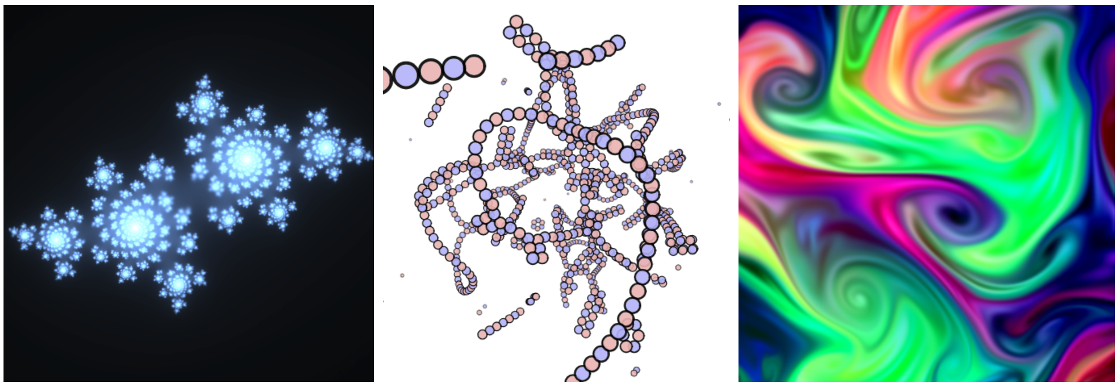


Figure 7.5.: (a) The Julia set generated through a feedback loop approach (b) Numeric simulation of the n-body problem with 500 particles with positive unit charge and 500 with negative unit charge. All the $n^2 = 10^6$ interactions (Lennard-Jones potential) are computed and provide a real-time simulation. (c) Numeric simulation of the Navier-Stokes equations

a real-time visualization increased drastically compared with a corresponding CPU implementation. We also simulated the behavior of a fluid by approximating the solution of the Navier-Stokes equations on a GPU (see Fig. 7.5c or Applet ▷12). The total number of lines of code required to implement this simulation applet with *CindyGL* dropped to about one fourth as compared to an equivalent plain WebGL implementation.

7.3.4. Educational value

Kaneko (2017) argues that the dynamic geometry software on personal computers is often not used in classrooms because of the technical obstacles. However, providing the teaching content on modern touch devices such as iPads and tablets can overcome this obstacle. Aside from PCs, both *CindyJS* and *CindyGL* are suitable for these modern devices because they only depend on a portable plugin-less web technology. Kaneko showed that using this technology has a positive influence on learning.

So, for the teachers, the use of *CindyGL* provides a suitable technique to provide GPU-accelerated content. Using *CindyGL*, the instructors can distribute the interactive content by sharing a single HTML file.

For the students, the task to generate visualization with this framework can be very fruitful as well. Different learning content from mathematics, computer science, physics and other fields can be connected in such tasks. Because of the seamless integration of GPU code in a familiar scripting environment of a DGS, the creation process does not

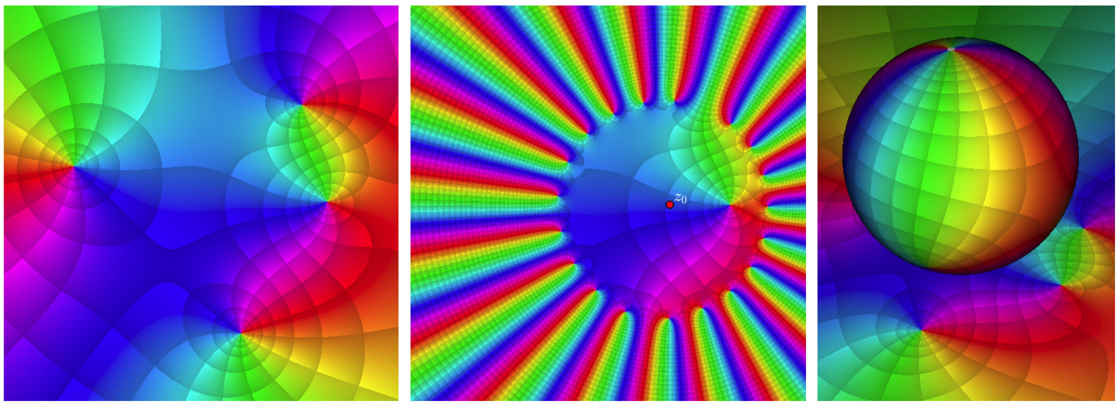


Figure 7.6.: *CindyGL* rendered visualizations of the function $f : \mathbb{C} \to \mathbb{C}, z \mapsto \frac{z-1}{z^3+1-i}$: (a) Phase portrait of f, (b) truncated Taylor expansion of f at $z_0 = \frac{1}{2}$, (c) stereographic projection of f on the Riemann sphere.

require of users to know or learn another programming language. *CindyGL* eases the programming of shaders. Non-experts can use it to visualize mathematical concepts without having to learn about shader programming.

Advanced tasks were assigned to two university students as part of their bachelor's theses, with *CindyGL* provided as a tool. Both students started experimenting with *CindyGL*, drew their mathematical conclusions, and, furthermore, created interactive GPU-based teaching material without previous knowledge of shader programming. In his bachelor thesis, Konnerth (2017) simulated several concepts of complex analysis with phase portraits of partial sums of the Taylor series (see Fig. 7.6b) and a raycasted Riemann-sphere (see Fig. 7.6c) using CindyGL. In another bachelor's thesis, the rolling shutter effect was simulated on the GPU, based on CindyGL.

A seminar aiming for GPU-based mathematical visualizations for a group of mixed Bachelor- and Master-students in Mathematics without previous experience in graphics card programming has been conducted as well. After two sessions of 90 minutes of introduction into *CindyJS* and *CindyGL*, the students were able to develop within the consecutive six sessions several elaborate visualizations that used these tools. A raycaster for three-dimensional Apollonian fractals, an illumination simulation, a fluid simulation, a visualization tool for hyperbolic geometry, and a volumetric renderer that has been used to visualize a variant of Conway's Game of Life and a Runge-Kutta based three-dimensional simulation of the reaction-diffusion equation have been developed. The interactive widgets developed by the participants are available online[6].

[6] https://geo.ma.tum.de/en/teaching/visualization-gpu.html

Part III.

Application: DPMs for Visualizations

Chapter 8.

Visualizations of implicit surfaces

Although mathematics is a discipline in which it is desirable to achieve results by formal deductions, the illustration of mathematical objects is also beneficial. The resulting visualizations often help in understanding and serves for better understanding. Typical illustrations can be images, animations, or shapes in three-dimensional space. Some visualizations originating from mathematics may also have a special aesthetic value. For instance, *algebraic surfaces* from the field of algebraic geometry are challenging to visualize. In this chapter, we will put a particular focus on zero sets of real polynomials in three variables.

In the second half of the 19th and early 20th centuries, a culture of mathematical model building emerged, that included the creation of algebraic surfaces out of plaster. Alexander von Brill and Felix Klein conducted several seminars at the *Royal Technical University in Munich* on the construction of mathematical models (Fischer, 2017). Many of the models sold by the Brill-Verlag, which was later taken over by Martin Schilling, are replicas of the models that have been originally created in Munich (Schilling, 1903).

Many of these models and replicas still can be found in various mathematical institutes. In 1985, Gerd Fischer produced an aesthetic photographic collection of several of these models (Fischer, 2017) and gave me kind permission to use some of his photographs of the old models of algebraic surfaces in this work. These photos are depicted in Figure 8.1. Can we create images that resemble these photographs in real time directly on the computer?

During the development of CindyGL, it was helpful to have some several benchmark projects that utilized the framework in a non-trivial way. One mathematical demanding benchmark project was the implementation of a raycaster for algebraic surfaces. Three-dimensional mathematical models of algebraic surfaces should be depicted on a two-dimensional screen. CindyGL is aimed to be a powerful, user-friendly tool that makes it possible to implement a real-time renderer for a broad set of implicitly defined

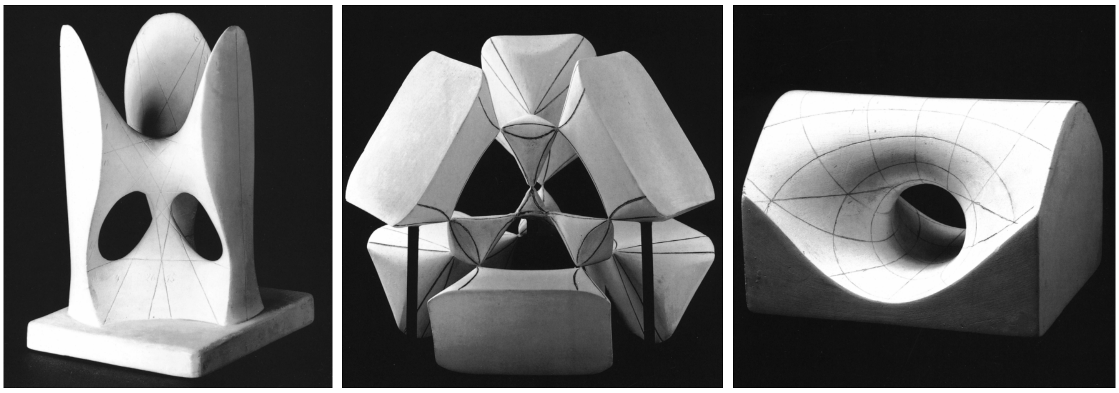


Figure 8.1.: Plaster models of algebraic surfaces: (a) Clebsch diagonal surface, (b) Kummer surface and (c) Parabolic ring cyclide. The photographs were kindly provided by Gerd Fischer.

functions. This benchmark tested the possibility for interactive real-time raycasting and the ability to express and trans-compile a system that involves multiple numerical algorithms and user-defined mathematical formulas. The numerical algorithms were supposed to be specified in a high-level programming language. Later it will turn out that the presented raycaster can be easily converted into a path tracer (Section 8.4).

A further motivation was that this project would give rise to a web-capable raycaster that is based on JavaScript and WebGL. The software *SURFER* of Stussak (2009) was a strong source of inspiration. Even though *SURFER* also utilizes GPU shaders, it is based on Java. Currently, Java-based applets are not supported in several major modern browsers anymore[1]. Another source of inspiration has been the raycaster of Reimers and Seland (2008), which uses the Bernstein basis for better numeric stability.

The CindyGL based raycaster was made available online[2]. Since it accesses WebGL and JavaScript only, the raycaster is supported even on high-end mobile devices without the installation of any plugins or additional software. Such an easily available tool for rendering of algebraic surfaces could be used for mathematics communication. IMAGINARY has used this raycaster to render mathematical love greetings from algebraic equations specifying heart shapes. Through this way, the New York Times also featured the developed raycaster for the Valentine's Day 2019[3].

[1]For instance not in Chrome and Firefox https://www.java.com/en/download/faq/chrome.xml, https://java.com/en/download/faq/firefox_java.xml

[2]The raycaster is available online at https://cindyjs.org/gallery/main/Raytracer/

[3]https://www.nytimes.com/2019/02/14/science/math-algorithm-valentine.html and https://love.imaginary.org/

In the development process of the raycaster, the following opposing objectives had to be weighed out:

Mathematical accuracy The rendered surfaces should be as accurate as possible. However, in general, this is a severe problem and no existing real-time algorithm that correctly renders every surface is known to the author. Both the raycasters for algebraic surfaces of Reimers and Seland (2008) and Stussak (2009) have some problems rendering (exotic) surfaces of a higher degree close to singularities.

Real-time The surfaces should be render-able in real-time on GPU shaders. This means that an approach that relies on Gröbner base computations during its render process is likely not to work due to the double exponential running time. The choice of a suitable robust algorithm to compute the roots of a real-valued polynomial is crucial in this aspect.

GPU suitable Approaches that require a lot of code jumps are less suitable for the GPU than approaches that have a more predictable and well-parallelizable program-flow.

Web suitable For portability, WebGL should be the rendering backend. However, WebGL currently supports only single precision floats (32-bit) whereas desktop applications such as *SURFER* utilize double float precision (64-bit).

Compilation time After a user of the raycaster enters a formula, not much time should last until the shader-code is trans-compiled. Due to the unrolling of parallelized loops, the code complexity is dependent on the desired degree and precision. The major part is not induced by the trans-compilation of CindyGL. The compilation of the GLSL code generated by CindyGL through the vendors graphic card driver takes up most of the time. Within the WebGL API, complexity reduction of the program seems to be the only possibility to influence this compilation time.

Several experiments of different approaches and algorithms were tried out to find a trade-off. The best approach found through these experiments and considerations is presented here.

In the following, let $F : \mathbb{R}^3 \to \mathbb{R}$ be algebraic and irreducible. The aim is to visualize the surface $V(F) = F^{-1}(\{0\})$ by raycasting.

Mathematically, the process of raycasting can be described as follows: For each pixel p of a rasterization screen, there is a ray (an affine linear function) $r_p : \mathbb{R} \to \mathbb{R}^3$ that passes through the pixel on an imaged projection plane and possibly "hits" the surface $V(F)$ at some coordinate $s_p \in R^3$ (possible within a predefined clipping space such a filled sphere or cube). The pixel p can be colored according to s_p. To obtain good visualizations, the normalized gradient $\frac{\Delta F(s_p)}{\|\Delta F(s_p)\|}$ can be taken into consideration; the shading can be done based on the scalar product of a light direction and the normal of the surface. An important observation is that the function $f_p = F \circ r_p : \mathbb{R} \to \mathbb{R}$ is a polynomial and the degree of f_p is bounded by the degree of F. A root t of f_p corresponds to the root $r_p(t)$ of F. Hence, the problem of raycasting algebraic surfaces can essentially be reduced to the problem of locating the real roots of a univariate polynomial for each pixel. This scheme is suitable for massively parallel computations on the GPU.

In pseudo-code a generic raycasting-scheme looks as follows:

Algorithm 20: Gerneral scheme to visualize the surface $F(x, y, z) = 0$ through raycasting

```
  Input: An algebraic function $F : \mathbb{R}^3 \to \mathbb{R}$
  Output: A two dimensional visualization of $V(F) = F^{-1}\{0\}$
1 Compute $dF : \mathbb{R}^3 \to \mathbb{R}^3$ through symbolic computation on the CPU
2 while program is running do
3     Consider user inputs to specify the zoom/rotation/position of the viewer
4     compute $\chi[p], \chi : \textsc{PixelSpace} \to \textsc{ColorSpace}$ everywhere as
5         Consider ray $r_p$ that lies behind the pixel $p$
6         $(a, b) \leftarrow$ an interval such that $[r_p(a), r_p(b)] \subset \mathbb{R}^3$ corresponds to the
            intersection of $r_p$ and the clipping area.
7         $t \leftarrow \textsc{FirstRoot}(F \circ r_p, a, b)$, the smallest root in $(a, b)$ of the polynomial
            $f_p = F \circ r_p$
8         return a color for the intersection point $r_p(t)$ based on $dF(r_p(t))$
9     Display image $\chi$
```

One very simple implementation of this scheme in CindyGL without clipping space is given in the following example:

Example 64. A simple example is the function $F(x, y, z) = x^2+y^2+z^2-1$ with $V(F) = \{(x, y, x) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 = 1\} = S^2$. For simplicity, we assume that a pixel p with coordinates (x, y) is associated with the ray $r_p(t) = (x, y, t)$ and we are looking in positive t-direction. We yield the quadratic polynomial $f_p(t) = F \circ r_p(t) = t^2 + (x^2 + y^2 - 1)$, having the real roots $\pm\sqrt{x^2 + y^2 - 1}$ if $x^2 + y^2 - 1 \geqslant 0$. Let s_p be the first intersection of the ray r_p. It exists if $x^2 + y^2 - 1 \geqslant 0$ and has the value $s_p = r_p(-\sqrt{x^2 + y^2 - 1}) = (x, y, -\sqrt{x^2 + y^2 - 1})$. Since $s_p \in S^2$, we obtain the normal $\frac{dF(s_p)}{|dF(s_p)|} = s_p$ for free. This yields the following *CindyScript* code that renders the sphere:

```
lightdir = [.3,.4,-1]; lightcolor = [1,.8,.6]; background = [.7,.7,.7];
```

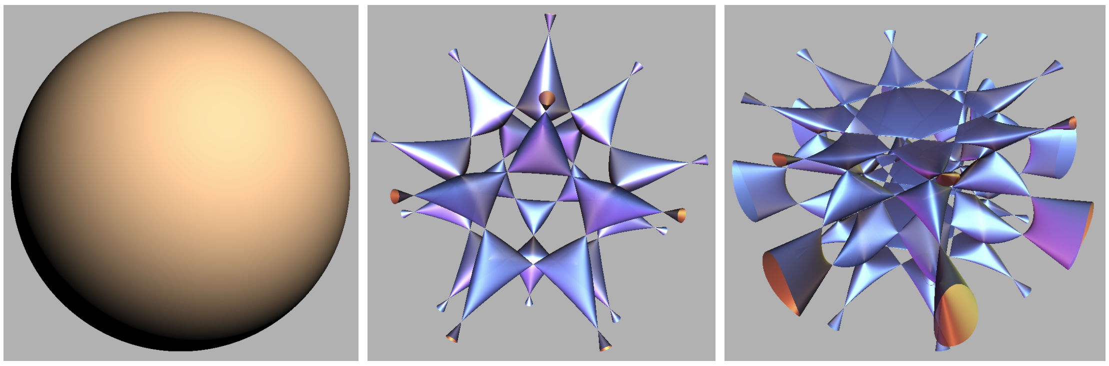


Figure 8.2.: Algebraic surfaces raycasted via the `colorplot` command: (a) the sphere (b) Barth Sextic and (c) Endraß Octic.

```
colorplot(
  if(1-x^2-y^2>=0,
    s = (x,y,-|sqrt(1-x^2-y^2)|); // intersection with the sphere
    (s*lightdir) * lightcolor, // shading based on normal s
    background
  )
);
```

CindyGL can trans-compile the code within the `colorplot` command to GPU shader code, and an image (see Fig. 8.2a) of the sphere is rendered on the GPU.

For general algebraic surfaces of higher degree, appealing visualizations can be rendered in real-time (see Figs. 8.2b and 8.2c for examples). For each ray r_p, we can evaluate $\deg F + 1$ values of F along the ray r_p and obtain the polynomial f_p by interpolation (this avoids the need of symbolic computations), of which the roots have to be determined. However, for higher degrees, the roots of the functions f_p have to be found using a numerical method. So the main difficulty can be reduced to a suitable and as safe as possible method for approximating real-roots of real polynomials.

Several CindyGL-based experiments for different numeric implementation for First-Root have been run and compared: There is an implementation of Aberth-Ehrlich method (compare Applet ▷13), another implementation that iteratively applies Rolle's theorem and bisection method on f_p and its derivatives (see Applet ▷14), and an implementation that utilizes Descartes's rule of signs in Bernstein basis (compare Applet ▷15). The last method turned out to meet the requirements that have been introduced at the beginning of the chapter the most. It runs stable for a large set of 'tame" algebraic surfaces, is very efficient and still has a comparable little compilation time. Thus, in the following, we will present our implementation of this algorithm.

In short, Descartes's rule of signs is used to isolate the real roots of the univariate polynomials f_p. We represent the polynomials f_p in the Bernstein basis to enhance the numeric stability. Furthermore, the Bernstein basis reduces the code-complexity of the implementation and thus its compilation time because the number of sign variations needed to apply Descartes's rule corresponds to the number of sign variations of the coefficients in Bernstein basis with respect to the investigated interval (Sagraloff and Mehlhorn, 2016). Once the real roots are isolated, the bisection method is used to approximate the points where r_p intersects the surface.

8.1. Square-free polynomials

A polynomial over $\mathbb{R}$ (or $\mathbb{C}$) is square-free if it does not contain a multiple complex root.

It turned out that the assumption that the input f_p is square-free over $\mathbb{R}$ dramatically enhances the performance of the FirstRoot. If the computations were performed on a well-representable field such as $\mathbb{Q}$ instead of $\mathbb{R}$, then instead taking the roots of f_p, the roots of its square-free part $\tilde{f}_p = \frac{f_p}{\gcd(f_p, f_p')}$ could be taken (Heintz et al., 1991). However, in the definition of several interesting algebraic surfaces such as the Barth Sextic, irrational numbers occur, thus restricting the used field to $\mathbb{Q}$ is not suitable here, and real numbers are approximated through floating-point numbers. In this case, any gcd implementation turns out to be non-suitable because the problem of computing the roots and the square-free polynomial is numerically ill-posed and multiple roots would split into clusters (Kobel, 2015).

The property of a polynomial to be square-free (over $\mathbb{R}$ or $\mathbb{C}$) is equivalent to its discriminant $\mathrm{Disc}(f_p)$ to be zero. If we assume that the ray map $(p,t) \mapsto r_p(t)$ is affine-linear, the coefficients of $f_p = F \circ r_p$ are polynomial in the coordinates of p, then $\mathrm{Disc}(f_p)$, which is a multiple of the resultant $\mathrm{Res}(f_p, f_p')$, is polynomial in p. Thus, in the general case (if F is not constructed with "evil" intentions, meaning if F has constant discriminant 0), the set of points p, such that f_p is not square-free over $\mathbb{R}$ is a subset of the zero-set of a non-vanishing polynomial and has Lebesgue measure zero. Most rasterized pixels will induce a square-free f_p. In the following, we will ignore this problem, however, some rendering artefacts occur for evil-chosen F (comp. Figure 8.3a). Similar issues also occur for the software SURFER (comp. Figure 8.3b). Some techniques, such as image-post processing or using supersampling could help to reduce the issue.

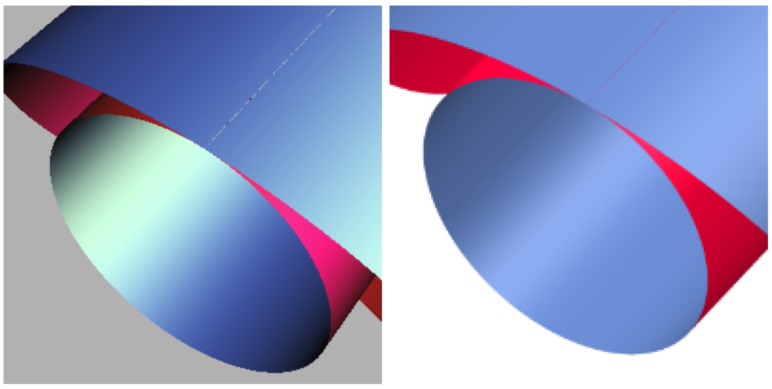


Figure 8.3.: Render errors of the surface $(y^2 + x^2 - 1) \cdot (y^2 + (x - 1)^2 - 4) = 0$ due to double-roots: (a) An image rendered with Applet ▷16, (b) A rendering produced with the software SURFER

8.2. Extracting roots of a univariate square-free polynomial

Many of the following results are taken from Spencer (1994), Sagraloff and Mehlhorn (2016) and Eigenwillig (2008).

Assume that $P \in \mathbb{R}[X]$ is a square-free real polynomial. Our aim is to *isolate* the real roots of P in a given interval $I = (l, u) \subset \mathbb{R}$. That means, for each root $x_k \in (l, u)$ we want to find an interval $I_k = (a_k, b_k) \subset I$ such that x_k is the only root contained in I_k. Since we assumed P to be square-free, we know that this root induces a sign-switch within I_k and $P(a_k) \cdot P(b_k) < 0$. After isolation of the roots in I, classical bisection method can be used on the intervals I_k to find an approximate solution to each root x_k with linear convergence.

How can we determine whether a given interval contains precisely one root?

Descartes' Rule of Sign gives one answer for the interval $(0, \infty)$. If we define $\mathrm{var}(v_0, \ldots, v_n)$ to be the number of sign variations in a sequence $v_0, \ldots, v_n \in \mathbb{R}$ (where zero-entries are skipped), then there is a relation of the sign variations of the coefficients of a polynomial and the number of its positive roots:

Theorem 65 (Descartes' Rule of Sign). *Let $P(x) = \sum_{i=0}^{n} p_i x^i$ with exactly k positive roots, counted by multiplicity. Furthermore, let $v = \mathrm{var}(p_0, \ldots, p_n)$ the number of sign variations in the coefficients of P. Then $v \geqslant k$ and $v \equiv k \mod 2$.*

We will not restate a proof here. A complete proof can be found in (Eigenwillig, 2008, Theorem 2.2). The number of sign variations is easy to compute.

We will mainly utilize two important intermediate consequences of Descartes' Rule of Sign:

- If $v = 0$, there are no positive roots at all (since $k \leqslant v = 0$).
- If $v = 1$, there is exactly one positive root ($k \leqslant v = 1$ has the same parity as v).

How can we use this result to obtain information about the number of roots of a polynomial in an interval $I = (a, b)$? The trick is to apply the Möbius transformation $x \mapsto \frac{x-a}{b-x}$ that maps the interval (a, b) to the interval $(0, \infty)$. Its inverse is $x \mapsto \frac{bx+a}{x+1}$. More precisely, we will define the number $v_I(P) := \operatorname{var}(\tilde{p}_0, \dots, \tilde{p}_n)$ as the number of sign variations of the (formal) polynomial $\tilde{P}(x) = (x+1)^n P(\frac{bx+a}{x+1})$. The term $(x+1)^n$ cancels all the denominators of the rational terms in $P(\frac{bx+a}{x+1})$ and makes $\tilde{P}$ a degree n polynomial. Every root of P in the interval (a, b) corresponds to a positive root of the polynomial $\tilde{P}$ and vice-versa. Hence, we can achieve a Descartes-like statement for the number of roots in (a, b) by counting the sign variations of the coefficients of $\tilde{P}$. Also the multiplicity of the roots would be transferred (Eigenwillig, 2008), but since we are aiming for square-free polynomials, we will not show it here.

How can we obtain the coefficients of $\tilde{P}$ if the polynomial P is given? It would require a certain computational effort to obtain $\tilde{p}_0, \dots, \tilde{p}_n$ from the coefficients $p_1, \dots, p_n$. This makes it difficult to obtain a term for $v_I(P)$ that is efficiently computatble on the GPU. However, it will turn out, if we represent P in the Bernstein basis, instead of the monomial basis $\{x^0, \dots, x^n\}$, then the transformation to $\tilde{P}$ becomes simple. Let us first define the Bernstein-basis for an interval $I = (a, b)$:

Definition 66. The i-th Bernstein polynomial of degree n for the interval $I = (a, b)$ is defined as follows:

$$B_i^n[a, b](x) = \binom{n}{i} \frac{(x-a)^i (b-x)^{n-i}}{(b-a)^n}$$

with $i \in \{0, \dots, n\}$ (graphs can be seen in Figure 8.4a)

It can be shown that the $n + 1$ Bernstein polynomials of degree n are linear independent and thus they form a basis of the $n + 1$-dimensional vector space of univariate polynomials with bounded degree n, which we will from now on call Π_n.

Let us represent our polynomial $P \in \Pi_n$ in the Bernstein basis, i.e. $P = \sum_{i=0}^n b_i B_i^n[a, b]$. The graph of P can be considered as the Bézier curve with the nodes $(a + \frac{i}{n}(b-a), b_i)$ (comp. Figure 8.4b).

With $\phi_I : \Pi_n \to \Pi_n$ we denote the (formal) re-parametrization map that is defined as

$$\phi_I(P)(t) = (t+1)^n P\left(\frac{bt+a}{t+1}\right).$$

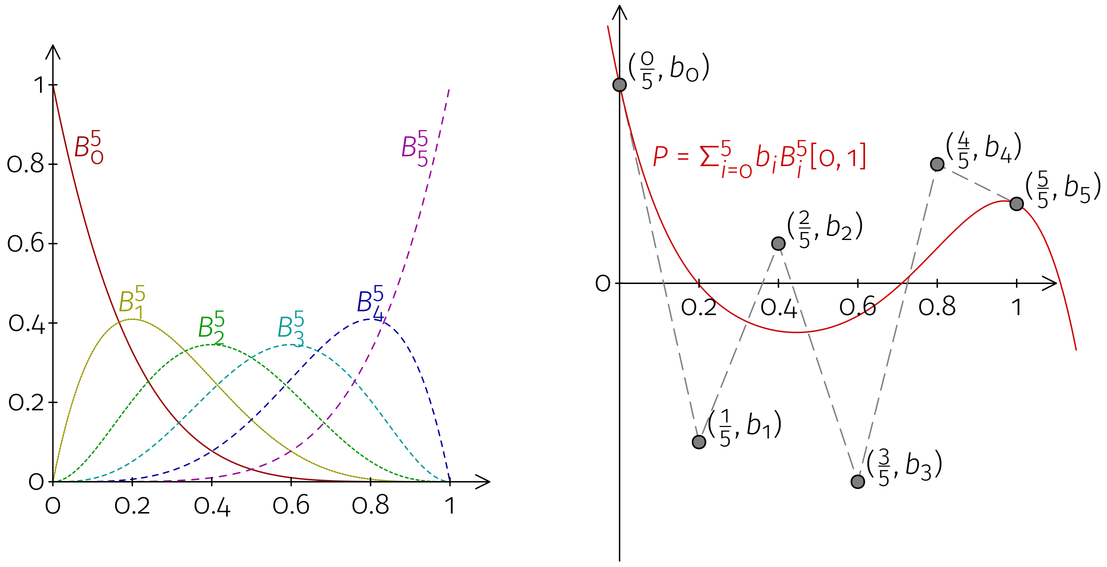


Figure 8.4.: (a) the Bernstein polynomials on the interval $[0, 1]$, (b) The graph of a polynomial in Bernstein form can be seen as a Bézier curve to the control polygon.

$\phi_I : \Pi_n \to \Pi_n$ is well-defined: The term $(t+1)^n$ cancels the denominators of the rational terms in $P\left(\frac{bt+a}{t+1}\right)$, i.e. ϕ_I can be considered as a formal transformation of a degree n polynomial to another polynomial of maximal degree n. Furthermore, $\phi_I : \Pi_n \to \Pi_n$ turns out to be linear: The required properties $\phi_I(\lambda P) = \lambda\phi_I(P)$ and $\phi_I(P+Q) = \phi_I(P) + \phi_I(Q)$ can be easily checked by noting that the evaluation of the polynomials on each side of the equations respectively gives rise to the same evaluated values. Since its values at a finite number of sampling points uniquely determine a polynomial in Π_n, this already proves both equations required for linearity.

We aim to compute the coefficients in the monomial basis of the re-parameterization polynomial $\tilde{P} = \phi_I(P)$. With the linearity of ϕ_I in mind, we can compute the $\tilde{P} = \phi_I(\Sigma_{i=0}^n b_i B_i^n[a,b])$ by transferring the basis elements $B_i^n[a,b]$. The transformation of a Bernstein polynomial gives a nice result:

$$\phi_I(B_i^n[a,b])(t) = (t+1)^n B_i^n[a,b]\left(\frac{bt+a}{t+1}\right) = \binom{n}{i}\frac{(tb-ta)^i(b-a)^{n-i}}{(b-a)^n} = \binom{n}{i}t^i$$

Thus $\tilde{P} = \phi_I(\Sigma_{i=0}^n b_i B_i^n[a,b]) = \Sigma_{i=0}^n b_i \phi_I(B_i^n[a,b]) = \Sigma_{i=0}^n b_i \binom{n}{i} t^i$ and finally $\tilde{p}_i = b_i\binom{n}{i}$. Since the binomial coefficients are positive, the number of sign variation is the same for the coefficients of a polynomial in Bernstein basis for an interval I and the coefficients from the ϕ_I-transferred polynomial in standard basis. Altogether we have accomplished the following variant of Descartes' Rule in Bernstein form:

Lemma 67. *Let $P(x) = \sum_{i=0}^{n} b_i B_i^n[a,b](x)$ be a square-free polynomial in Bernstein-form that has k roots in $I = (a,b)$. Let $v_I = \operatorname{var}(b_0, \dots, b_n)$ denote the number of sign variations of the Bernstein-coefficients of P. Then $v_I \geqslant k$ and $v_I \equiv k \mod 2$.*

The number v_I can also be considered as the number of the intersections of the control polygon of P with the zero-line (For instance, in Figure 8.4b $v_{(0,1)} = 4$ holds). Lemma 67 gives rise to the following function that can be used to compute an approximation of the first root of a polynomial in a given interval.

Function 21: FirstRoot(P, a, b)

```
Input: A square-free polynomial $P \in \mathbb{R}[X]$, $a, b \in \mathbb{R}$, $a < b$, $\varepsilon \in \mathbb{R}_{>0}$
Output: An $\varepsilon$-approximation of the first root of $P$ in $(a, b)$
obtain Bernstein-coefficients $(b_0, \dots, b_n)$ of $P$ for the interval $I = (a, b)$
compute $v_I$, the number of sign variations of $(b_0, \dots, b_n)$
if $v_I = 0$ then
    return $\infty$
else if $v_I = 1$ then
    $x \leftarrow$ run bisection method on $(a, b)$ to approximate root of $P$ up to $\varepsilon$
    return $x$
else
    $m \leftarrow \frac{a+b}{2}$
    if $P(m) = 0$ then
        return min(FirstRoot($P, a, m$), $m$)
    else
        return min(FirstRoot($P, a, m$), FirstRoot($P, m, b$))
```

It is clear, if Algorithm 21 terminates, then the result is correct by Lemma 67.

However, there are two questions to answer. The first question is, how it can be ensured that the algorithm terminates. The second question is, how can it be implemented on a DPM or a GPU, which both do not support recursion in their specification.

8.2.1. Termination

The answer to termination of Algorithm 21 lies in the "one-circle theorem" and the "two-circle theorem" (compare Figure 8.5).

Let $z_1, \dots, z_n \in \mathbb{C}$ be the roots of P and let $\sigma = \min_{i,j \in [n], i<j} |z_i - z_j|$ a number indicating the minimum distance between every pair of roots of P. Since P is square free, $\sigma > 0$ and furthermore, σ can be bounded from above by the discriminant and the leading coefficient of P. Here we will show that σ multiplied by a constant gives a (generous)

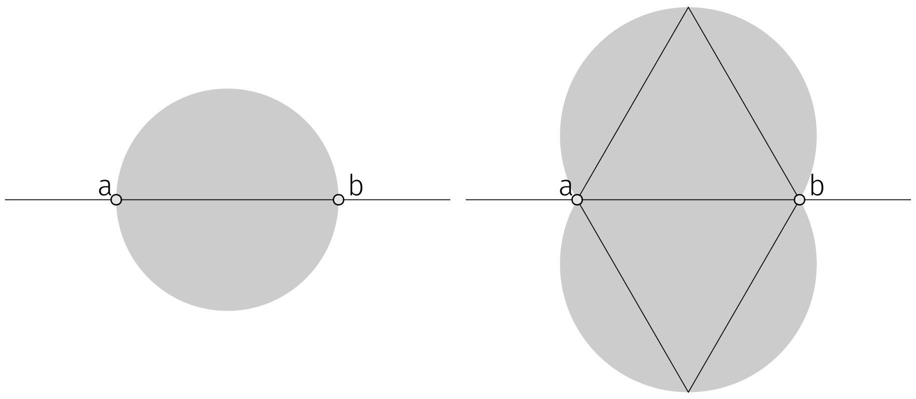


Figure 8.5.: (a) *one-circle theorem*: No roots within the complex circle imply $v_{(a,b)} = 0$.
(b) *two-circle theorem*: If there is exactly one (necessarily real) root within the union of the indicated two circles then $v_{(a,b)} = 1$.

lower bound of the size of the intervals that are considered in the recursion in Algorithm 21: Every interval I of length less than $\sqrt{3}\sigma$, either has $v_I = 0$ or $v_I = 1$. Thus the depth of recursion is bounded and the Algorithm 21 terminates.

The v_I of Lemma 67 gives an upper bound of the number of occurring roots. Luckily, in two clearly defined cases, the upper bound becomes tight. We will obtain a statement by bounding the value v_I from above by the number of roots appearing in the (complex) environment of I. The following two statements are special cases of the partial converse of Descartes' Rule by Obreshkoff (Obreshkov, 1952). These special cases are extracted from Eigenwillig (2008).

Theorem 68 ("one-circle theorem"). *Let $I = (a, b)$ be an interval and $D_I \subset \mathbb{C}$ the open disc with center $\frac{a+b}{2}$ and radius $\frac{|b-a|}{2}$. If a polynomial P has no root in D_I, then $v_I = 0$.*

Proof. A straightforward proof can be obtained if P is transferred via ϕ_I. The number of variations v_I corresponds to the number of sign variations of $\tilde{P} = \phi_I(P)$ in monomial-basis. The Möbius-transformation ϕ_I maps the open disc D_I to the right half-plane $R = \{z \in \mathbb{C} \mid \operatorname{Re}(z) > 0\}$. The one-circle theorem translates into the statement that $\tilde{P}$ has no sign-variation in its monomial basis if there is no root in R.

So let $\tilde{P} \in \mathbb{R}[x]$ be a polynomial with no roots in R. Every root of $\tilde{P}$ thus either is a non-positive real number or a pair of complex conjugate roots with a non-positive real part. Therefore $\tilde{P}(z)$ can be written as the product of the leading-coefficent $\tilde{p}_n \neq 0$ and terms of the form $(z - r)$ with $r \in \mathbb{R}_{\leqslant 0}$ and terms of the form $(z - \xi)(z - \bar{\xi}) = (z^2 - 2\operatorname{Re}(\xi)z + \xi\bar{\xi})$ for $\xi \in \mathbb{C} \setminus \mathbb{R}$ and $\operatorname{Re}(\xi) < 0$. All the summands in each of these factors $(z - \xi)$ and

$(z^2 - 2\operatorname{Re}(\xi)z + \xi\bar{\xi})$ are non-negative, therefore they do not cause a sign switch in their product. Hence all coefficients of $\tilde{P}$ have either the sign of $\tilde{p}_n \neq 0$ or are zero. The number of sign variations carries over to the Bernstein form and thus $v_I = 0$. □

If an interval $I \subset \mathbb{R}$ contains a real root then v_I is odd by Lemma 67. To show that Algorithm 21 terminates, we still need to show that for a sufficiently small interval I containing one root, also $v_I = 1$ holds. The so-called "two-circle theorem" can yield this property:

Theorem 69 ("two-circle theorem"). *Let $I = (a, b)$ be an interval and let T_1 and T_2 be the two equilateral triangles in $\mathbb{C}$ having I as one side. Let D be the union of the two open discs which have the circumcircle of T_1 and T_2 respectively as their boundary. If a polynomial has exactly one root in D, then $v_I = 1$.*

The proof for the "two-circle theorem" is essentially similar to the proof of the one-circle theorem. The set D, which is the union of two open discs tangent to a and b is mapped via ϕ_I to the union of half-planes with 0 on its boundary. Möbius-transformations preserve angles and the circles of the theorem intersect the real-line at angle $\frac{\pi}{3}$. Thus the angle of the two half-planes that together form $\phi_I(D)$ intersecting the real axis at 0 is also $\frac{\pi}{3}$. If there is exactly one root in $\phi_I(D)$, it must be real, and it will cause one sign variation in the coefficients of $\phi_I(P)$ if $\phi_I(P)$ has degree 1. By induction on the degree of $\phi_I(P)$ it can be shown that any root outside of $\phi_I(D)$ does not induce a sign-flip. Each factor for a non-positive real root or a pair of complex roots outside of $\phi_I(D)$ does not contribute to a sign variation. This check is technical (though elementary) and is omitted here. A complete proof can be found in Eigenwillig (2008).

From the two theorems above we can conclude that Algorithm 21 terminates for square-free polynomials: Once $|b - a| < \sqrt{3}\sigma$, the maximal number of roots in the one-circle shape or two-circle shape above the interval (a, b) is 1 and thus either $v_I = 0$ or $v_I = 1$ and the recursion breaks. Furthermore any value k such that $\frac{(u-l)}{2^k} < \sqrt{3}\sigma$ is an upper bound of the maximum depth of the recursion. Furthermore, only in the proximity of one of the n roots the recursion does not stop earlier. This bounds the number of investigated intervals of Algorithm 21 to $O(n \log \frac{u-l}{\sigma})$.

8.2.2. Recursive tree traversal

Algorithm 21 is a recursive scheme to isolate and find roots.

The problem is that recursions, in general, cannot be implemented directly on GPU shader code. The closest constructs that we are allowed to use in this context are loops. Moreover, it is not possible to build a stack or queue to mimic the recursion.

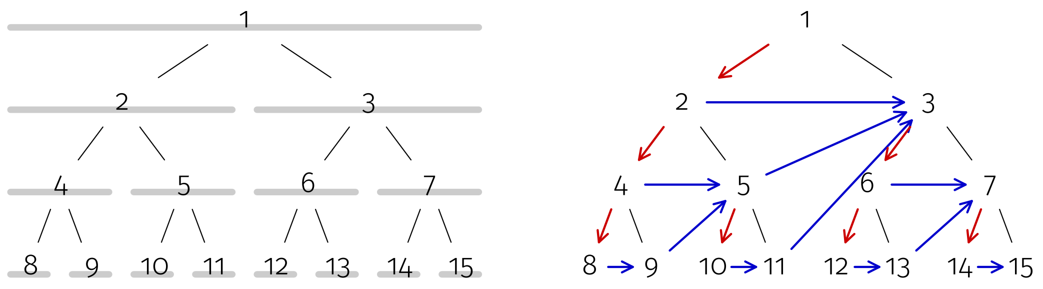


Figure 8.6.: Indexing binary trees such that node n has the two children $2n$ and $2n + 1$: (a) Enumeration of intervals, (b) The functions GoDeeper and GoNext that determine the order of traversed nodes for a DFS on a (truncated) binary tree.

Because the test $P(\frac{a+b}{2}) = 0$ in Line 10 of Algorithm 21 almost never holds, we will ignore it for reasons of clarity and comprehensibility. Without this case, Algorithm 21 traverses a binary tree where each node either has zero or two children. Traversing a binary problem without recursion or the implementation of a call-stack can be handled by introducing unique identifiers for each node of the tree. These identifiers were inspired by an indexing scheme that Cormen et al. (2009) used for indexing an in-place HEAP data-structure. The node representing the starting interval (l, u) gets the identifier 1, its children, representing the intervals $(l, \frac{l+u}{2})$ and $(\frac{l+u}{2}, u)$ obtain the identifiers 2 and 3. More generally, a node with identifier n that represents the interval (a, b) has the children $2n$ and $2n+1$ that represent the intervals $(a, \frac{a+b}{2})$ and $(\frac{a+b}{2}, b)$ respectively (comp. Figure 8.6a). This gives rise to an injective assignment of identifiers to the nodes: The identifier n of a node is unique since the parity of n indicates whether the node is the right or left child of its (by induction unique) parent with identifier $\lfloor \frac{n}{2} \rfloor$. The binary representation of n read from the highest bit to the lowest gives the exact position of the node n in the binary tree: If the leading 1 is ignored, then each 0 or 1 corresponds to a left or right turn in the path from the root to the node.

Instead of performing a recursion, where usually a call stack is built up, we use an iterative approach where only the identifier n of the currently traversed node is stored.

Based on n and $\mathrm{flp2}(n) = 2^{\lfloor \log_2(n) \rfloor}$, n rounded down to the next power of 2, the corresponding interval can be computed via

$$(a, b) = (l + (u - l)\frac{n - \mathrm{flp2}(n)}{\mathrm{flp2}(n)}, l + (u - l)\frac{n - \mathrm{flp2}(n) + 1}{\mathrm{flp2}(n)}).$$

This formula follows from induction on $\lfloor \log_2 n \rfloor$.

Now, let us perform the DFS on a binary tree with indexed nodes without using recursion. Every node n either has two children or is a leaf and does not have any children. This corresponds to the recursion in Algorithm 21: Either both children are visited, or the recursion does not go deeper. Depending on the case, the next node either will be $\textsc{GoDeeper}(n)$ or $\textsc{GoNext}(n)$ (comp. Figure 8.6b).

In the first case, if the recursion goes more in-depth, the left child is visited first, and we yield $\textsc{GoDeeper}(n) = 2 \cdot n$.

If we do not go deeper during the DFS, i.e., the recursion stops, then we distinguish two cases to determine the next node to be traversed. If n is even, then the currently visited node was a left child of a parent. Thus the right child of the same parent, which has index $2\lfloor\frac{n}{2}\rfloor + 1 = n + 1$ will be traversed next. If n is odd, then the current node is a right child. Determining the next node is more difficult, so we reduce the problem if we recursively take the next breaking node from its parent with identifier $\lfloor\frac{n}{2}\rfloor = \frac{n-1}{2}$. So for $n > 1$ we yield the following recursive rule for the next node after a return:

$$\textsc{GoNext}(n) = \begin{cases} n+1 & \text{if } n > 1 \text{ is even} \\ \textsc{GoNext}(\frac{n-1}{2}) & \text{if } n > 1 \text{ is odd} \\ 1 & \text{if } n = 1 \end{cases} \tag{8.1}$$

We forced termination of the recursion by further defining $\textsc{GoNext}(1) = 1$. Without the special treatment for the case $n = 1$, the recursion of $\textsc{GoNext}(n)$ would end in a 0-1 circle for every n of the form $2^k - 1$. In this case, the DFS should end. Then the recursion terminates in all cases and its value can be resolved to a non-recursive expression, for which we need another binary helping function. Let

$$\operatorname{ttz}(n) = \min\{\frac{n}{2^k} \mid 2^k \text{ with } k \in \mathbb{N}_0 \text{ divides } n\} \in \mathbb{N}$$

the number that is obtained from n if all trailing zeros are truncated from its binary representation. $\operatorname{ttz}(n) = n$ holds for odd n and $\operatorname{ttz}(n) = \operatorname{ttz}(\frac{n}{2})$ holds for even n. If bitwise arithmetic operations on integers are available[4], then $\operatorname{ttz}(n)$ on integers can be efficiently computed through them: $\operatorname{ttz}(n) =$ `n/(n & -n)` where `&` is the bitwise and-operator (Warren, 2013).

The recursion in Equation (8.1) resolves to the simple expression

$$\textsc{GoNext}(n) = \operatorname{ttz}(n+1).$$

Again, induction on $\lfloor\log_2 n\rfloor$ proves the equality: If $n = 1$, then $\operatorname{ttz}(n+1) = 1 = \textsc{GoNext}(n)$. Let $n > 1$. If n is even, then $n+1$ is odd and $\operatorname{ttz}(n+1) = n+1 = \textsc{GoNext}(n)$. If n is odd, then $\operatorname{ttz}(n+1) = \operatorname{ttz}(\frac{n+1}{2}) = \operatorname{ttz}(\frac{n-1}{2}+1)$, and by induction, $\operatorname{ttz}(\frac{n-1}{2}+1) = \textsc{GoNext}(\frac{n-1}{2}) = \textsc{GoNext}(n)$.

[4] which is the case for WebGL 2 through OpenGL ES 3.0

These two computable helping functions GoDeeper and GoNext give rise to the following non-recursive function that is equivalent to Algorithm 21:

Algorithm 22: FirstRootNonRecursive(P, l, u)

```
Input: A square-free polynomial $P \in \mathbb{R}[X]$, $l, u \in \mathbb{R}$, $l < u$, $\varepsilon \in \mathbb{R}_{>0}$
Output: An $\varepsilon$-approximation of the first root of $P$ in $(l, u)$
1  $n \leftarrow 1$
2  $x \leftarrow \infty$
3  do
4      $s \leftarrow 2^{\lfloor \log_2 n \rfloor}$
5      $a \leftarrow l + (u - l)\frac{n-s}{s}$
6      $b \leftarrow l + (u - l)\frac{n-s+1}{s}$
7      obtain Bernstein-coefficients $(b_0, \ldots, b_n)$ of $P$ for the interval $I = (a, b)$
8      compute $v_I$, the number of sign variations of $(b_0, \ldots, b_n)$
9      if $v_I = 0$ then
10         $n \leftarrow$ GoNext($n$) // For integers, this is equal to
            (n+1)/((n+1) & -(n+1)) where & is the bit-wise AND.
11     else if $v_I = 1$ then
12         $x \leftarrow$ run bisection method on $(a, b)$ to approximate root of $P$ up to $\varepsilon$
13         return $x$
14     else
15         $n \leftarrow$ GoDeeper($n$) $= 2 \cdot n$
16 while $n \neq 1$
17 return $x$
```

On particular older shader models, also this do-while loop is not valid. If it suffices to render pixels with an upper bound on σ, then the do-while-loop can be replaced with a loop with a fixed number of iterations. The number of iterations is a constant times $n \log \frac{u-l}{\sigma}$.

8.3. Interpolation of a function to Bernstein coefficients

In Line 7 of Algorithm 22, the representation in Bernstein form of $f_p = F \circ r_p$ on a given interval (a, b) is required. Given (a, b) and r_p, how can this representation be obtained from the trivariate polynomial F?

One approach would be to determine for the ray r_p the polynomial $F \circ r_p$ in monomial basis. Then a further transformation could yield the Bernstein form for $F \circ r_p$ on each of the intervals. A pure symbolic approach would not be suitable for GPU computations,

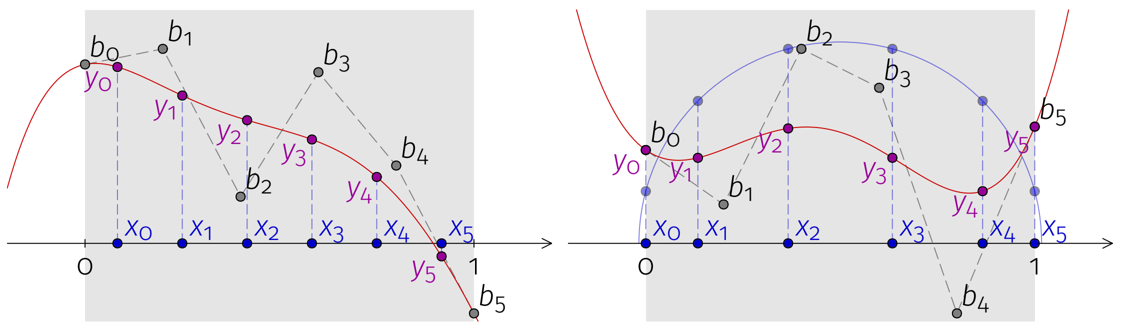


Figure 8.7.: The Bernstein-coefficients $b = A_x{}^{-1}y$ of $f_p^{(\alpha,\beta)}$ on the interval $I = (0,1)$ can be obtained through interpolation at the values $y_k = f_p^{(\alpha,\beta)}(x_k)$ for a fixed set of interpolation nodes x: (a) Equidistant nodes x, (b) A better choice for x and local interpolation are the expanded Chebyshev nodes for a slightly expanded interval such that b coincides with y at the boundary. Using $b_0 = y_0$ and $b_n = y_n$ reduces the numeric noise and ensures the correctness of $v_{(\alpha,\beta)} \mod 2$.

and a numeric conversion from monomial basis to Bernstein basis is ill-conditioned (Farouki, 1991).

Another computational simpler approach is to use polynomial interpolation to obtain a representation in Bernstein-basis. Let us suppose we are interested in the Bernstein coefficients of $f_p = F \circ r_p : \mathbb{R} \to \mathbb{R}$ in the interval (α, β). We will study

$$f_p^{(\alpha,\beta)} := F \circ r_p \circ (\lambda \mapsto \alpha + \lambda(\beta - \alpha)).$$

The Bernstein coefficients of $f_p^{(\alpha,\beta)}$ for the interval $[0,1]$ coincide with the Bernstein coefficients of f_p in the interval (α,β). So, we are interested in the $[0,1]$-Bernstein coefficients of $f_p^{(\alpha,\beta)}$. The function $f_p^{(\alpha,\beta)}$ is a polynomial of degree n (the degree of F). Therefore interpolation at $n+1$ points uniquely determines the polynomial $f_p^{(\alpha,\beta)}$. Let us fix a vector of interpolation nodes $(x_0, \dots, x_n) \in \mathbb{R}^n$, $x_0 \leqslant x_1 < \dots \leqslant x_n$ (the values x_k are not necessarily strictly contained in $[0,1]$). We compute the values $y_k = f_p^{(\alpha,\beta)}(x_k)$. As we shall see in the following, the $[0,1]$-Bernstein coefficients of $f_p^{(\alpha,\beta)}$ depend linearly on the evaluated values y. The inverse *Bernstein-Vandermonde matrix*, which we are going to define, describes the linear map that can be used to obtain the $[0,1]$-Bernstein coefficients.

With $A_x \in \mathbb{R}^{n\times n}$ we denote the *Bernstein-Vandermonde matrix* for the interval $[0,1]$, degree n and the interpolation nodes x, i.e.

$$A_x = \begin{pmatrix} \binom{n}{0}(1-x_0)^n & \binom{n}{1}x_0(1-x_0)^n & \dots & \binom{n}{n}x_0^n \\ \binom{n}{0}(1-x_1)^n & \binom{n}{1}x_1(1-x_1)^n & \dots & \binom{n}{n}x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ \binom{n}{0}(1-x_n)^n & \binom{n}{1}x_n(1-x_n)^n & \dots & \binom{n}{n}x_n^n \end{pmatrix}$$

The *Bernstein-Vandermonde matrix* A_x fulfills $A_x b = y$ with

$$y_i = \Sigma_{k=0}^{n} b_k B_k^n[0,1](x_i) = f_p^{(a,b)}(x_i)$$

where $b_0, \dots b_n$ are the Bernstein coefficients of $f_p^{(a,b)}$ for the interval $[0,1]$. A_x is regular if all nodes in x are distinct (Marco et al., 2007), and thus $b = A_x^{-1}y$.

Let us rewrite this interpolation of $f_p^{(a,b)}$ in [0,1]: The interpolation nodes of f_p for an arbitrary interval (α,β) can be computed through $\tilde{x}_k = \alpha + x_k(\beta-\alpha)$ and $\tilde{y}_k = f_p(\tilde{x}_k)$. The Bernstein-coefficients $\tilde{b}$ of f_p in the interval (α,β) can then be computed by $\tilde{b} = A_x^{-1}\tilde{y}$.

Since A_x is independent from the used ray and interval, it suffices to compute A_x^{-1} once on the CPU in high precision. This matrix then can be used multiple times (for each ray and even optionally for each recursion node) within Algorithm 22, which will be parallelized on the GPU, to obtain the Bernstein-coefficients.

We will distinguish two different implementations for Line 7 of Algorithm 22 that directly or indirectly use the Bernstein-Vandermonde matrix. However, as we will see those two implementations differ in their compilation time and their numeric stability:

Repeated local interpolation For each interval $[a,b]$, the Bernstein coefficients of f_p in this interval are obtained through interpolation of $f_p^{(a,b)}$ in the interval $[0,1]$.

A single global interpolation de Casteljau (1963)'s algorithm is a numeric stable scheme to subdivide Bézier curves or polynomials in Bernstein form. Given a polynomial in Bernstein form, it can be used to compute Bernstein coefficients on a subinterval. Only once for the interval $[l,u]$ (which indicates where r_p intersects the clipping sphere), the Bernstein coefficients are obtained through interpolation of $f_p^{(l,u)}$. For each of the sub-intervals $[a,b] \subset [l,u]$, De Casteljau's algorithm can be used (twice) to extract the Bernstein-coefficients of the sub-intervals from the Bernstein coefficients of the global interval $[l,u]$. A recursive subdivision scheme with De Casteljau's algorithm that spilts the Bernstein coefficients in each recursive step of FINDROOT can not be implemented without a call-stack of a flexible size.

The first approach requires more evaluations of F. After that, each computations of both approaches requires $O(n^2)$ operations. Over real numbers, those algorithms are mathematically equivalent. However their numeric properties differ.

In the next section, we use Applet ▷16 to run numeric experiments to compare these approaches. We will also study some further adjustments, such as the choice of the interpolation nodes x. Moreover, the alternatives to the interpolation that avoid the need of using the inverse Bernstein-Vandermonde matrix in the computation of $y = A_x^{-1}b$ are studied as well.

8.3.1. Enhancing the numeric stability of the interpolation

Currently, *CindyGL* trans-compiles the described algorithm successfully to *GLSL* shading language 1.0, which provides 32-bit floats as the highest possible precision (Simpson and Kessenich, 2009). This comparable low accuracy makes some numeric considerations necessary to allow for rendering surfaces of higher degree.

A good set of benchmark examples of variable degree are generalizations of Chmutov surfaces. Let T_n is the nth Chebyshev polynomial of the first kind and let

$$C_n(x, y, z) := T_n(x) + T_n(y) + T_n(z) + 1.$$

$C_n(x, y, z)$ is a tri-variate polynomial of degree n. The zero sets of $C_n(x, y, z)$ with arbitrary $n \in 2\mathbb{N}$ give rise to algebraic surfaces that are rich in detail and can be used to test the numeric behavior of the visualization of high-degree surfaces (comp. Figures 8.8 and 8.10). These surfaces for $n \leqslant 10$ are rendered well for any of the presented approaches. Under the provided precision of 32-bit floats, none of the variations of the presented real-time approach can render the surfaces $C_n(x, y, z) = 0$ for $n > 20$ in a satisfactory manner. Values of n between 10 and 20 give form test-examples to fine-tune the numeric stability.

Some approaches show artifacts after zooming far away (equivalently increasing the size of the clipping sphere). An example is the comparable low-degree Taubin's heart surface defined by $(x^2 + \frac{9}{4}y^2 + z^2 - 1)^3 - x^2z^3 - \frac{9}{80}y^2z^3 = 0$ (see first column of Figures 8.8 and 8.10). A well chosen numeric method should not expose this behavior. So the example of this zoomed far away surface has been added to the set of benchmark examples as well.

All these experiments were carried out with Applet ▷16, where several interpolation parameters can be adjusted.

Starting point: Equidistant nodes and interpolation by computing a matrix-vector product

In Figures 8.8a to 8.8d, the Bernstein-coefficients for each sub-interval (a, b) in Algorithm 22 have been obtained through local interpolation and a naive interpolation-approach $b = A_x^{-1}y$. The matrix A_x^{-1} is computed once in high precision on the CPU. The compilation time of the matrix-vector product in $b = A_x^{-1}y$ is low and the efficiency at running-time is high, since the evaluation is accelerated through the usage of built-in operations.

In Figure 8.8a, equidistant nodes for $(0, 1)$ (excluding 0 and 1) have been used. The visual results are far from satisfactory.

Chebyshev nodes

The results, in particular for C_{10}, can be enhanced if instead of equidistant nodes, Chebyshev nodes in the same interval $(0, 1)$ are used for interpolation (compare Figure 8.8a vs Figure 8.8b). For a given n, the Chebyshev nodes over the interval $(-1, 1)$ are defined as the roots of the Chebyshev polynomials T_n, i.e. the values $x_k = \cos(\pi \frac{2\cdot k+1}{2\cdot n+2})$ for $k \in \{0, 1, \dots, n\}$. For other intervals, the nodes have to be rescaled accordingly. In a certain sense, these nodes are the best choice for interpolation nodes (Mason and Handscomb, 2002; Stewart, 1996): The Chebyshev nodes $(x)_{k=0}^{n}$ minimize the term $\sup_{x\in(-1,1)} \left|\prod_{k=0}^{n}(x - x_k)\right|$ to 2^{-n}. The term $\sup_{x\in(-1,1)} \left|\prod_{k=0}^{n}(x - x_k)\right|$ plays a crucial role in bounding the error for node-based interpolation. If P_n is a degree n interpolating polynomial of f in $(-1, 1)$ passing through the nodes $(x_k, f(x_k))$, then for every $x \in (-1, 1)$ there is some $\xi \in (-1, 1)$ such that

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^{n}(x - x_k).$$

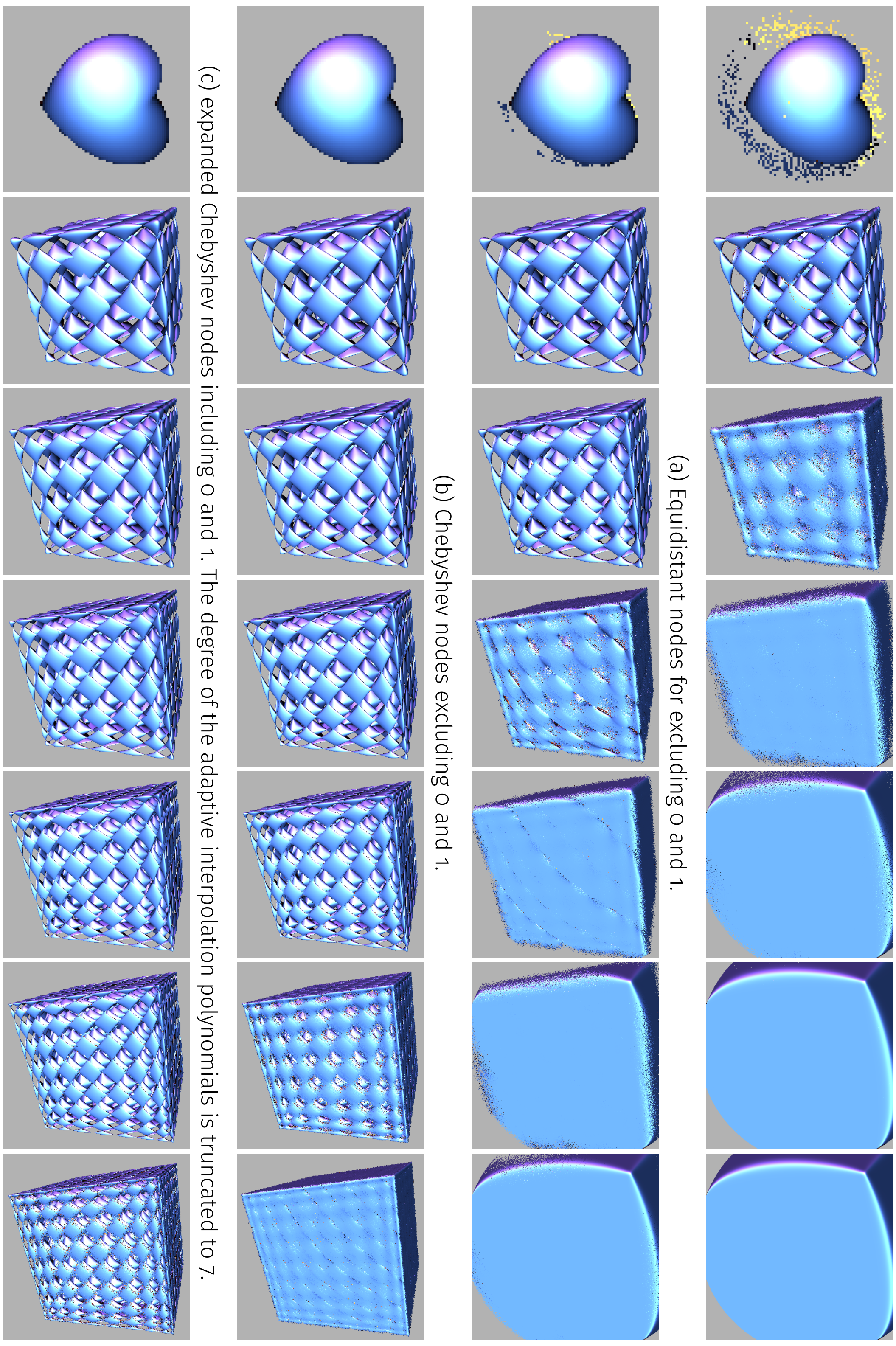
If the values x_k are set to the Chebyshev nodes the error of the interpolating polynomial within $(-1, 1)$ can be bound as follows:

$$|f(x) - P_n(x)| \leqslant \frac{\sup_{\xi\in(-1,1)} \left|f^{(n+1)(\xi)}\right|}{2^n(n+1)!}.$$

For other intervals (a, b), the values x_k have to be rescaled accordingly and the error bound within the interval (a, b) becomes:

$$\left|f(x) - P_n\right| \leqslant \frac{\sup_{\xi\in(a,b)} \left|f^{(n+1)}(\xi)\right|}{2^n(n+1)!} \left(\frac{b-a}{2}\right)^{n+1}. \tag{8.2}$$

Perturbations in the values y_k have a small impact on the interpolating polynomial.



(a) Equidistant nodes for excluding 0 and 1.

(b) Chebyshev nodes excluding 0 and 1.

(c) expanded Chebyshev nodes including 0 and 1. The degree of the adaptive interpolation polynomials is truncated to 7.

(d) far-expanded Chebyshev nodes including 0 and 1. The degree of the adaptive interpolation polynomials is truncated to 7.

Figure 8.8.: In different columns: Renderings of a far zoomed out Taubin's heart surface of degree 6 and the surfaces $T_n(x) + T_n(y) + T_n(z) + 1 = 0$ for $n \in \{10, 12, 14, 16, 18, 20\}$. All algorithms interpolate locally in order to compute v_i in FirstRoot.

Expanding the Chebyshev nodes

A trick that solves the noise problem occurring in the far zoomed out structure in Figures 8.8a and 8.8b, is to include 0 and 1 into the set of interpolation nodes. For instance, the previously defined nodes could be stretched, i.e., the interpolation is performed for a wider interval. Instead of choosing the Chebyshev for the interval $(0, 1)$, the Chebyshev nodes for a slightly bigger interval $(0 - \delta, 1 + \delta)$ such that 0 and 1 are contained in x can be chosen (as in Figure 8.7b). We will call these nodes *expanded Chebyshev nodes*. If $x_{k_0} = 0$ and $x_{k_1} = 1$, then the k_0th and k_1th columns in the matrix A_x will become unit vectors and consequently the corresponding rows in A_x^{-1} will become unit vectors. The appearance of those unit vectors corresponds to the fact that the Bezier-curve always traverses the first and last node of its control polygon. Including 0 and 1 to the base nodes thus guarantees (up to evaluation of F) the correctness of $b_0 = f_p^l(0)$ and $b_n = f_p^l(1)$, which then implies that $v_l \mod 2$ will be computed correctly. In particular, the two base-cases $v_l = 0$ and $v_l = 1$ in Algorithm 22 will not be interchanged. The visual difference becomes apparent in the first image (out-zoomed Taubin's heart) of Figure 8.8c vs. Figure 8.8b, where we used both polynomials of degree 6; We only choose the 7 Chebyshev interpolation nodes for slightly different intervals.

Truncating the degree of interpolation and non-algebraic surfaces

A further strategy is to restrict the degree of the polynomials to a fixed degree and to use fewer interpolation nodes than required. That increases the efficiency and dramatically reduces the compilation time for high-degree surfaces (during the compilation, the loops are unrolled; thus the compilation time grows quadratically in n) and furthermore, it prevents numeric noise for high-degree surfaces. The used Bernstein-coefficients then do not correspond to the polynomial f_p in a given interval anymore. Instead, these Bernstein-coefficients correspond to an adaptive interpolation of lower degree. This strategy is inspired by Boyd (2013), who uses adaptive Chebyshev interpolation to find roots of univariate equations. We rendered the high-degree images in Figure 8.8c by bounding the degree of the polynomial for interpolation by 7, i.e., 8 interpolation nodes have been chosen. With adaptive interpolation of comparable low degree, it is possible to visualize surfaces of drastically higher-degree, though there is no general mathematical justification that some spots might be omitted while rendering. For instance, one could construct unfortunate F that mimics another $\tilde{F}$ that is only equal at the interpolation points.

Nevertheless, for common algebraic surfaces that do not show too many fine structures, the results are still good, and this approach makes it possible to render a wide

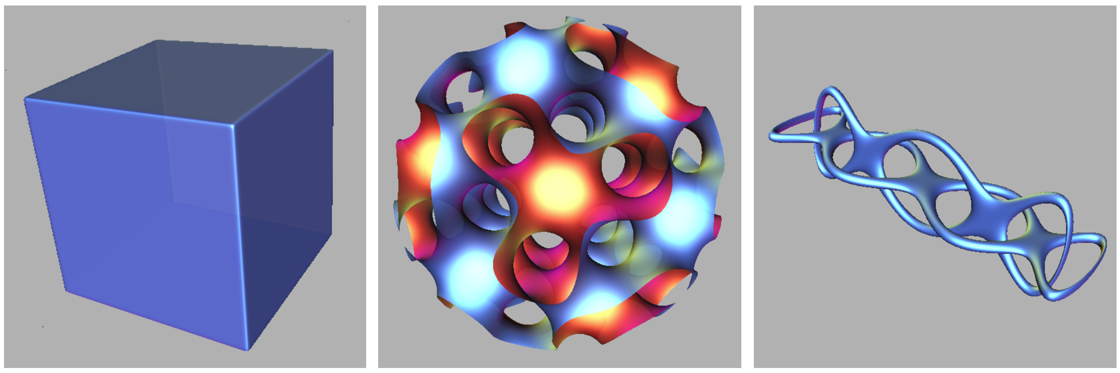


Figure 8.9.: Renderings of surfaces of high-degree. Polynomials of degree 7 were used for adaptive approximations in order to approximate the roots. The local interpolation was performed on expanded Chebyshev nodes in $(0-\delta, 1+\delta)$ such that 0 and 1 are interpolation nodes. (a) degree 80: a "cube" $x^{80} + y^{80} + z^{80} - 1 = 0$, (b) The gyroid $\cos(x)\sin(y) + \cos(y)\sin(z) + \cos(z)\sin(x) = 0$ as an example of a non-algebraic surface, (c) degree 28: $\Big(T_3(\frac{x}{3})^2 + y^2 + z^2 + \frac{99}{100})^2 - 4(T_3(\frac{x}{3})^2 + y^2)\Big) \cdot \Big((T_4(\frac{7x}{2})^2 + y^2 + z^2 + \frac{99}{100})^2 - 4(T_4(\frac{7x}{2})^2 + z^2)\Big) = 0$

class of simple surfaces such as cubes up to degree 100 without visible errors. In Figure 8.9a and Figure 8.9c surfaces of degree 80 and 28 have been well visualized through the same approach that uses only 8 expanded Chebyshev nodes and adaptive interpolation of degree 7. And also some non-algebraic surfaces such the Gyroid can be rendered with this approach surprisingly well (comp. Figure 8.9b). A heuristic explanation for that behaviour is that the (expanded) Chebyshev nodes give rise to a very accurate approximation. Let P_n the degree n interpolation polynomial that corresponds to the computed Bernstein coefficients in interval (a, b). According to Equation (8.2), the approximation error of P_n can be bound by

$$|f_p(x) - P_n| \leqslant \frac{\sup_{\xi \in (l,u)} \left| f_p^{(n+1)}(\xi) \right|}{2^n (n+1)!} \left(\frac{b-a}{2} \right)^{n+1} \delta^{n+1}$$

where $\frac{1}{\delta} = \cos(\frac{\pi}{2n+2})$ is required in this formula because of the expanded nodes. For $n = 7$, each subdivision step, which halves the size of the investigated interval, scales down the error by a further factor $\left(\frac{1}{2}\right)^8 = \frac{1}{256}$. The approximation error depends on the $(n+1)$th derivative of f_p, which can be bound on the compact clipping space if F is analytic. The smoother F, the better the bound. No evidence has been found of the termination of the algorithm that uses only approximations. Aborting the recursion on small intervals is helpful and helps the visualization. Furthermore, wrong decisions are made if $v^{P_n}_{(a,b)} \neq v^{f_p}_{(a,b)}$ and $v^{P_n}_{(a,b)} \in \{0, 1\}$. Using extended Chebyshev nodes, i.e. including the boundaries to the interpolation points guarantees that $v^{P_n}_{(a,b)} \equiv v^{f_p}_{(a,b)}$ mod 2 and the two base-cases for 0 and 1 are not interchanged. Other errors, for instance overlooking three close roots of f_p as one root of P_n, might occur.

However, accurate real-time renderings for the surface $C_n(x, y, z) = 0$ for $n \geqslant 18$ still remain a challenge. An experiment using even further expanded Chebyshev nodes induced more stability, though for $n = 20$ some flickering rendering errors still occur (Figure 8.8d). Here, the nodes

$$x^{(d)} = \{-0.3827, -0.2483, \mathbf{0}, 0.3244, 0.6756, \mathbf{1}, 1.2483, 1.3827\}$$

were used instead of

$$x^{(c)} = \{\mathbf{0}, 0.0761, 0.2168, 0.4005, 0.5995, 0.7832, 0.9239, \mathbf{1}\},$$

and the visual results improved (Figure 8.8c). At first glance, this enhancement seems counter-intuitive, because the nodes $x^{(c)}$ should give rise to a better interpolation within the "decision-making interval" $[0, 1]$ than $x^{(d)}$. At this point, the reason for errors seems to lie in an inexact evaluation of f_p and F. The strategy to draw conclusions from evaluations of f_p at points that are very close to each other is problematic.

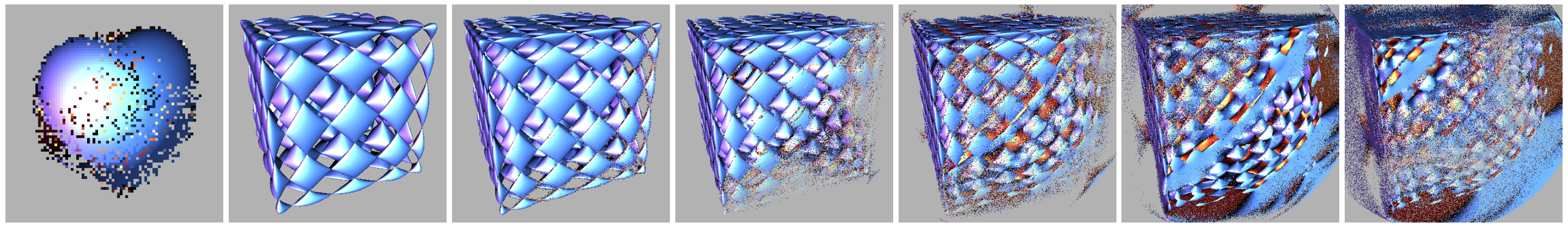
Exploration of variants using global interpolation and de Casteljaus' algorithm for refinement

A strategy to circumvent the aforementioned source of numeric errors would be to avoid interpolation on narrow intervals and use de Casteljau (1963)'s algorithm to obtain the Bernstein coefficients for subintervals in Algorithm 21.

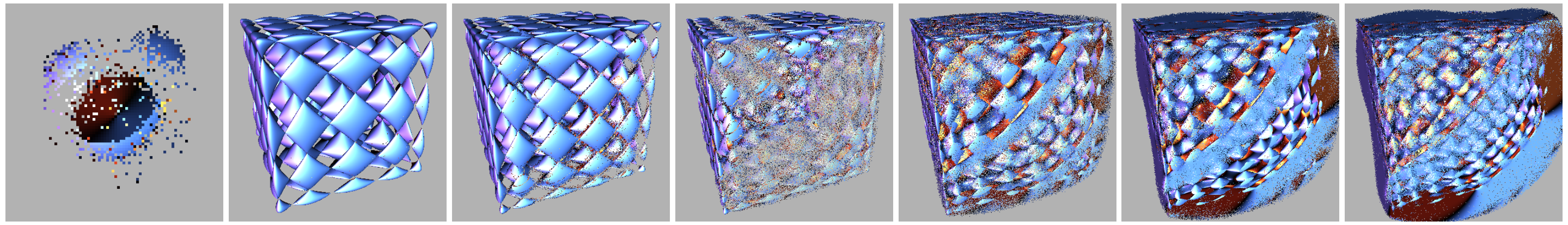
In Figures 8.10a and 8.10b, the $[0, 1]$-Bernstein-coefficients for $f_p^{(l,u)}$ are calculated only once per pixel. Then the Bernstein-coefficients of subintervals are obtained numerically stable by using de Casteljau's algorithm. Compared to non-truncating local approaches (Figures 8.8a and 8.8b), the results for high-degree surfaces are enhanced, but some numeric noise also arises in surfaces of lower degree.

Increasing the numeric stability of interpolation

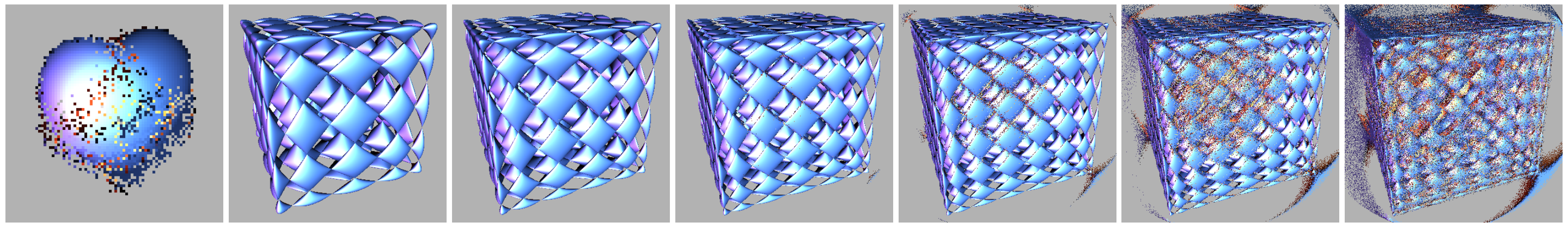
In general, the Bernstein-Vandermonde matrix A_x is ill-conditioned and utilizing its inverse might be a reason for numeric flaws (Marco et al., 2007). Numeric enhancements can be crucial, because we are aiming to produce an implementation for WebGL where only single precision floats (32-bit) are supported.



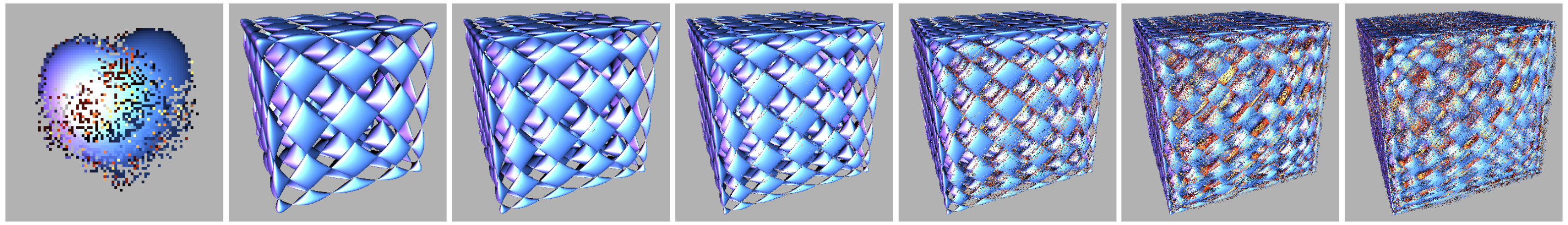
(a) Equidistant nodes. The initial Bernstein coefficients are obtained through $b = A_x^{-1}y$.



(b) Chebyshev nodes. The initial Bernstein coefficients are obtained through $b = A_x^{-1}y$.



(c) Equidistant nodes. The initial Bernstein coefficients are obtained by algorithm of Marco et al. (2007).



(d) Chebyshev nodes, enhanced interpolation. The initial Bernstein coefficients are obtained by algorithm of Marco et al. (2007).

Figure 8.10.: Same surfaces as in Figure 8.8. Global interpolation gives Bernstein-coefficients once for each ray. Subdividing these Bernstein-coefficients via De Casteljau's Algorithm gives the Bernstein-coefficients for each investigated sub-interval.

A more numerically stable algorithm for interpolation into the Bernstein-form of the same asymptotic speed is given by Marco et al. (2007). This algorithm can replace the computation of A_X^{-1} and the matrix-vector product $b = A_X^{-1}y$ in this context. Based on Neville elimination, a bidiagonal decomposition of the Bernstein-Vandermonde matrix A_X is computed once on the CPU, which then can be used in $O(n^2)$ to interpolate the polynomial $f_p^{(l,u)}$. As a drawback, the algorithm of Marco et al. (2007) requires that the interpolation nodes x are strictly contained within $(0, 1)$. In Applet ▷16, the algorithm enhances the approaches with global interpolation and subdivision, albeit for $n \geqslant 16$ significant render errors are still visible (compare Figures 8.10c and 8.10d).

If Marco et al. (2007)'s algorithm is applied to the local approaches in Applet ▷16 it gives no visible improvements. An explanation for that is that only low-degree interpolation renders there well for local interpolation without errors. These low-degree interpolations are stable enough for the naive approach utilizing A_X^{-1}. The bottleneck lies rather in the inaccuracy of low-degree interpolation than in inaccurate interpolation. The enhanced interpolation algorithm cannot be applied to visual better variants of the local variants that rely on the inclusion of the points 0 and 1, and so its possible benefits of higher-degree surfaces cannot be used.

Conclusion

For real-time visualization, the approach that uses adaptive polynomials of bounded or truncated degree that are obtained through local interpolation on expanded Chebyshev nodes turned out to be the most suitable variant (compare Figures 8.8c and 8.8d). By bounding the degree of the polynomials the problems due to numerically weak algorithms do not show up. The lower code complexity reduced the compilation times drastically. The truncation allows for the visualization of several algebraic surfaces of high-degree and smooth non-algebraic implicit surfaces. This approach has been chosen for the final Applet ▷15.

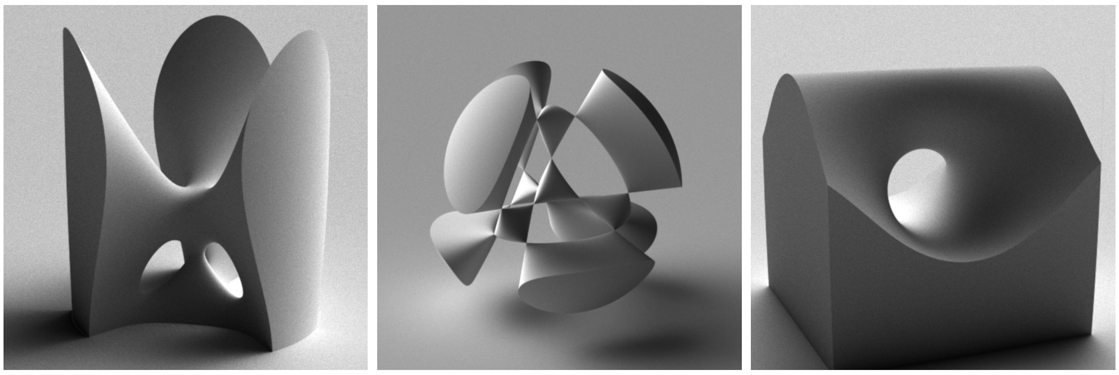
8.4. Monte-Carlo path tracing of surfaces

In the introduction of this chapter, we have posed the question of whether it is possible to generate images as Gerd Fischers photographs of the old models on the computer? His photographs (Figure 8.1) show visually pleasing effects due to the indirect illuminations. A surface patch might appear brighter because it reflects light from another illuminated part of the surface. Can we simulate such a behavior? Instead of tracing a single ray for each pixel (raycasting), we need to trace further back where the light comes from (path tracing).

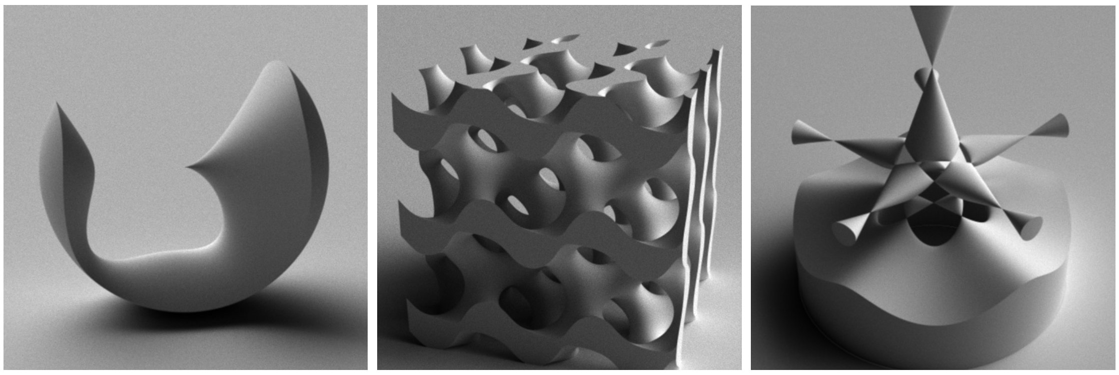
In the preceding sections of this chapter, we focused on raycasting. For each pixel, on the screen, a single intersection of the ray "behind its pixel" with the surface has been computed. We have developed an efficient procedure to intersect rays with surfaces. It turns out that this procedure is also the crucial component in a path tracer. So we have developed a path tracer that utilizes the previously developed surface-ray intersection procedure in Applet ▷17. All photographs depicted in Figure 8.1 have been rendered as Monte-Carlo simulation in Figure 8.11a by using Applet ▷17. One can observe how indirect light spreads through the depicted virtual sculptures.

The solution of the rendering equation (Kajiya, 1986) has been approximated by Monte-Carlo path tracing. We are tracing back the path of light starting from the viewer. Imagined photons are simulated backward iteratively up to a certain depth or until a light source is reached. The obtained brightness values are added successively. We assumed perfectly diffuse surfaces with Lambertian reflectance, i.e., surfaces scatter illumination equally in all directions, and that intensity of radiation obeys Lambert's cosine law (Pharr et al., 2016). The user has to wait some time until the noise in the Monte Carlo rendered scene becomes invisible. It is new to apply this procedure in the browser to render algebraic surfaces of high degree. The implementation only uses backward tracing and produces pleasing images in a fraction of a second. However, to obtain a smooth real-time path tracer for more complex surfaces, techniques such as next event estimation would be needed to be implemented.

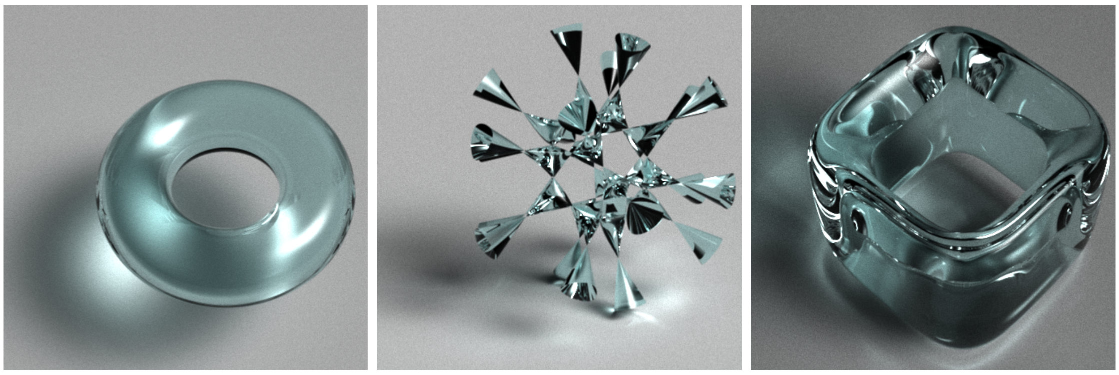
In addition to the diffuse surfaces, in the used path tracing algorithm, it was not much effort to model one side of the algebraic surfaces as being built out of a transparent material. In Applet ▷17, the user can switch the material to glass. At the surface (and a part of the boundary of the clipping sphere), incident light is partly reflected and partly refracted. The refraction of light follows Snell's law. In physical-based rendering, the fraction of reflected and partly refracted light should follow the Fresnel equations, however here the faster approximation of Schlick (1994) has been implemented. One can observe how caustics on the floor became visible in Figure 8.11c.



(a) Monte-carlo simulations of surfaces with Lambertian reflectance: the Clebsch diagonal surface, the Kummer surface and the Parabolic ring cyclide. These are computer-generated versions of the images from Figure 8.1.



(b) Monte-carlo simulations of surfaces with Lambertian reflectance: Vis a Vis ($z^2 + z^3 + y^2 + y^4 + x^3 - x^4 = 0$), a variant of the Gyroid ($\cos(x)\sin(y) + \cos(y)\sin(z) + \cos(z)\sin(x) + \frac{2}{3} = 0$) and the Togliatti Quintic.



(c) Monte-carlo simulations of glass shapes bounded by algebraic surfaces: A torus, the Barth Sextic and the algebraic surface $(x^4 + y^4 - 1)^2 - (2/3 - z^4) = 0$.

Figure 8.11.: Images generated with Applet ▷17, a path tracer for surfaces.

8.5. Analytic landscapes

A further application of the renderer of algebraic and non-algebraic surfaces lies in the visualization of complex functions. Let $f : \mathcal{C} \to \mathcal{C}$ be a meromorphic function. Meromorphic functions are functions that are holomorphic, i.e. conformal, on the entire domain except for a discrete set D of isolated points. The graph of f would live in the four-dimensional real space and is difficult to capture visually. However, instead of visualizing $f : \mathcal{C} \to \mathcal{C}$, the graph of the function $\tilde{f} : \mathbb{R}^2 \setminus D \to \mathbb{R}$ with $\tilde{f}(x, y) = |f(x + i \cdot y)|$ lies in $\mathbb{R}^3$ and can be visualized.

It turns out, if f is rational, i.e. $f = \frac{p}{q}$ for two polynomials $p, q : \mathcal{C} \to \mathcal{C}$ (wlog. p and q have distinct roots) then the analytic landscape $S_f := \{(x, y, |f(x + iy)|) \mid x + iy \in q^{-1}(\mathcal{C}^\times)\}$ is a truncated algebraic surface:

$$\begin{aligned}(x, y, z) \in S_f &\Leftrightarrow z = |f(x + iy)| = \frac{|p(x + iy)|}{|q(x + iy)|}\\ &\Leftrightarrow z^2 \cdot |q(x + iy)|^2 - |p(x + iy)|^2 = 0 \wedge z \geqslant 0\end{aligned}$$

As p and q are polynomials, also their squared norms are polynomials in x and y: $|p(x + iy)|^2 = \operatorname{Re}(p(x + iy))^2 + \operatorname{Im}(p(x + iy))^2$, etc. Thus, the function

$$\begin{aligned}F(x, y, z) &= z^2 \cdot |q(x + iy)|^2 - |p(x + iy)|^2\\ &= z^2 \cdot (\operatorname{Re}^2 q(x + iy) + \operatorname{Im}^2 q(x + iy)) - (\operatorname{Re}^2 p(x + iy) + \operatorname{Im}^2 p(x + iy))\end{aligned}$$

is algebraic with $\deg F = \max\{2 + 2 \deg q, 2 \deg p\}$ and $F^{-1}(\{0\}) \cap (\mathbb{R}^2 \times \mathbb{R}_{\geqslant 0})$ coincides with S_f, the graph of $\tilde{f}$, the analytic landscape for $f : \mathcal{C} \to \mathcal{C}$. In the Chapter before we have introduced an algorithm that can render algebraic surfaces efficiently and accurately enough for visualizations through raycasting. This algorithm can be applied here as well.

The raycasting algorithm requires aside from F, also the gradient $dF : \mathbb{R}^3 \to \mathbb{R}^3$ for shading. The complex derivatives $p' : \mathcal{C} \to \mathcal{C}$ and $q' : \mathcal{C} \to \mathcal{C}$ can be pre-computed through symbolic computation. With the help of the Cauchy-Riemann equations $dF : \mathbb{R}^3 \to \mathbb{R}^3$ can be derived from p' and q' :

$$dF(x, y, z) = 2 \begin{pmatrix} z^2 \cdot (\operatorname{Re} q \operatorname{Re} q' + \operatorname{Im} q \operatorname{Im} q') - (\operatorname{Re} p' \operatorname{Re} p + \operatorname{Im} p' \operatorname{Im} p) \\ z^2 \cdot (-\operatorname{Re} q \operatorname{Im} q' + \operatorname{Im} q \operatorname{Re} q') - (-\operatorname{Re} p' \operatorname{Im} p + \operatorname{Im} p' \operatorname{Re} p) \\ z\,((\operatorname{Re} q)^2 + (\operatorname{Im} q)^2) \end{pmatrix},$$

where the arguments for p, p', q and q' always are $x + iy$.

In Applet ▷18, the algebraic surfaces F are rendered for a user specified rational complex function f (screenshot in Figure 8.12a). In Applet ▷19 (screenshot in Figure 8.12b

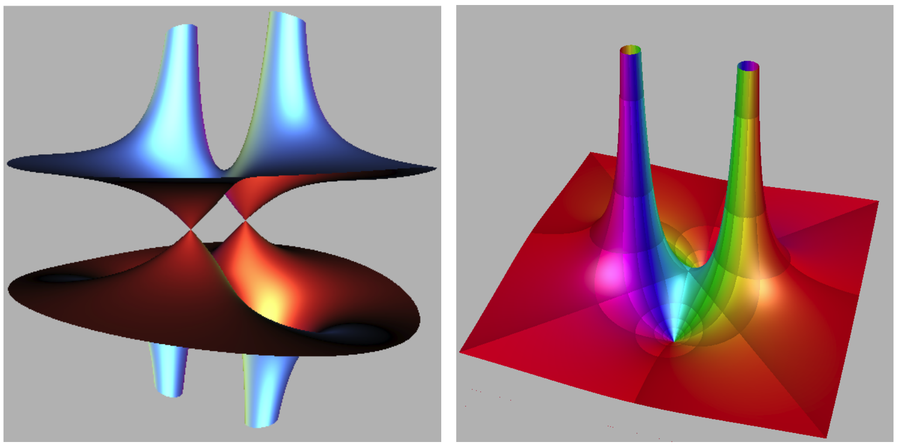


Figure 8.12.: The complex function $f(z) = \frac{z^2+1}{z^2-1}$ with two zeros and two poles induces an algebraic surface of degree 6. (a) The associated algebraic surface $z^2\left(y^4 + \left(2x^2+2\right)y^2 + x^4 - 2x^2 + 1\right) - \left(y^4 - \left(2-2x^2\right)y^2 + x^4 + 2x^2 + 1\right) = 0$ and (b) the truncated algebraic surface that equals to the analytic landscape of f. The color of $(x, y, |f(x+iy)|)$ is assigned based on the phase and modulus of $f(x+iy)$

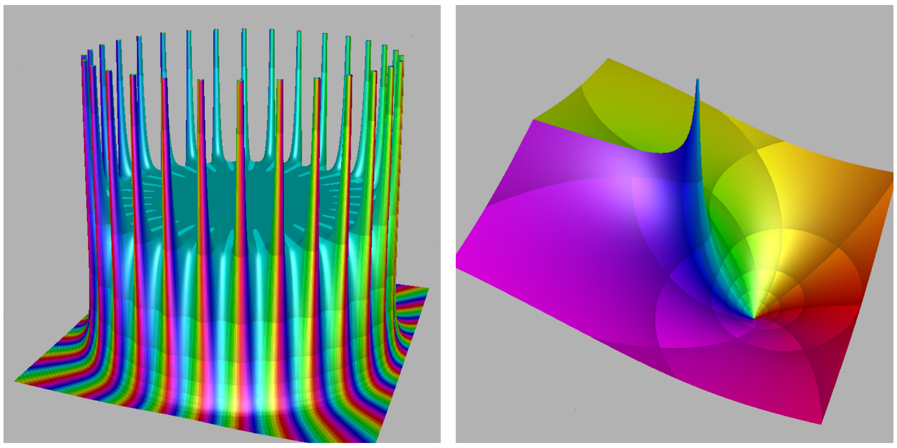


Figure 8.13.: Analytic landscapes for the functions (a) $z \mapsto \frac{1}{z^{30}-1}$, (b) $z \mapsto \log(z)$ rendered with the adaptive interpolation approach.

and Figure 8.13) the landscapes are furthermore clipped such that only patches with non-negative z-value are displayed. Furthermore, the color of each point $(x, y, |f(x+iy)|)$ indicates the phase of the complex number $f(x+iy)$ according to Wegert and Semmler (2010).

The root-finder used in Applet ▷19 uses adaptive interpolation with degree 7 polynomials. With this, surfaces of reasonable high-degree can still be rendered accurately and efficiently while avoiding numeric noise. This makes it possible to render well analytic landscapes for rational functions such as $\frac{1}{z^{30}-1}$, which gives rise to a surface of algebraic degree 62 (comp. Figure 8.13a). Also non-rational complex functions $f : \mathcal{C} \to \mathcal{C}$, which do not give rise to an algebraic surfaces, can be rendered surprisingly well with this approach (comp. Figure 8.13b, Figures 8.14b and 8.14c).

8.5.1. Comparison with photographs of old plaster models of complex functions

In the collection of old plaster models of Schilling (1903) also models that illustrate complex functions have been included. However, at that time it was usual to make two separate sculptures for the real and imaginary part instead of choosing the modulus of the function values as the height and representing the argument with a color. A simple computation shows that a graph of the real and imaginary part of a rational complex function $f = \frac{p}{q}$ is an algebraic surface:

$$\begin{aligned}(x, y, z) \in \{(x, y, \tfrac{\mathrm{Re}}{\mathrm{Im}} f(x+iy)) \mid x+iy \in q^{-1}(\mathcal{C}^\times)\} \Leftrightarrow z = \tfrac{\mathrm{Re}}{\mathrm{Im}} f(x+iy) = \tfrac{\mathrm{Re}}{\mathrm{Im}}\left(\frac{p(x+iy)}{q(x+iy)}\right)\\ \Leftrightarrow z \cdot |q(x+iy)|^2 - \tfrac{\mathrm{Re}}{\mathrm{Im}}\left(p(x+iy) \cdot \overline{q(x+iy)}\right) = 0,\end{aligned}$$

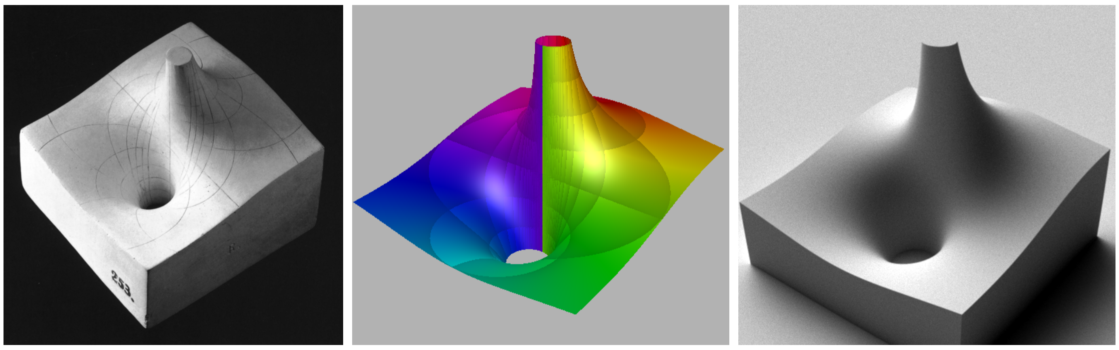
which gives rise to the tri-variate polynomials, whose zero set is the surface:

$$F(x, y, z) = z \cdot (\mathrm{Re}^2\, q(x+iy) + \mathrm{Im}^2\, q(x+iy)) - \tfrac{\mathrm{Re}}{\mathrm{Im}}\left(p(x+iy) \cdot \overline{q(x+iy)}\right).$$

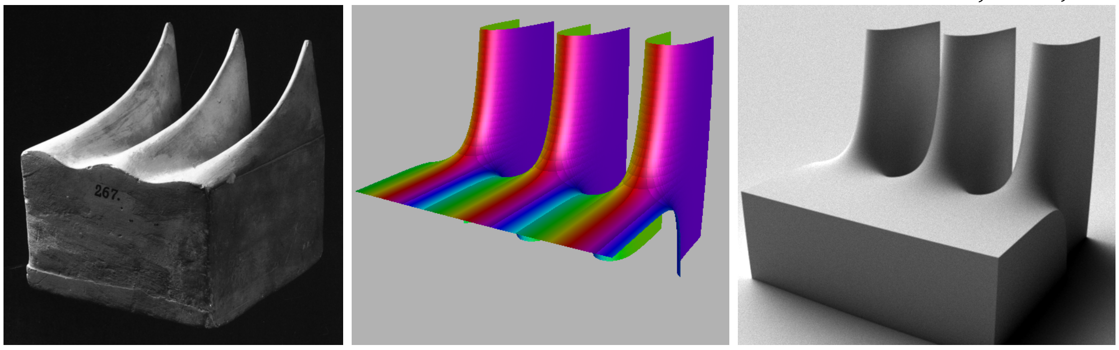
Again, with help of the Cauchy-Riemann equations, dF can be derived.

Options to render only the real or imaginary part of a complex function have been added to Applet ▷19. It also turned out that the internal approximation through low-degree polynomials gives raise to a suitable way to visualize the real and imaginary part for many non-rational meromorphic functions.

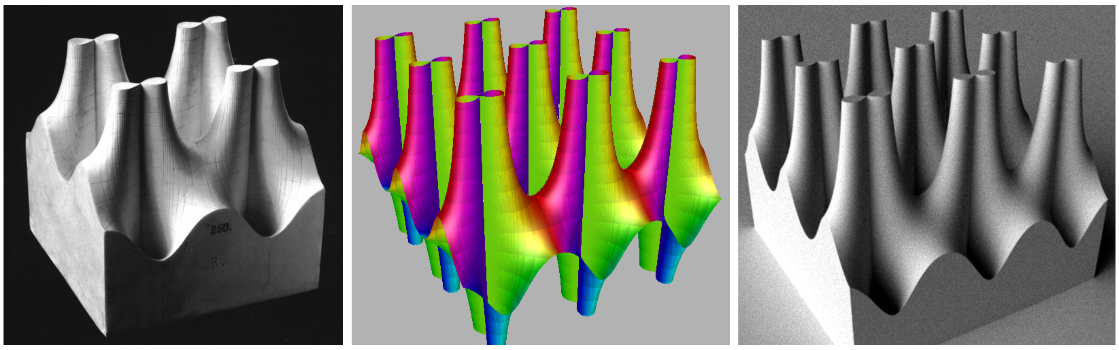
We want to reproduce some of the pictures of the old sculptures, more precisely of the photographs in the first column of Figure 8.14, on the computer. The second column of Figure 8.14 depicts simulations of the images by raycasting.



(a) Real part of the complex function $\frac{1}{z}$, i.e the plot of the function $(x, y) \mapsto \mathrm{Re}(\frac{1}{x+iy}) = \frac{x}{x^2+y^2}$.



(b) Real part of the complex function $\exp(z)$, i.e the graph $(x, y) \mapsto \mathrm{Re}(\exp(x + iy)) = e^x \cdot \cos(y)$.



(c) Graph of real part of a Weierstraß $\wp$ function.

Figure 8.14.: The first column consists of photographs of plaster models that were kindly provided by Gerd Fischer. The second column contains the plot of the same function through raycasting. The graph is colored according to the complex phase of the corresponding function value. The third column consists of renderings generated through Monte-Carlo path tracing of the same graph.

In order to obtain images that are more close to Gerd Fischers' photographs, more physical based rendering is desirable and an option to use Monte-Carlo path tracing to approximate the rendering equation as it has been introduced in Section 8.4 has been added to Applet ▷19. The images can be seen in the third column of Figure 8.14.

The Weierstraß $\wp$ function for the model in Figure 8.14c has been computed by making use of its periodicity. The summation in its "fundamental cell" has been approximated by evaluating a truncated sum of the series of the Weierstaß $\wp$ function, which is a rational function.

Chapter 9.

Deformation of images

Another strength of the presented *CindyGL* framework is its ability to deform images. Programs such as

```
colorplot(
  imagergb(image, f(#))
);
```

compute deformations on the GPU. In Applet ▷20, variants of the given program can be explored. The user can upload images or use the webcam as live-feed input for `image`.

The function $f : \mathbb{R}^2 \to \mathbb{R}^2$ specifies an (inverse) deformation of an image. The color of `image` at position `f(#)` is assigned to the pixel with coordinate `#`. In more mathematical terms, `image` can be considered as a function $\tilde{\chi} : \mathbb{R}^2 \to [0,1]^3$, which assigns a color to each coordinate. The program outputs the image $\tilde{\chi} \circ f : \mathbb{R}^2 \to [0,1]^3$.

These operations are a generalization of deformations. A bijective map specifies a deformation (in the classical sense) of the original image to the deformed image. To compute such a deformation in the given setting, f has to be chosen as the inverse of the deformation function. We also allow non-injective functions f, meaning that two different patches of the “deformed” image can possibly originate from the same position of the original image. Instead of speaking of “deforming images”, it is more precise to say that the original images are *looked through a specified function*. However, in this chapter, we will often use the term “deformation”, since the verb “to deform” is more common in language, although technically speaking it only covers the particular family of injective functions f.

For instance, the function $f(x,y) = (-y,x)$ corresponds to a 90° (clockwise) rotation, but also more complicated functions can be used to create interesting effects (Compare Figure 9.1).

Since CindyGL performs these operations on the GPU, real-time interactivity is possible: Both, the image and the function can change in real-time. Together with the ability

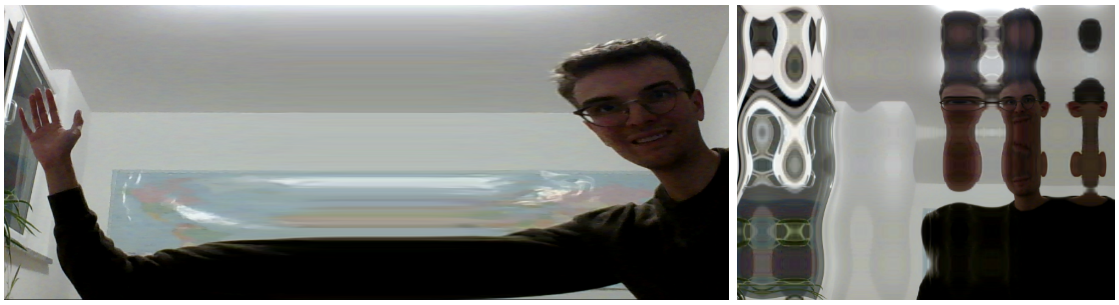


Figure 9.1.: webcam images viewed through the functions $f(x,y) = (x^3, y)$ and $f(x,y) = (x,y) + 0.05 \cdot (\sin(40x), \cos(40y))$, respectively.

of CindyJS to have a webcam as input feed, even the image of a camera is deformed in real time. Deforming a live-image is advantageous in Mathematics education: The interactivity is drastically increased when the user of a program experiments with a picture of himself or has the possibility to interplay with the physical world.

This scheme has many applications. For instance, von Gagern and Mercat (2010) introduce several mathematical image transformations ("filters") that are suitable for the GPU and present applications in the fields of education and arts. The filters include the generation of (hyperbolic) tilings by wallpaper groups and the deformation of images using conformal maps. von Gagern and Mercat note that using live footage could be used to demonstrate mathematical concepts to the public in an appealing and fascinating way and, therefore, could raise people's interest in the underlying mathematical concepts. The filters presented in their work can be implemented as the aforementioned transformation scheme with relatively little effort. Mercat, for instance, has already transferred his *conformal webcam* to *CindyGL*[1].

Richter-Gebert (2017) has developed the tool *iOrnament Crafter*. It is a post-processing tool for the app *iOrnament*. Conform deformations can be applied to the double periodic previously drawn ornaments. Internally, these deformations are modeled through conformal lookup functions specifying where the original ornament is accessed to define the color of the resulting deformation at a given coordinate. In particular, spirals of ornaments can be generated with the help of the complex logarithm. In the next section, we will give mathematical foundations for these deformations and show how these effects can be produced using this scheme together with CindyGL.

Also, Kovács (2019) used CindyGL in this setting in the classroom. He used the inversion at the unit circle as deformation function f and pointed the camera to a blackboard.

[1]See http://bit.ly/webcamconf

Drawing geometric objects such as lines or other circles on the chalkboard, the objects deformed through circle inversion were visible to students instantaneously. Specific properties of the inversion such as the deformation of lines to circles or the preservation of tangents could be demonstrated easily.

Images stored on the computer usually have a bounded rectangular size. What happens if for some position (x, y) on the screen $f(x, y) \in \mathbb{R}^2$ lies outside of the images domain of the image? By default, CindyGL outputs the color black in this case. It is possible to lay out the image periodically in the entire two-dimensional plane by using the modifier `repeat->true` for the `imagergb` command. In mathematical terms, the image could be defined as a function $\chi : \mathbb{R}^2/\langle w_1, w_2\rangle \to [0, 1]^3$ instead of $\tilde{\chi} : \mathbb{R}^2 \to [0, 1]^3$, hence as a map from the torus to the color space, where $\langle w_1, w_2\rangle$ denotes the lattice generated by the vectors $w_1, w_2 \in \mathbb{R}^2$ spanning the fundamental tile of the image. Points in $\mathbb{R}^2$ are identified whenever their difference is a sum of integral multiples of w_1 and w_2.

In Applet ▷20, it is possible to set the `repeat`-modifier to true and explore “deformations” of different *periodically laid out* input images. After having fixed a function $f : \mathbb{R}^2 \to \mathbb{R}^2$, the image defined through the concatenation $\chi \circ \pi \circ f$ is displayed, where $\pi : \mathbb{R}^2 \to \mathbb{R}^2/\langle w_1, w_2\rangle$ denotes the canonical projection in the quotient space.

Two further modifiers for `imagergb`, that are also adjustable in Applet ▷20, are `interpolate` and `mipmap`. The first modifier specifies the behavior of `imagergb` whenever the image is accessed at a position that does not coincide with the middle of the pixel. With `interpolate->false`, the color of the closest pixel is returned, whereas `interpolate->true` causes that, the color is obtained by linear interpolation of the four closest pixels.

Setting `mipmap->true` to the value true, triggers CindyGL to precompute some mipmaps, i.e., some downsampled textures are precomputed to reduce aliasing artifacts. When a color is requested from `imagergb`, then the possibility of the OpenGL Shading model to communicate between neighboring pixels is used: An estimation of the Jacobian of the current deformation is calculated, which then specifies the required level of detail of the texture. Technically, a pixel P can be considered as a square-shaped area on a screen with center (x, y), i.e. $P = [-\frac{\varepsilon}{2}, \frac{\varepsilon}{2}]^2 + (x, y)$. The statement `imagergb(image, f(x,y), mipmap->true)` gives a very rough heuristic for the color

$$\frac{1}{\varepsilon^2} \int_{p\in[-\frac{\varepsilon}{2},\frac{\varepsilon}{2}]} \tilde{\chi}(f((x, y) + p))dp \approx \frac{1}{\varepsilon^2} \int_{p\in[-\frac{\varepsilon}{2},\frac{\varepsilon}{2}]} \tilde{\chi}(f(x, y) - J_f(x, y)(p))dp,$$

where the integral on the right hand side is approximated by a lookup in the mipmap-texture based on the determinant of J_f.

High-frequency effects arising in sparsely sampled patches of the accessed texture

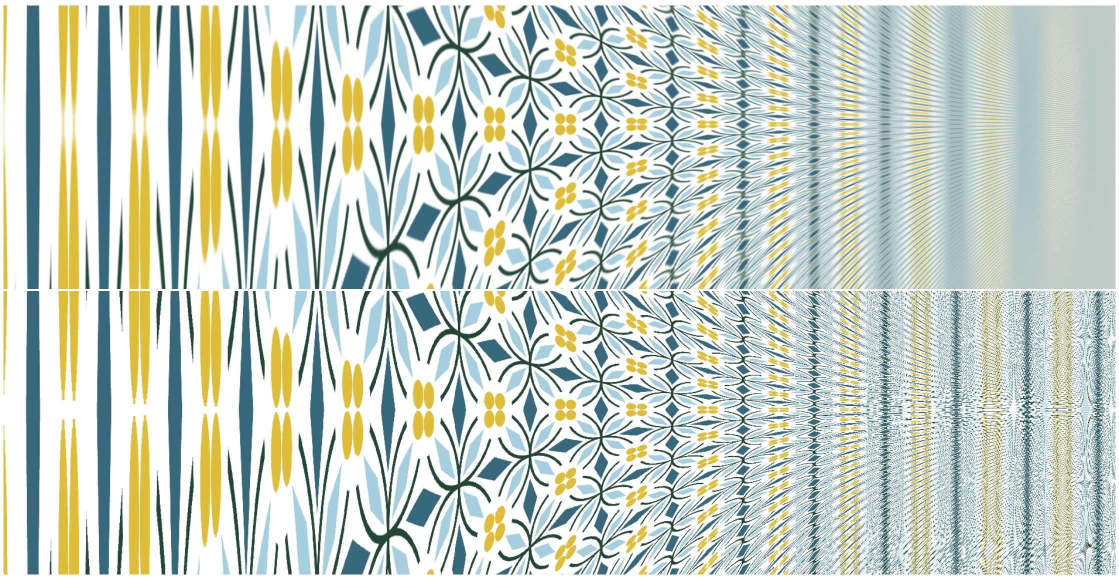


Figure 9.2.: A deformation of a regular pattern induced by $f(x,y) = (x, y\cdot\exp(2x))$ rendered once with the modifiers `interpolate` and `mipmap` set to true (above) and once without (below).

are reduced (compare the right sides of figure Fig. 9.2).

9.1. Holomorphic functions

Particular nice looking (inverse) deformations $f : \mathbb{R}^2 \to \mathbb{R}^2$ are those which preserve the angles (including their orientation), i.e. *conformal* deformations. If we identify $\mathbb{R}^2$ with the complex numbers $\mathbb{C}$, $f : \mathbb{R}^2 \supseteq U \to \mathbb{R}^2$ is conformal, if and only if $f : \mathbb{C} \supseteq U \to \mathbb{C}$ is *holomorphic* and the derivative of f vanishes nowhere in U (Olver, 2015).

Most functions from complex analysis, such as compositions of polynomials, rational functions, trigonometric functions, the exponential map, the logarithm, and the complex square root, are *holomorphic almost everywhere* (everywhere but on a set with vanishing measure they are locally holomorphic). Furthermore, if they are non-constant, the set of points with vanishing derivative is discrete in their domain. Thus, they are conformal almost everywhere.

Maps in one complex variable have a methodological advantage: Many of them are easy to specify if a set of elementary functions such as the complex trigonometric functions are provided.

In Applet ▷21, a function $f : U \to \mathbb{C}$ in one complex variable z can be specified. An

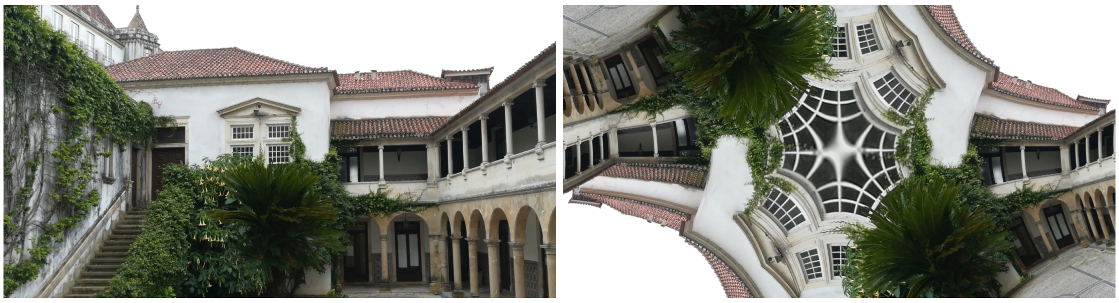


Figure 9.3.: (a) An image from Coimbra, (b) the same image watched through the conformal function $f(z) = z^2$.

image using the following program is rendered:

```
colorplot(
  imagergb(image, f(z), repeat->true)
);
```

Thereby, to each pixel on the screen a complex number z is associated. The pixel at the position z on the screen gets the color of the periodically laid out image at the position $f(z)$. Different periodic tiles, as well as the webcam image can be selected as input. In formulas, for a given map f, the image $\chi \circ \pi \circ f$ is displayed where the periodic tile with width $w_1 \in \mathbb{R}_{>0}$ and height $\frac{1}{i} w_2 \in \mathbb{R}_{>0}$ is specified through $\chi : \mathbb{C}/\langle w_1, w_2 \rangle \to [0,1]^3$. Again, $\pi : \mathbb{C} \to \mathbb{C}/\langle w_1, w_2 \rangle$ denotes the canonical quotient map.

The idea to compute images or the webcam looked through a holomorphic maps is not new (Mercat, 2009; de Smit et al., 2012; Mercat, 2015).

Several concepts of complex analysis can be demonstrated and explained by this framework. Let $g : \mathbb{C} \to \mathbb{C}$ be a (bijective) transformation. Then, with $f = g^{-1}$, the transferred image is rendered. In Applet ▷21, draggable complex variables $a, b \in \mathbb{C}$ can be used within the definition of f. How do complex numbers act on themselves through addition and multiplication? It becomes visible that the action $g(z) = z + a$ ($f(z) = z - a$) induces a translation along the vector a whereas the multiplication $g(z) = a \cdot z$ ($f(z) = z/a$) causes a mixture of stretching and rotation around 0, moving the point 1 to a. The inversion at the unit-circle can be entered through the anti-conformal map $g(z) = f(z) = \frac{1}{\bar{z}}$. This has been studied in a classroom by Kovács (2019).

A particular nice non-injective example is the squaring function $f(z) = z^2$ (see Figure 9.3 and Figure 9.4b). The entire complex plane except for zero is doubled in a conformal way. Mercat (2009); de Smit et al. (2012) have already studied this image effect induced by the complex squaring function. The framework CindyGL can apply it

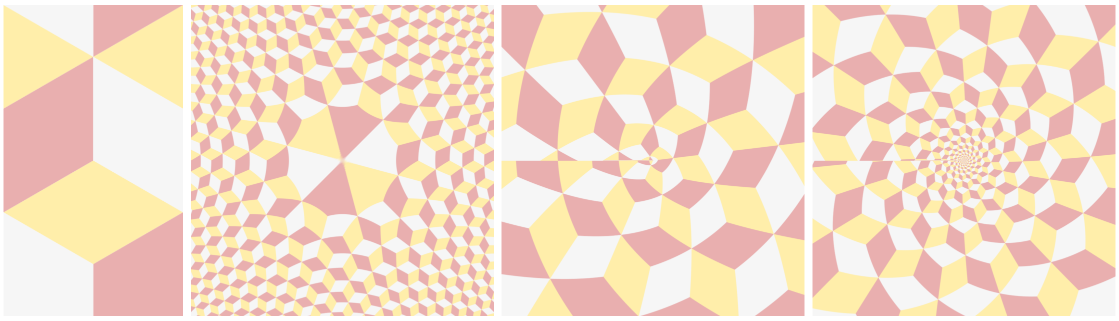


Figure 9.4.: The image $\chi \circ \pi \circ f$ for different functions f, where $\pi : \mathbb{C} \to \mathbb{C}/\langle 1, i\sqrt{3}\rangle$ denotes the canonical quotient map. The basic tile $\chi : \mathbb{C}/\langle 1, i\sqrt{3}\rangle \to [0,1]^3$ is depicted in (a). In (b), the complex squaring function $f_b(z) = z^2$ has been used. Subfigures (c) and (d) show visible branch-cuts along $\mathbb{R}_{<0}$ when using the complex logarithm or square root to define $f_c(z) = \sqrt{z} \cdot (2+i)$ or $f_d(z) = \operatorname{Log}(z) \cdot (2+i)$, respectively. The factor $(2+i)$ has been added to make the branch cut more apparent.

to a webcam and create interesting effects. If the webcam points to the screen depicturing the live feed image watched through $f(z) = z^2$, we generate images of Julia-sets as described in Section 4.3.2 (in particular, observe Figure 4.2a).

Can the set of functions that render locally conformal deformations of the original image be extended? On the contrary, if non-holomorphic functions, such as `Log(z)` or `sqrt(z)`, are entered, a sharp line left of zero becomes visible (compare Figure 9.4c and d) disturbing the integrity of the image. The visible lines arise from the branch cuts of the implemented functions.

Let $S = \{z \in \mathbb{C} \mid \operatorname{im}(z) \in (-\pi, \pi]\}$ be a horizontal strip in the complex plane. Then the complex exponential function defines a bijection between S and $\mathbb{C}^\times$. The logarithm in CindyJS is implemented as the inverse function of $\exp : S \to \mathbb{C}^\times$, i.e., as $\operatorname{Log} : \mathbb{C}^\times \to S$, the principal branch of the complex logarithm. The imaginary part of $\operatorname{Log}(z)$ with $z = e^{i\phi}e^{|z|}$ is defined to be $\phi \in (-\pi, \pi]$. The map $\operatorname{Log} : \mathbb{C}^\times \to S$ cannot be continuous: Consider the complex variable z moving along a path in $\mathbb{C}^\times$ crossing the ray of the negative real numbers from the upper half-plane to the lower halfplane. When z approaches the ray from above, the imaginary part of $\operatorname{Log}(z)$ approaches π. However, if z comes from below, then the imaginary part of $\operatorname{Log}(z)$ approaches $-\pi$. Thus, the imaginary part of $\operatorname{Log}(z)$ is forced to jump by 2π when z is crossing the negative real numbers. The complex square root in CindyJS is implemented as the function $z \mapsto \exp(\frac{1}{2}\operatorname{Log}(z))$, and the visible at the negative real axis origins from the branch-cut of the Log. These branch cuts can also be observed in Applet ▷19 by rendering the graph of the real and imaginary part.

Can we still generate images that use the branched complex logarithm without a visible cut? One way to make the logarithm continuous (and even holomorphic) is to replace it with a multi-valued function. This can also be considered as replacing the co-domain of the complex logarithm by $\mathbb{C}/\langle 2\pi i\rangle$, where $\langle 2\pi i\rangle$ denotes the subgroup of $(\mathbb{C}, +)$ that is generated by $2\pi i$, i.e. two elements in $\mathbb{C}$ are identified whenever their difference is an integer multiple of $2\pi i$. Since the group of translations $\langle 2\pi i\rangle$ acts proper and discontinuously on $\mathbb{C}$, the space $\mathbb{C}/\langle 2\pi i\rangle$ can be considered as a Riemannian manifold and the map $\log : \mathbb{C}^\times \to \mathbb{C}/\langle 2\pi i\rangle$ as conformal.

In order to generate images with the branched complex logarithm avoiding visible cuts, one can make use of the periodicity of the image. Let the vectors $w_1, w_2 \in \mathbb{C}$ span the dimensions of the fundamental tile of the periodically laid out image. The image $\chi : \mathbb{C}/\langle w_1, w_2\rangle \to [0,1]^3$ has a periodicity for every $k \in \mathbb{Z}^2 \setminus \{0\}$:

$$\forall z \in \mathbb{C} : \chi(z + k_1 w_1 + k_2 w_2) = \chi(z).$$

The branch-cut of the logarithm can be made invisible if the jumps by the value $2\pi i$ are aligned with an arbitrarily chosen period of the image:

Lemma 70. *Let $\chi : \mathbb{C}/\langle w_1, w_2\rangle \to [0,1]^3$ be an image with periods w_1 and w_2 and let $\pi : \mathbb{C} \to \mathbb{C}/\langle w_1, w_2\rangle$ be the canonical quotient map.*

Any meromorphic $h : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ together with a $k \in \mathbb{Z}^2 \setminus \{0\}$ give rise to a (potentially branched) function $L : U \to \mathbb{C}$ via

$$L(z) = \frac{k_1 w_1 + k_2 w_2}{2\pi i} \operatorname{Log}(h(z))$$

such that

- *$\pi \circ L : U \to \mathbb{C}/\langle w_1, w_2\rangle$ is conformal on $U = \mathbb{C} \setminus \{h^{-1}(0), h^{-1}(\infty)\}$, and, thus,*
- *the image $\chi \circ \pi \circ L : U \to [0,1]^3$, which is defined almost everywhere up to an isolated discrete set, does not have any visible branch-cuts.*

Proof. Let us outline a proof to these properties. For the scaled rotation

$$\lambda : \mathbb{C} \to \mathbb{C}, \quad z \mapsto \frac{k_1 w_1 + k_2 w_2}{2\pi i} z,$$

we introduce its lift between quotient spaces as $\tilde{\lambda} : \mathbb{C}/\langle 2\pi i\rangle \to \mathbb{C}/\langle k_1 w_1 + k_2 w_2\rangle$ and the quotient map $\tilde{\pi} : \mathbb{C}/\langle k_1 w_1 + k_2 w_2\rangle \to \mathbb{C}/\langle w_1, w_2\rangle$ (note that $\langle w_1, w_2\rangle$ is a subgroup of $\langle k_1 w_1 + k_2 w_2\rangle$). Then the following composition

$$U \xrightarrow{h|_U} \mathbb{C}^\times \xrightarrow{\log} \mathbb{C}/\langle 2\pi i\rangle \xrightarrow{\tilde{\lambda}} \mathbb{C}/\langle k_1 w_1 + k_2 w_2\rangle \xrightarrow{\tilde{\pi}} \mathbb{C}/\langle w_1, w_2\rangle$$

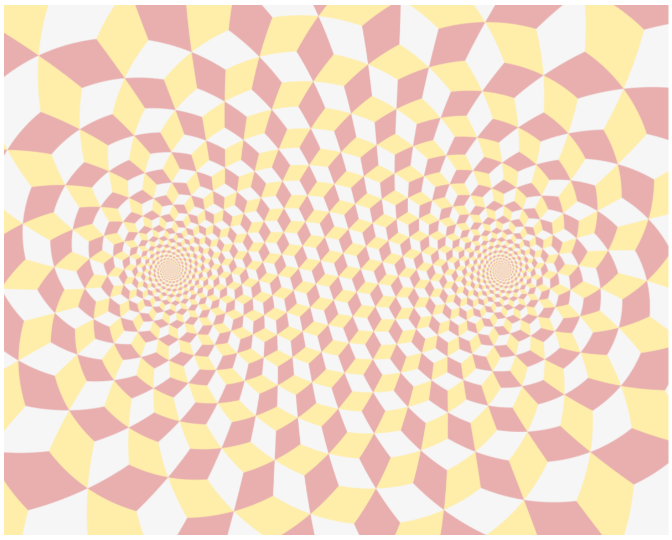


Figure 9.5.: The image $\chi \circ \pi \circ L$ with $L(z) = \frac{7+12i\sqrt{3}}{2\pi i}\operatorname{Log}\left(\frac{z+1}{z-1}\right)$ where the fundamental tile χ has the periodicities 1 and $i\sqrt{3}$.

consists of holomorphic functions between Riemann surfaces only. The entire composition as a function is equal to the function $\pi \circ L$, i.e. to the composition

$$U \xrightarrow{h|_U} \mathbb{C}^\times \xrightarrow{\operatorname{Log}} \mathbb{C} \xrightarrow{\lambda} \mathbb{C} \xrightarrow{\pi} \mathbb{C}/\langle w_1, w_2\rangle.$$

Since the first sequence as composition of holomorphic functions is holomorphic by itself, also $\pi \circ L$ is holomorphic.

The isolated set of zeros and poles of $h : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ determine the only points where $\pi \circ L$ is not locally conformal. The function $\pi \circ L : U \to \chi : \mathbb{C}/\langle w_1, w_2\rangle \to [0,1]$ is holomorphic. In particular it is continuous. If this continuous function is plugged into $\chi : \mathbb{C}/\langle w_1, w_2\rangle \to [0,1]^3$ then the branch cut will become invisible in the image $\pi \circ L \circ \chi$. $\square$

Figure 9.5 has been rendered using a function constructed according to Lemma 70. The two periodicities 1 and $\sqrt{3}i$ of the base tile have been taken into consideration and no branch-cut is visible anymore.

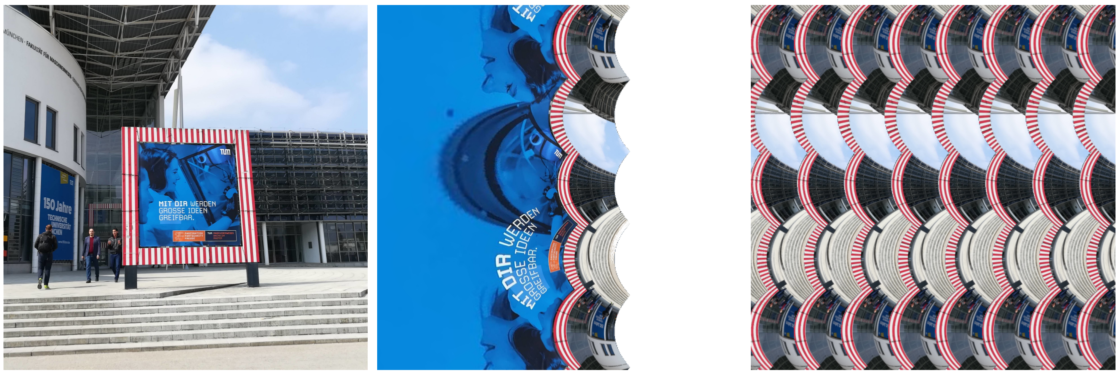


Figure 9.6.: (a) An image $\chi : \mathbb{C} \supseteq D \to [0,1]^3$, (b) its "polar" form $\chi_{\text{polar}} = \chi \circ \exp$, and (c) the image $\tilde{\chi}_{\text{polar}}$ build from χ_{polar} by artificially adding repetition in real-direction.

9.2. The Droste effect

The so-called *Droste Effect* is an effect of an image recursively containing itself (Figure 9.7a serves as a good example). This section adapts and extends the considerations presented in de Smit and Lenstra (2003) and Schleimer and Segerman (2016) such that it fits to the context of the previous section.

Let an image $\chi : D \to [0,1]^3$ with $D \subset \hat{\mathbb{C}}$ be given. We consciously allow the image to be potentially defined on the entire Riemann sphere $\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$, allowing for applying the following on *spherical* footage. We assume that $S \subset D \cap \mathbb{C}$ is a compact set containing 0 in its interior such that $\chi|_S$ depicts an object that is supposed to be cut out. This could be the contents of a frame for instance. We recursively replace $\chi|_S$ by a smaller conformal copy of a part of the image χ. In Figure 9.6a, for example, S could be the area of the blue banner within the red-white striped frame where $D \subset \mathbb{C}$ is the entire rectangular domain of the image.

Every cutting and pasting step should be conformal to make use of the visually pleasing properties of angle preservation. The final image should be a conformal deformation of the original image in every pixel, except for the cutting positions and the limit-point 0.

We consider the image χ in log-exp conjugated coordinates, which can be seen as a conformal variant of polar coordinates. In our setting, this corresponds to the image

$$\chi_{\text{polar}} = \chi \circ \exp .$$

The image χ_{polar}, an example of which has been generated in Figure 9.6b, already has periodicity $2\pi i$ because $\exp(z + 2\pi i) = \exp(z)$. We will denote this by interpreting

$\exp : \mathbb{C}/\langle 2\pi i\rangle \to \mathbb{C}$ and $\chi_{\text{polar}} = \chi \circ \exp : (\exp^{-1} D)/\langle 2\pi i\rangle \to [0,1]^3$. The Droste effect can be obtained if a second artificial period is constructed for χ_{polar}. In the following, we construct such a periodicity in real-direction based on the given set S, compare Figure 9.6c. The Droste effect has one free parameter, specifying the down scaling factor of an image. We will encode this by a sufficiently small parameter $\alpha > 0$ such that

$$\exp(\alpha) \cdot S \subset D. \tag{9.1}$$

still holds.

Since S contains $0 = \lim_{k\to-\infty} \exp(z + k\alpha)$ in its interior and $\infty = \lim_{k\to\infty} \exp(z + k\alpha)$ is contained in the open set $\hat{\mathbb{C}} \setminus S$, the discrete set

$$\{k \in \mathbb{Z} \mid \exp(z + k\alpha) \notin S\} \subset \mathbb{R}$$

attains a minimum for every $z \in \mathbb{C}$. Furthermore, the following assignment does not depend on the choice of a representative $z \in [z] \in \mathbb{C}/\langle \alpha, 2\pi i\rangle$ up to a multiple of $2\pi i$:

$$\textsc{AddPeriod}_S^\alpha(z) = z + \alpha \min\{k \in \mathbb{Z} \mid \exp(z + k\alpha) \notin S\}.$$

This gives rise to a well-defined “periodifying” operator

$$\textsc{AddPeriod}_S^\alpha : \mathbb{C}/\langle \alpha, 2\pi i\rangle \to \mathbb{C}/\langle 2\pi i\rangle$$

and to the tiling

$$\tilde{\chi}_{\text{polar}} = \chi_{\text{polar}} \circ \textsc{AddPeriod}_S^\alpha = \chi \circ \exp \circ \textsc{AddPeriod}_S^\alpha,$$

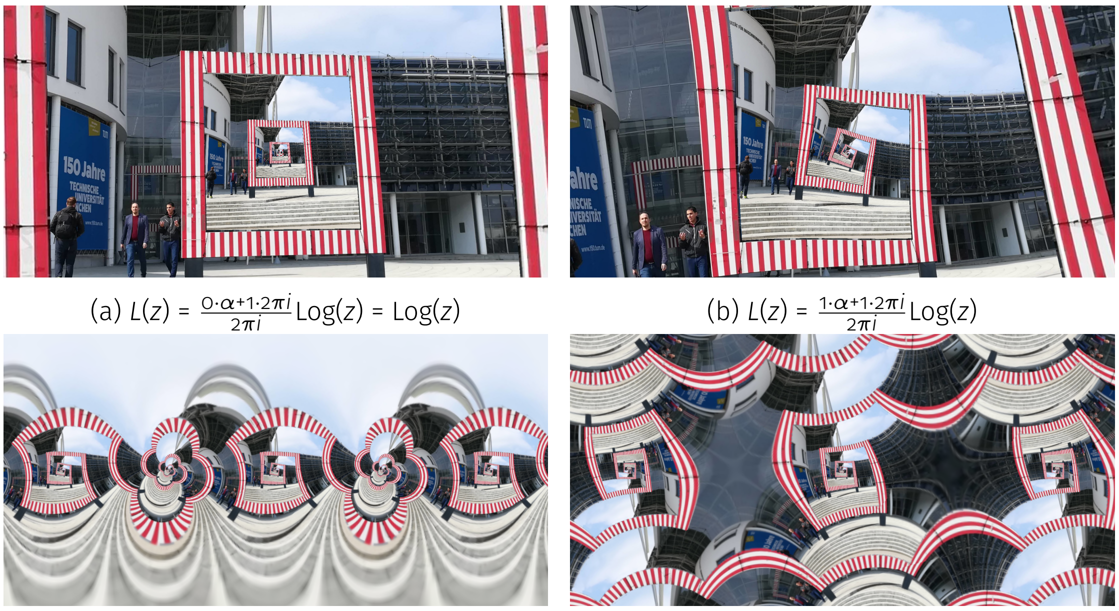
having both the periods $2\pi i$ and α.

Note that Equation (9.1) yields $(\exp \circ \textsc{AddPeriod}_S^\alpha)^{-1}(S) = \varnothing$ and, thus, $\tilde{\chi}_{\text{polar}}$ does not depict the region $\chi|_S$ anymore that was supposed to be cut out. Also, α has been chosen sufficiently small to avoid accessing the original picture at an undefined position.

For the back-conversion of the tiling $\tilde{\chi}_{\text{polar}} : \mathbb{C}/\langle \alpha, 2\pi i\rangle \to [0,1]^3$ from the “conformal polar coordinates” to the “normal coordinates”, it is desirable to use the complex logarithm. However, it is important to avoid visible branch-cuts. At this point, Lemma 70 gives a wide class of potential interesting images: Any complex function L chosen according to Lemma 70 can generate the branch-cut free image

$$\tilde{\chi} = \tilde{\chi}_{\text{polar}} \circ \pi \circ L = \chi \circ \exp \circ \textsc{AddPeriod}_S^\alpha \circ \pi \circ L.$$

This function is a version of the original image χ to which the Droste effect has been applied. For instance, the direct back-conversation via $L(z) = \frac{0\cdot\alpha + 1\cdot 2\pi i}{2\pi i}\operatorname{Log}(z) = \operatorname{Log}(z)$ reversing the previous exponentiation gives rise to an image $\tilde{\chi}$ with a straight Droste



(a) $L(z) = \frac{0\cdot\alpha+1\cdot 2\pi i}{2\pi i}\operatorname{Log}(z) = \operatorname{Log}(z)$

(b) $L(z) = \frac{1\cdot\alpha+1\cdot 2\pi i}{2\pi i}\operatorname{Log}(z)$

(c) $L(z) = \frac{\alpha+2\pi i}{2\pi i}\operatorname{Log}(2\tan(z))$

(d) $L(z) = \frac{\alpha+2\pi i}{2\pi i}\operatorname{Log}(4\sin(z))$

Figure 9.7.: Lemma 70 gives rise to complex functions L such that the images $\tilde{\chi} = \tilde{\chi}_{\text{polar}} \circ \pi \circ L$ show the Droste effect (in classical, twisted, and exotic variants) without visible branch cuts, even though L has branch cuts. The function $\tilde{\chi}_{\text{polar}} : \mathbb{C}/\langle\alpha, 2\pi i\rangle \to [0,1]^3$ is the same as was used for Figure 9.6.

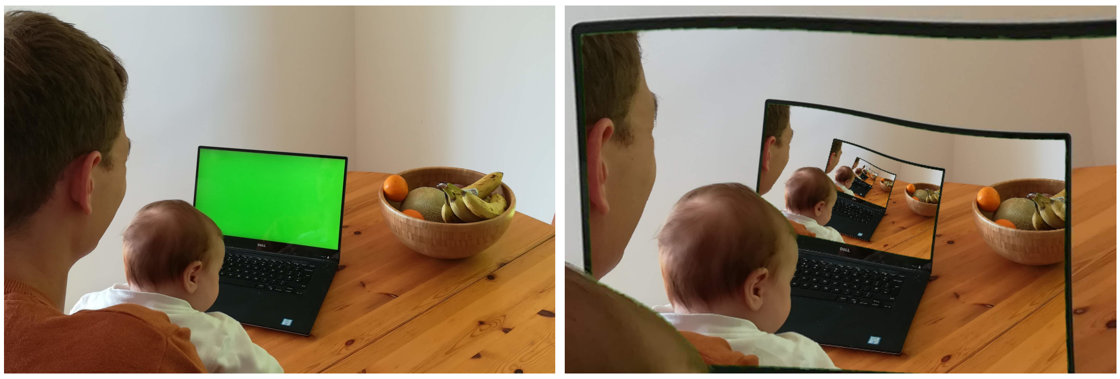


Figure 9.8.: (a) Original image, (b) Twisted Droste effect applied to the green area using Applet ▷22. Droste area selected by chroma keying.

effect (see Figure 9.7a). If $L(z) = \frac{1\cdot\alpha+1\cdot 2\pi i}{2\pi i}\mathrm{Log}(z)$ is used, then the twisted Droste-effect appears (see Figure 9.7b). The effect occurring in M. C. Escher's famous lithograph Print Gallery is in fact of this kind and has been studied from a mathematical point of view by de Smit and Lenstra (2003). Other parameters and the use of further mereomorphic functions give rise to more exotic twisted effects, for instance see Figures 9.7c and 9.7d.

The tiling $\tilde{\chi}_{\text{polar}} : \mathbb{C}/\langle\alpha, 2\pi i\rangle \to [0,1]^3$ used in Figure 9.6 can be applied as input image within Applet ▷21 by computing $\tilde{\chi} = \tilde{\chi}_{\text{polar}} \circ L$. To obtain sharper images, all these operators within

$$\tilde{\chi} = \chi \circ \exp \circ \textsc{AddPeriod}_S^\alpha \circ \pi \circ L$$

can be approximated in real-time and the need for rasterization of an intermediate image is therefore eliminated. In Applet ▷22, images are interactively "drostified" by specifying an object $S \subset \mathbb{C}$. It is also possible to detect the object as a green screen (see Figure 9.8) or to compute the Droste Effect on the image captured by the webcam in real time.

9.3. Spherical image processing

One further possible application of CindyGL is image processing on the sphere. Spherical images and footage can be obtained from spherical cameras.

As already mentioned before, the framework CindyJS has the ability to read live footage from the webcam as input. For our experiments the spherical camera *Ricoh Theta S* has been used since it can be connected to a computer as an external webcam and therefore it is suitable for live interactions with CindyJS.

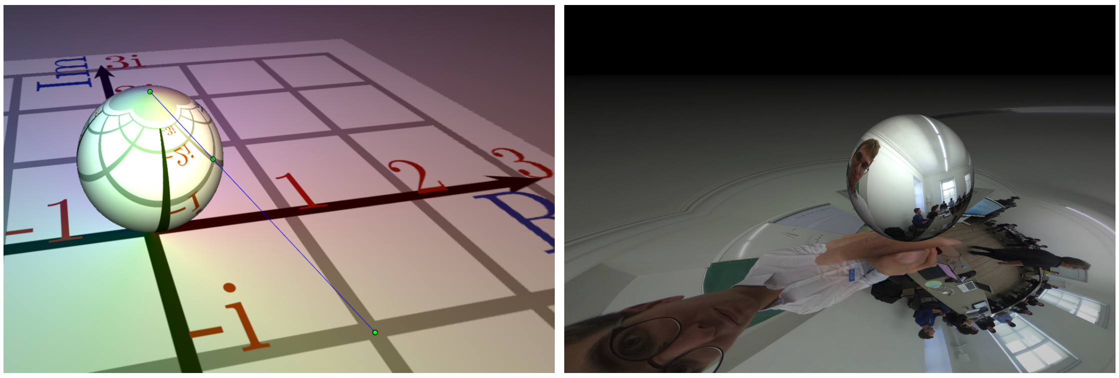


Figure 9.9.: Screenshots of Applet ▷23. The complex plane in Definition 72 "crosses the equator" of the sphere. For improved visibility, the plane has been shifted in the applet, inducing a scaling on the complex plane. Subfigure (a) depicts the Riemann sphere and the stereographic projection, and (b) spherical footage projected on the complex plane.

A spherical image can be considered as a function that assigns a color to each point on the sphere S^2. Commonly, a spherical image or video is stored as a conventional image or video in *equirectangular projection*. The two-dimensional coordinate space of the pixels directly transfers into longitude and latitude of the points on the S^2:

Definition 71 (equirectangular parametrization of the sphere). Any point on the sphere S^2 except for the poles can be described by a unique *latitude* $\phi \in (-\frac{\pi}{2}, \frac{\pi}{2})$ and longitude $\lambda \in (-\pi, \pi]$. Let $D = (-\frac{\pi}{2}, \frac{\pi}{2}) \times (-\pi, \pi] \cup \{-\pi, \pi\} \times \{0\}$. With $\operatorname{atan2}(y, x) \in (-\pi, \pi]$ we denote the angle of the vector $(x, y) \in \mathbb{R}^2 \setminus \{0, 0\}$ and set $\operatorname{atan2}(0, 0) := 0$. The bijective map $\rho : S^2 \to D$ is defined as follows:

$$\rho(x, y, z) = \begin{pmatrix} \arcsin(z) \\ \operatorname{atan2}(y, x) \end{pmatrix}, \qquad \rho^{-1}(\phi, \lambda) = \begin{pmatrix} \cos(\phi)\cos(\lambda) \\ \cos(\phi)\sin(\lambda) \\ \sin(\phi) \end{pmatrix}.$$

In this representation, idealized[2] spherical images are considered as functions $\tilde{\chi} : D \to [0, 1]^3$. The spherical image corresponds to the image $\tilde{\chi} \circ \rho : S^2 \to [0, 1]^3$.

9.3.1. Spherical images as functions on the Riemann sphere

For technical purposes we mapped an image in equirectangular parametrization to the sphere. However, for mathematical purposes it turned out to be more fruitful to

[2]In practice, only a finite set of pixels is stored and the special cases for the poles of the sphere are simply ignored.

follow the approach of Schleimer and Segerman (2016) and, subsequently, to encode the coordinate-space of a spherical image on the Riemann sphere $\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$. The process behind the re-formalization is the stereographic projection from S^2 to $\hat{\mathbb{C}}$.

Definition 72 (The stereographic projection from S^2 to $\hat{\mathbb{C}}$). The stereographic projection from S^2 to $\hat{\mathbb{C}}$ is a bijective map $\sigma : S^2 \to \hat{\mathbb{C}}$ with

$$\sigma(z, y) = \begin{cases} \frac{z}{1-y} & \text{if } (z, y) \neq (0, 1) \\ \infty & \text{if } (z, y) = (0, 1) \end{cases}, \qquad \sigma^{-1}(z) = \begin{cases} \left(\frac{2z}{|z|^2+1}, \frac{|z|^2-1}{|z|^2+1}\right) & \text{if } z \in \mathbb{C} \\ (0, 1) & \text{if } z = \infty \end{cases}.$$

Instead of considering a spherical image as a function $\tilde{\chi} : D \to [0,1]^3$ and performing computations on $\tilde{\chi}$ or $\tilde{\chi} \circ \rho : S^2 \to [0,1]^3$, we will consider it as a function $\chi : \hat{\mathbb{C}} \to [0,1]^3$ through $\chi = \tilde{\chi} \circ \rho \circ \sigma^{-1}$. This connection is visualized in Applet ▷23 and the screenshots in Figure 9.9. The raycasting of the scene including the plane and the sphere has been done utilizing techniques described in Chapter 8.

9.3.2. Pulling back analytic functions

Given a transformation $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ and an image $\chi : \hat{\mathbb{C}} \to [0,1]^3$, Schleimer and Segerman (2016) introduce the term *pullback* of f for the image $\chi \circ f$. Since the stereographic projection is conformal, images generated as pullback of conformal functions $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ give rise to conformal deformations of images on the sphere. For instance, the map $z \mapsto \frac{1}{z}$ induces a rotation turning the sphere upside-down. More generally, the Möbius transformations $z \mapsto \frac{z\alpha+\beta}{z\gamma+\delta}$ with $\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \in \mathrm{SU}(2)$ correspond to rotations of the sphere (Richter-Gebert and Orendt, 2009). The non-bijective example $z \mapsto z^2$ as pullback function produces a branch-point at the south- and north-pole of the Riemann sphere.

How to visualize the concept of pullback functions $\chi \circ f : \hat{\mathbb{C}} \to [0,1]^3$ on a two-dimensional screen of finite area?

In Applet ▷23, it can be studied how particular maps on $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ are pulled back to deformations of a sphere (depicted in Figure 9.10a).

In Applet ▷24, the user is positioned inside the sphere and can observe how the texture is changing for specified transformations $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ (see Figure 9.10b).

What kind of computation is done within Applet ▷24? Drag moves of the mouse are accumulated in a Möbius transformation USERROTATION : $\hat{\mathbb{C}} \to \hat{\mathbb{C}}$. Internal, a SU(2)-matrix containing the coefficients of the Möbius-transformation USERROTATION : $\hat{\mathbb{C}} \to \hat{\mathbb{C}}$ is iteratively multiplied with new SU(2)-matrices corresponding to the rotations induced by each of drag events made by the user. The matrix products carry over into the composition of the corresponding Möbius transformations (Richter-Gebert and Orendt,

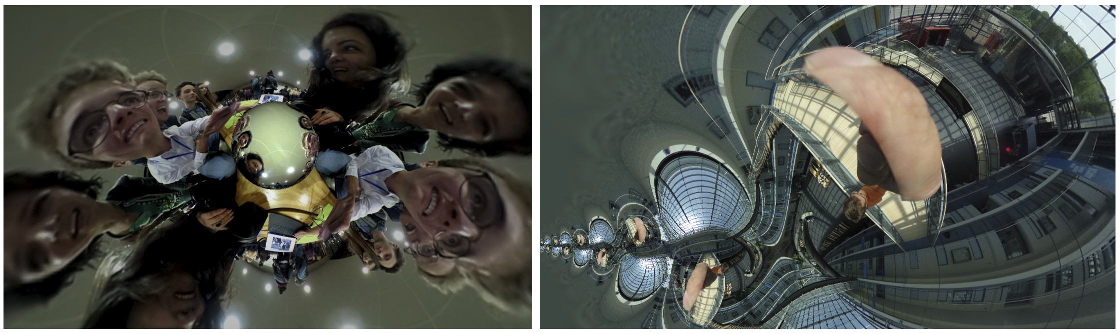


Figure 9.10.: (a) A pullback of the function $z \mapsto z^2$ generates two branch-points at the poles produced within Applet ▷23. (b) A pullback of the function $z \mapsto \sin(z)$ viewed from inside the Riemann sphere in Applet ▷24.

2009). Let $R \subset \mathbb{C}$ denote the rectangular area of the visible screen, $\textsc{Screen} : R \to \hat{\mathbb{C}}$ be the inclusion map and $\tilde{\chi} : D \to [0,1]^3$ be the camera live-feed, which can be accessed within CindyJS as a webcam. Applet ▷24, which aims to visualize the spherical image $\chi \circ f$, computes and displays the map

$$\tilde{\chi} \circ \rho \circ \sigma^{-1} \circ f \circ \textsc{UserRotation} \circ \textsc{Screen},$$

where $\tilde{\chi}$ is the real-time image from a spherical camera in equirectangular coordinates.

9.3.3. The spherical Droste effect

The deformations of Section 9.2 generating the Droste effect in Section 9.2 can be also pulled back to spherical images when considered as functions $\chi : \hat{\mathbb{C}} \to [0,1]^3$. The transformations induced through the "cut-and-paste process" can be interpreted as locally defined Möbius transformations. Viewing the Droste effect on the Riemann sphere gives the possibility to "look-back".

In the formulation of Section 9.2, we generate Droste Effects with the center $0 \in S$. What if an object S is to be cut out such that $0 \notin S \subset \hat{\mathbb{C}}$? A point from the interior of S can be shifted to 0 by previously applying a corresponding SU(2)-Möbius transformation $\textsc{MoveCenter} : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ and the set S can be pulled back through the same deformation.

Within Applet ▷22, it is possible to produce the spherical Droste effect in real-time (A screenshot is in Figure 9.11). For a camera live-feed $\tilde{\chi} : C \to [0,1]^3$ and a choosen

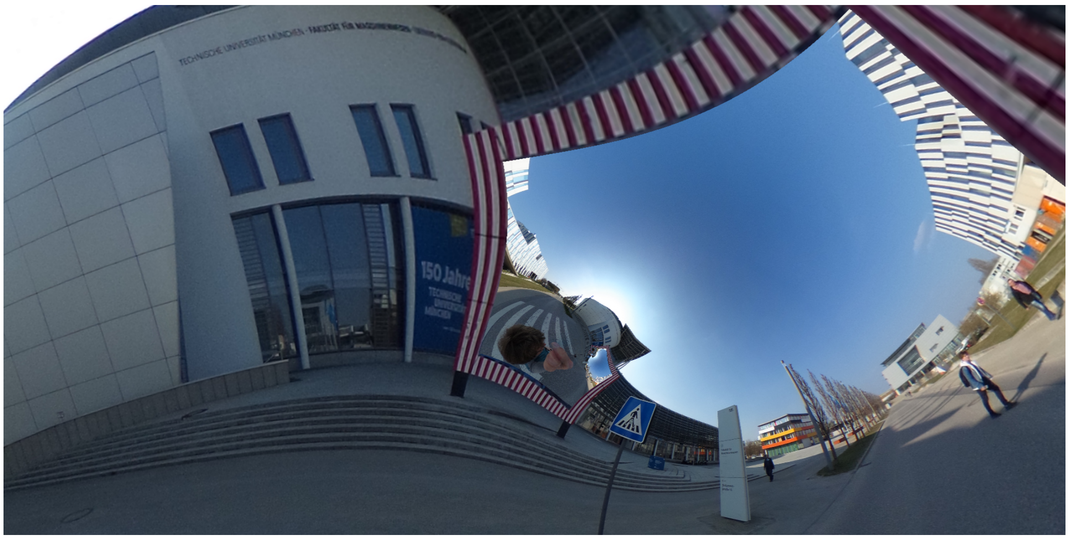


Figure 9.11.: A twisted Droste effect applied to a spherical image by using the techniques from Section 9.2 on a spherical image $\chi : \hat{\mathbb{C}} \to [0,1]^3$

Droste-function L according to Lemma 70, the applet renders the function

$$\tilde{\chi} \circ \rho \circ \sigma^{-1} \circ \textsc{MoveCenter} \circ \exp \circ \textsc{AddPeriod}^{\alpha}_{\textsc{MoveCenter}(S)}(z) \circ \pi \circ L \circ \textsc{UserRotation} \circ \textsc{Screen}.$$

Again, it is possible to detect the set S that has to be cut out from a green screen.

This project aimed to generate a playful framework for Mathematics Education. The potential use of spherical live images increases interactivity immensely. Matt Parker used the framework and applets presented in this section several times on stage during the show “You Can’t Polish a Nerd” [3].

[3]See https://www.youtube.com/watch?v=pgyI8aPctaI and https://www.youtube.com/watch?v=UqtaKJQM_GM.

Chapter 10.

Conclusion and outlook

The thesis started with the question of how GPU fragment shaders could be understood as a mathematical model. A definition of DPMs has been established. The abstraction gave a more transparent viewpoint on how to create a class of programs utilizing rather unconventional computational methods. In the context of this model, complexity classes were defined. Several interesting questions on the lower bounds on complexity arose, and some of them are still open. A particular open question remained: Can complexity classes be separated according to the dimension of the real domain allowed within the model?

This work includes a practical part as well. Several different ways for physical realizations of DPMs were outlined. A particular focus lied on realizations through GPU fragment shaders, which has been the initial motivation. A theorem that allows for uniform approximation through finitely sampled textures has been derived. It is likely that the requirements for uniform convergence could be weakened. Also, conditions for convergence in other spaces, for instance, in measure spaces, are still to be examined. In this context, programming of such GPUs has been eased in a way that is close to the original model. This resulted in the software *CindyGL*, which includes shader programming within dynamic geometry software.

Several algorithms for visualizations of mathematical content, which include ray-casting of surfaces, and deformation of footage, were presented using this enhanced setting. Both of the presented implemented pieces of software in the derived scheme have caught public attention and have been used by external entities for mathematics communication.

Bibliography

Aaronson, S., 2005. Guest column: Np-complete problems and physical reality. ACM Sigact News 36 (1), 30–52.

Barnsley, M. F., 1988. Fractals everywhere.

Barnsley, M. F., 2006. Superfractals. Cambridge University Press.

Blum, L., Cucker, F., Shub, M., Smale, S., 2012. Complexity and real computation. Springer Science & Business Media.

Blum, L., Shub, M., Smale, S., 1989. On a theory of computation and complexity over the real numbers: *np*-completeness, recursive functions and universal machines. Bulletin (New Series) of the American Mathematical Society 21 (1), 1–46.

Botana, F., Abánades, M. A., 2014. Automatic deduction in (dynamic) geometry: Loci computation. Computational Geometry 47 (1), 75–89.

Boyd, J. P., 2013. Finding the zeros of a univariate equation: proxy rootfinders, chebyshev interpolation, and the companion matrix. SIAM review 55 (2), 375–396.

Casanova, H., Legrand, A., Robert, Y., 2008. Parallel algorithms. CRC Press.

Catanzaro, B., Garland, M., Keutzer, K., 2011. Copperhead: Compiling an embedded data parallel language. ACM SIGPLAN Notices 46 (8), 47–56.

Cooley, J. W., Tukey, J. W., 1965. An algorithm for the machine calculation of complex fourier series. Mathematics of computation 19 (90), 297–301.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., 2009. Introduction to algorithms. MIT press.

de Casteljau, P., 1963. Courbes et surfaces à pôles. André Citroën, Automobiles SA, Paris.

de Smit, B., Lenstra, H. W., 2003. The Mathematical Structure of Escher's Print Gallery. na.

de Smit, B., McClure, M., Palenstijn, W. J., Sparling, E. I., Wagon, S., 2012. Through the looking-glass, and what the quadratic camera found there. The mathematical Intelligencer 34 (3), 30–34.

Eigenwillig, A., 2008. Real root isolation for exact and approximate polynomials using descartes' rule of signs.

Epstein, D. B., 1992. Word processing in groups. AK Peters/CRC Press.

Erk, K., Priese, L., 2008. Theoretische Informatik: Eine umfassende Einführung. Springer-Verlag.

Evans, A., Romeo, M., Bahrehmand, A., Agenjo, J., Blat, J., 2014. 3d graphics on the web: A survey. Computers & Graphics 41, 43–61.

Farouki, R. T., 1991. On the stability of transformations between power and bernstein polynomial forms. Computer Aided Geometric Design 8 (1), 29–36.

Fischer, G., 2017. Mathematical Models: From the Collections of Universities and Museums – Photograph Volume and Commentary. Springer Fachmedien Wiesbaden, Wiesbaden.
URL `https://doi.org/10.1007/978-3-658-18865-8`

Fischer, M. J., Fischer, M. J., Rabin, M. O., 1974. Super-exponential complexity of presburger arithmetic.

Gregg, C., Hazelwood, K., 2011. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In: Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on. IEEE, pp. 134–144.

Hadeler, K.-P., Müller, J., 2017. Cellular Automata: Analysis and Applications. Springer.

Head, T., 2009. Parallel computing by xeroxing on transparencies. In: Algorithmic Bioprocesses. Springer, pp. 631–637.

Heintz, J., Recio, T., Roy, M.-F., 1991. Algorithms in real algebraic geometry and applications to computational geometry. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 6, 137–163.

Herlihy, M., Shavit, N., 2011. The Art of Multiprocessor Programming. Morgan Kaufmann.

Hölzl, R., 2001. Using dynamic geometry software to add contrast to geometric situations–a case study. International Journal of Computers for Mathematical Learning 6 (1), 63–86.

Hopcroft, J. E., 2008. Introduction to automata theory, languages, and computation. Pearson Education India.

Kajiya, J. T., 1986. The rendering equation. In: ACM SIGGRAPH computer graphics. Vol. 20. ACM, pp. 143–150.

Kaneko, M., 2017. Using tangible contents generated by CindyJS and its influence on mathematical cognition. In: International Conference on Computational Science and Its Applications. Springer, pp. 199–215.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A., 2012. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. Parallel Computing 38 (3), 157–174.

Kobel, A., 2015. Polynomial gcd in the presence of floating-point errors. Mathematics Stack Exchange.
URL `https://math.stackexchange.com/q/1376517`

Konnerth, G.-M., 2017. Interactive tools for visualising phase plots of complex functions. Bachelor's thesis, Technical University of Munich.

Kovács, Z., 02 2019. Teaching inversion interactively with webcams via CindyJS.

Levine, J., 2009. Flex & Bison: Text Processing Tools. " O'Reilly Media, Inc.".

Liste, R. L., 2014. El color dinámico de GeoGebra. Gaceta De La Real Sociedad Matematica Española 17 (3), 525–547.

Marco, A., Martı, J.-J., et al., 2007. A fast and accurate algorithm for solving bernstein–vandermonde linear systems. Linear algebra and its applications 422 (2-3), 616–628.

Marrin, C., 2011. Webgl specification. Khronos WebGL Working Group.

Maskit, B., 2012. Kleinian groups. Vol. 287. Springer Science & Business Media.

Mason, J. C., Handscomb, D. C., 2002. Chebyshev polynomials. Chapman and Hall/CRC.

McShane, E. J., 1934. Extension of range of functions. Bulletin of the American Mathematical Society 40 (12), 837–842.

Mercat, C., 2009. Applications conformes - inversez-vous le portrait! Accessed: 2020-02-28.
URL `http://images.math.cnrs.fr/Applications-conformes.html`

Mercat, C., 10 2015. La diffusion: un lieu pour une mathématique plus humaine? Espace Mathématique Francophone 2015, Rachid Bebbouchi, USTHB, Oct 2015, Alger, Algérie. hal-01313156.

Montag, A., 2014. Interactive image sequences converging to fractals, bachelor's Thesis. Available at `http://aaron.montag.info/ba/main.pdf`.

Montag, A., Richter-Gebert, J., 2016. CindyGL: authoring GPU-based interactive mathematical content. Springer, 359–365.

Montag, A., Richter-Gebert, J., 2018. Bringing together dynamic geometry software and the graphics processing unit. arXiv preprint arXiv:1808.04579.

Mumford, D., Series, C., Wright, D., 2002. Indra's pearls: the vision of Felix Klein. Cambridge University Press.

Munshi, A., Gaster, B., Mattson, T. G., Ginsburg, D., 2011. OpenCL programming guide. Pearson Education.

Nickolls, J., Dally, W. J., 2010. The GPU computing era. IEEE micro 30 (2).

Novikov, P. S., 1954. On algorithmic unsolvability of the word problem.

Nvidia, 2017. CUDA C programming guide, v8.0.

Obreshkov, N., 1952. Generalization of descartes' theorem for imaginary roots. In: Doklady Akademii Nauk SSSR (NS). Vol. 85. pp. 489–492.

Olver, P. J., 2015. Complex analysis and conformal mapping. University of Minnesota.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., Purcell, T. J., 2007. A survey of general-purpose computation on graphics hardware. In: Computer graphics forum. Vol. 26. Wiley Online Library, pp. 80–113.

Pharr, M., Jakob, W., Humphreys, G., 2016. Physically based rendering: From theory to implementation. Morgan Kaufmann.

Ponce Campuzano, J., 02 2019. Complex Analysis: A visual and interactive introduction. Available at `https://www.dynamicmath.xyz/cabook/`.

Reese, J., Zaranek, S., 2012. Gpu programming in matlab. MathWorks News&Notes. Natick, MA: The MathWorks Inc, 22–5.

Reimers, M., Seland, J., 2008. Ray casting algebraic surfaces using the frustum form. In: Computer Graphics Forum. Vol. 27. Wiley Online Library, pp. 361–370.

Richter-Gebert, J., 2017. iornament–aus dem mathematischen maschinenraum. Mitteilungen der DMV 25 (2), 75–81.

Richter-Gebert, J., Kortenkamp, U., 2010. The power of scripting: DGS meets programming. Acta Didactica Napocensia 3 (2), 67–78.

Richter-Gebert, J., Kortenkamp, U. H., 2000. User Manual for The Interactive Geometry Software Cinderella. Springer Science & Business Media.

Richter-Gebert, J., Kortenkamp, U. H., 2012. The Cinderella. 2 Manual: Working with The Interactive Geometry Software. Springer Science & Business Media.

Richter-Gebert, J., Orendt, T., 2009. Geometriekalküle. Springer-Verlag.

Sagraloff, M., Mehlhorn, K., 2016. Computing real roots of real polynomials. Journal of Symbolic Computation 73, 46–86.

Schilling, M., 1903. Catalog mathematischer modelle: fur den hoheren mathematischen unterricht. Martin Schilling.

Schleimer, S., Segerman, H., 2016. Squares that look round: Transforming spherical images. arXiv preprint arXiv:1605.01396.

Schlick, C., 1994. An inexpensive BRDF model for physically-based rendering. In: Computer graphics forum. Vol. 13. Wiley Online Library, pp. 233–246.

Scholz, D., 2014. Pixelspiele. Springer.

Simpson, R. J., Kessenich, J., 2009. The opengl es shading language. Language Version 1.

Singh, D., Reddy, C. K., 10 2014. A survey on platforms for big data analytics. Journal of Big Data 2 (1), 8.
URL `https://doi.org/10.1186/s40537-014-0008-6`

Spencer, M. R., 1994. Polynomial real root finding in bernstein form.

Stewart, G. W., 1996. Afternotes on numerical analysis. Vol. 49. Siam.

Stoltenberg-Hansen, V., Lindström, I., Griffor, E. R., 1994. Mathematical theory of domains. Vol. 22. Cambridge University Press.

Stussak, C., 2009. RealSurf - a GPU-based realtime ray caster for algebraic surfaces. In: Proceedings of the 25th Spring Conference on Computer Graphics.

Sutter, H., 2005. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's journal 30 (3), 202–210.

Tarditi, D., Puri, S., Oglesby, J., 2006. Accelerator: using data parallelism to program gpus for general-purpose uses. In: ACM SIGARCH Computer Architecture News. Vol. 34. ACM, pp. 325–335.

von Gagern, M., Kortenkamp, U., Kranich, S., Montag, A., Richter-Gebert, J., Strobel, M., Wilson, P., 2019. The CindyJS Project: A JavaScript framework for interactive (mathematical) content. Available at `https://github.com/CindyJS/CindyJS` and `https://cindyjs.org`.

von Gagern, M., Kortenkamp, U., Richter-Gebert, J., Strobel, M., 2016. CindyJS. Springer, 319–326.

von Gagern, M., Mercat, C., 2010. A library of OpenGL-based mathematical image filters. In: International Congress on Mathematical Software. Springer, pp. 174–185.

Vukicevic, V., 2014. Bugzilla@Mozilla [webcl] add opencl in gecko. Accessed: 2017-05-20. URL `https://bugzilla.mozilla.org/show_bug.cgi?id=664147#c30`

Warren, H. S., 2013. Hacker's delight. Pearson Education.

Wegert, E., Semmler, G., 2010. Phase plots of complex functions: a journey in illustration. Notices AMS 58, 768–780.