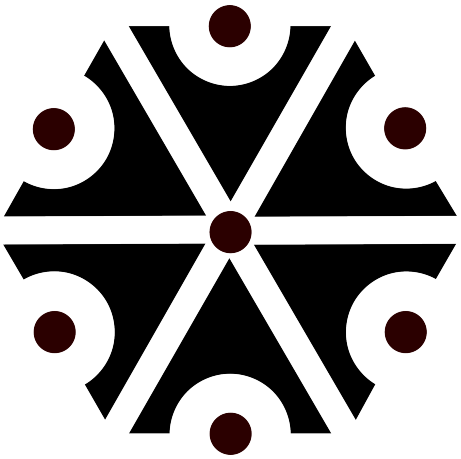
PERUN

Bc. RICHARD DOBŘICHOVSKÝ



A Lightning Plug-in for The Foundry's NUKE Compositor

Master's Thesis

Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University in Prague

Supervisor: Ing. Jaroslav Sloup

May 2016

Richard Dobřichovský: *PERUN,* A Lightning Plug-in for The Foundry's NUKE Compositor, Master's Thesis

Set with the Classicthesis LaTeXtemplate by André Miede.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Richard Dobřichovský**

Study programme: Open Informatics
Specialisation: Computer Graphics and Interaction

Title of Diploma Thesis: **A Lightning Plug-in for the Foundry's Nuke Compositor**

Guidelines:

Study the physics of lightning and light propagation through the atmosphere. Make a survey of existing methods used in computer graphics to simulate lightning and rendering of light bolts. Based on the examined literature implement a physically based lightning generator that will utilize GPU to accelerate the whole computation.

Explore the possibilities of creating plug-ins for the Foundry's Nuke compositor and transform the standalone implementation into a Nuke plug-in. Implement lighting of the surrounding scene by generated bolt of lightning.

Demonstrate the developed plug-in functionality by compositing lightning into several different scenes. Compare results of simulation with real video sequences of lightning.

Bibliography/Sources:

[1] Theodore Kim, Ming C. Lin: Fast Animation of Lightning Using an Adaptive Mesh. IEEE Transactions on Visualization and Computer Graphics, 13(2):390-402, March 2007.

[2] Srinivasa G. Narasimhan, Shree K. Nayar: Shedding Light on the Weather. In Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR'03, pages 665-672, 2003. IEEE Computer Society.

[3] Vladimir A. Rakov, Martin A. Uman: Lightning: Physics and Effects. Cambridge University Press, 1st edition, 2003.

Diploma Thesis Supervisor: Ing. Jaroslav Sloup

Valid until the end of the winter semester of academic year 2017/2018

prof. Ing. Jiří Žára, CSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 29, 2016

ACKNOWLEDGEMENTS

I would like to thank my advisor, Ing. Jaroslav Sloup, for his constructive approach and advice. Also, I would like to thank my parents for their full support during my studies.

DECLARATION

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

In Prague, the 27th of May, 2016

ABSTRACT

The goal of this thesis is to create a physically based lightning simulator for use within the VFX industry. The thesis studies the physics of real lightning creation and explores existing methods for its simulation, both visual and physically based. We use the Dielectric Breakdown Model as our physically based method of choice and offer four variations of a conjugate gradient solver used within to solve systems of linear equations in order to calculate electric potential. Further, we also discuss the rendering of electrical bolts and scene illumination induced thereby. We implement our Perun lightning generator as both a standalone application and a plug-in for The Foundry's Nuke compositor. Finally, we demonstrate the use of the implemented plug-in to composite lightning into several shots and offer an insight to the process of doing so.

Keywords: Perun, lightning, simulation, Dielectric Breakdown Model, NUKE, plug-in, video, VFX

ABSTRAKT

Cílem této práce je vytvoření fyzikálně založeného simulátoru blesků jakožto efektu k použití ve filmové postprodukci. Tato práce nejprve zkoumá fyzikální podstatu blesků a prezentuje existující metody pro simulaci jejich tvaru, jak fyzikálně založené, tak visuální. Vybranou fyzikálně založenou metodou pro tuto práci je metoda průrazu dielektrikem; implementujeme čtyři varianty metody sdružených gradientů pro řešení systémů lineárních rovnic k výpočtu elektrického potenciálu. Dále se v práci zabýváme renderováním elektrických výbojů a osvětlováním scén, jichž jsou součástí. Náš generátor blesků Perun implementujeme jako samostatnou aplikaci i jako plug-in pro kompositor Nuke. Nakonec demonstrujeme užití tohoto plug-inu vložením vygenerovaných blesků do několika scén a objasňujeme postup k tomu zvolený.

Klíčová slova: Perun, blesk, simulace, Dielectric Breakdown Model, NUKE, plug-in, video, VFX

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 MOTIVATION

Although lightning is not a natural phenomenon one would experience ever so often in real life, it would be hard to imagine a science-fiction film without it. In fact, on average on Earth, a lightning flash occurs more than 50 times per second, which is definitely not marginal. Therefore we find it a valid topic to look into in terms of computer simulation.

In film/video production, effects such as lightning are normally added to the footage in post-production, as recording such effects together with the scene itself would be rather impractical, or even impossible. Many effects found in science fiction films don't even exist in reality, or would be harder and more expensive to create physically than as CGI.

It is surprising to see, how many today's films or series incorporate some form of lightning: The Matrix Trilogy, Watchmen, Marvel's Agents of S.H.I.E.L.D. or Star Wars Rebels just to name a few. These effects, either pre-rendered from 3D or generated directly in 2D, are blended into the footage during *compositing*, i.e. assembling multiple images to make a final image. Today, this is done digitally with a wide range of software, one of the most widely used in professional post-production being The Foundry's NUKE. An effect that NUKE does not have under the hood is lightning. This might not be a critically missing feature, as large studios will have their own, in-house developed effects plug-ins and a team of software developers to create effects tailored for the particular film in the making; smaller studios though have to rely on expensive 3rd-party plug-in collections, buy stock footage or use completely different software for compositing. Therefore we decided to create a dedicated lightning effect plug-in for NUKE to fill a small hole on the market.

1.2 SYMBOLICS

The project was named Perun, after the ancient Slavic god of thunder and lightning and the highest god of the Slavic pantheon. The prevailing theory is that Perun's name comes from the Slavic verb *perti* meaning "to beat", although Téra offers a wider etymological view on the subject in [33].

Most polytheistic religions have their own god of thunder and lightning—the Nordic Thor, the Celtic Taranis, the Greek Zeus or the Roman Jupiter—but Perun was chosen to reflect our Slavic roots.

The symbol shown on the title page is called a thunder mark, a symbol of Perun's. This particular symbol and several other variations can be found engraved on chimneys or roof beams of many village houses in Eastern Slavic countries. They were believed to protect the buildings from lightning strikes.

Perun is the central character in Havlíček-Borovský's satirical poem Křest sv. Vladimíra [13], set in the time of tzar Vladimir I., who sentences Perun to death for declining to thunder on his nameday. The people are left without a god and therefore disobedient, so tzar Vladimir I. issues a competition for a new god. Though the poem is unfinished and does not explicitly end with Vladimir's christening, it is believed to also reflect christianisation, which occurred under Vladimir's reign in the region.

1.3 OUTLINE

The thesis can be broken down into several distinct parts. In chapter 2 we start with a review of the physical phenomenon of lightning, what it is, what forms it can take and how it gets created in the atmosphere.

In chapter 3 we continue with an overview of existing methods for lightning simulation in computer graphics and later present our chosen physically based method for shape generation in detail.

Next, in chapter 4, we discuss the rendering of electrical bolts and scene illumination in relation to practical compositing.

Chapter 5 first presents a survey of options available for Nuke plug-in creation and later explains the use of the NDK C++ API for the implementation of the Perun lightning plug-in for Nuke.

An essential part of the thesis is showcasing the use of the plug-in in actual scenes—in chapter 6 we accompany the videos on the enclosed CD with a text description explaining how the scenes were composited.

Finally we analyse our achievements, asses the shortcomings and offer several suggestions how the plug-in could be improved or expanded in chapter 7.

2 PHYSICAL PROPERTIES OF LIGHTNING

Although we do not experience lightning ever so often in our lives, on average, lightning strikes the surface of the Earth more than 50 times per second in over 200,000 thunderstorms a day [19]. Apart from that, lightning that strikes the Earth's surface, referred to as cloud-to-ground lightning (CGs, figure 1), only accounts for a quarter of lightning flashes that occur in the atmosphere. The rest mostly fall into the category of cloud discharges, dubbed ICs, consisting of intracloud, intercloud and cloud-to-air discharges (figure 2). Lastly, other unusual forms of lightning also exist such as ball lightning or lightning effects that occur in higher levels of the Earth's atmosphere.

Lightning is a closely studied phenomenon, although all mechanisms are not yet well understood, especially for non-CG lightning. This is mainly due to the fact, that it is near to impossible to measure and analyse the processes that happen relatively far away, and with almost no precise prediction possible as to when and where they will occur. On the other hand, CGs can even be "artificially" induced—or better, "made to strike where we want it to"—during a thunderstorm, a meteorological rocket or balloon with string attached can be launched to help the lightning take place at a specific point, making measurements much easier. Rakov and Uman present an extensive compilation of research on all types of lightning in their book [29].

In this project, we will primarily be dealing with CG lightning, although ICs retain very similar properties to CGs, therefore we expect the same principles can be used for their simulation.

2.1 PLASMA

It would be wise to start with an explanation of what lightning actually is, or, more precisely, what we see—plasma. Plasma is the so-called fourth state of matter and can have many forms, as Kulhánek nicely demonstrates in the 13 chapters of his book [19]—each dealing with a different setting. The term *plasma* was coined by Langmuir in 1928 [20] and Kulhánek presents a general definition, based on Langmuir's:

1. Plasma contains unbound positive and negative particles able to conduct electrical current.

2. Plasma shows "collective behaviour", i.e. reacts to electric and magnetic fields and is capable of creating them.

Figure 1: A cloud-to-ground lightning strike.

3. Plasma is quasineutral—the overall charge of the particles is roughly zero on a larger scale.

Plasma is commonly created by ionising gases, which is usually achieved by increasing temperature and/or pressure. In the process of ionisation, electrons break free from their atom nuclei creating a highly conductive mixture of ions.

Plasma is the most common form of matter in the Universe—the few non-plasmatic objects in space are rock planets, moons or asteroids, Earth being one of the few places in the Universe, where plasma is relatively rare, nevertheless still present: in the ionosphere, in lightning, the aurora borealis, or even artificially created in fluorescent lamps that we use every day [19].

2.2 DOWNWARD NEGATIVE CG LIGHTNING FORMATION

Downward negative cloud-to-ground lightning accounts for around 90% of all CG lightning. It is initiated from cumulonimbus clouds, mostly referred to by people as simply "thunderclouds"—which is, unlike most other folk denominations, actually accurate: the term cumulonimbus is merely a conjunction of Latin *cumulus* (heap) and *nimbus* (storm cloud).

A cumulonimbus cloud is polarised, with the top carrying a positive charge and the bottom a negative charge. Why this polarisation occurs is not entirely known—the prevailing theory is that ice crystal collisions enable electrons to jump from one to another resulting in lighter, positively charged crystals to ascend and heavier, negatively charged crystals to descend [19]. This polarisation creates an electromagnetic field which repels electrons on the Earth's surface under the

Figure 2: Intracloud/intercloud lightning.

cloud and attracts positively charged particles, creating a positively charged patch which travels with the cloud itself.

Electrons from the lower part of the cumulonimbus start creating negatively charged channels towards the Earth's surface; this happens in steps, with an interval between 20 to 50 µs. In each step, branching can also occur, resulting in the characteristic shape of the stroke. Due to the step-characteristic of the process, the channel is called the *stepped leader*. The same process, in a far smaller scale, happens at the surface with opposite polarity. When these channels meet, a highly conductive path through the atmosphere is created and the charge inequality between the bottom of the cloud and Earth's surface results in the *return stroke*, the current of which can peak up to 30 kA. During the return stroke, the temperature of the gas in the channel rises up to 30,000 °K and the gas gets ionised—creating the luminous effect we observe. This leader-return stroke sequence can happen several times in one flash, only the subsequent leaders are called *dart leaders*, because they no longer travel in steps. Before subsequent dart leaders are initiated, so called J- and K- processes occur within the cloud; it is unclear whether they are crucial for dart leader creation, but the serve as a medium to transport additional negative charge towards the existing channel[29].

The overall duration of a flash is typically between 200 and 300 ms and usually consists of 3-5 individual strokes, though as few as 1 or as many as 30 can occur. A breakdown of the process described above is depicted in figure 3.

2.3 OTHER FORMS CG LIGHTNING

Depending on its direction and polarity, apart from downward negative, CG lightning can also be downward positive, upward nega-

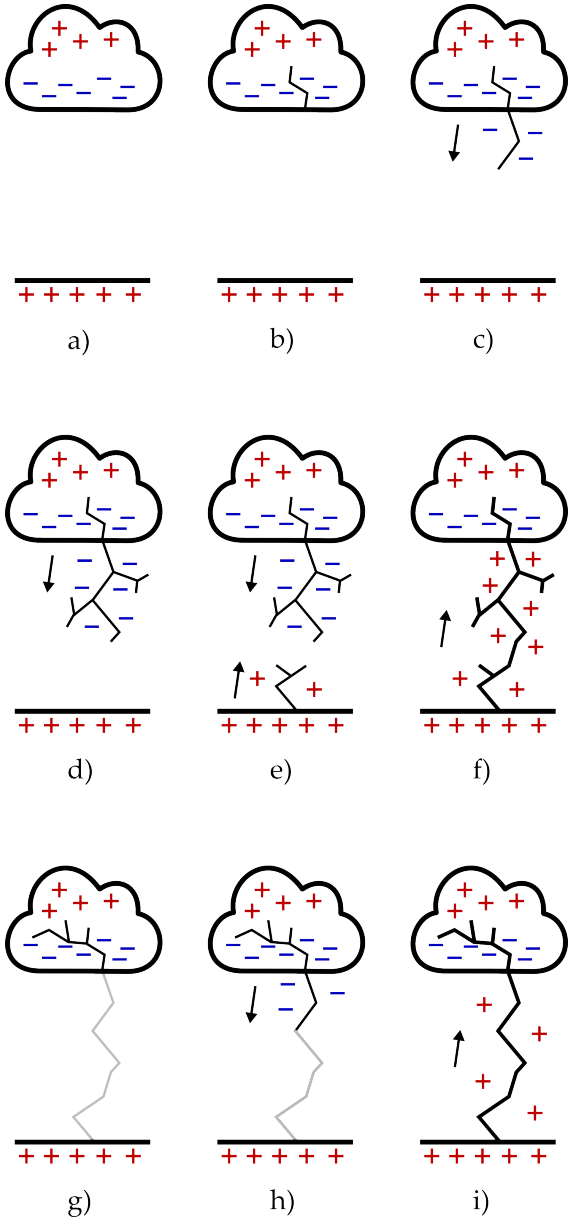


Figure 3: A visual breakdown of CG lightning formation. a) initial charge distribution b) preliminary breakdown c) and d) stepped leader propagation e) attachment process f) first return stroke g) J- and K-processes h) dart leader propagation i) second return stroke

tive and upward positive. The direction can be easily identified by the branching effect present: downward lightning has a "stem" at the cloud and branches toward the ground, upward branching in the opposite direction. Upward lightning is observed on high buildings such as skyscrapers, with the percentage of upward lightning increasing with structure height, statistically from approximately 80 m with 0% probability to structures higher than 500 m with 100% probability—being only struck by upward lightning.

As mentioned earlier, 90% of CG lightning on Earth is downward negative; the remaining 10% is mostly filled up by downward positive lightning. Positive lightning simply differs from negative lightning in polarity: for downward lightning, the stepped leader coming from the cloud has a positive charge. According to Kulhánek, it is not uncommon for the leader to first travel several kilometres horizontally before turning to the ground, therefore sometimes being called "lightning out of nowhere". The energy involved in the strike is also an order of magnitude higher and therefore far more devastating if it strikes an unprotected object.

Statistically, upward lightning is relatively rare—but this is not much of a surprise: skyscrapers only occupy a truly marginal area of the Earth's surface.

3 SHAPE GENERATION

In this chapter, we discuss the creation of the lightning bolt's shape. It is amazing how much effort has to be put into merely producing a wiggly line. First, we present a survey of approaches that deal with lightning shape generation. Based on this survey, we then further explore the details of the physically based method chosen for implementation—the Dielectric Breakdown Model.

3.1 METHOD OVERVIEW

Surprisingly, a relatively wide range of methods already exists to programmatically generate lightning. In general, they can be broken up into two main categories: those physically based, seeking to generate the shape of lightning by simulating electrical discharge in the atmosphere, and those that merely attempt to visually match real-world lightning. Naturally, both approaches have their advantages and shortcomings which we will attempt to analyse in this section.

3.1.1 *History*

Historically, lightning intended to be used in film would be drawn by hand and/or rotoscoped—such an attempt is shown in figure 4. This was nothing exceptional as rotoscoping was a widely used technique to create visual effects prior to the advent of digital compositing. A notable use of traditional rotoscoping would be the glow of lightsabres in the *Star Wars* original trilogy, released between 1977 and 1983. Rotoscoping is of course still used in digital compositing when automatic methods fail or when applying such methods would be inefficient, such as creating a garbage matte.

Another approach at hand would be to reuse real photographs or high-speed camera footage—a technique applicable to both traditional and digital compositing, although we are unsure to what extent such an approach has been used in film production.

3.1.2 *Visual Methods*

Most probably the first attempt to computationally generate lightning was made by Carpenter in 1980, who worked on synthesising fractal geometry on a computer. The video *Vol Libre*, accompanying his paper[4], primarily showcases his fractal methods to generate terrain and, possibly as a side effect of the research, the use of the gener-

Figure 4: Lightning in Karel Zeman's 1958 film *The Fabulous World of Jules Verne* (original title *Vynález zkázy*).

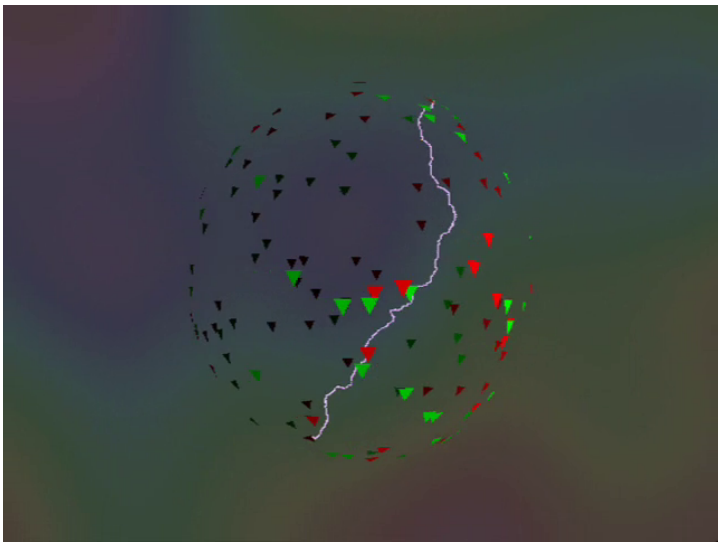


Figure 5: Lightning in Carpenter's 1980 video Vol Libre, presented at SIGGRAPH '80.

ated curves as lightning. A screenshot from *Vol Libre* is presented in figure 5. Although not the lightning itself, his fractal methods were soon used in production, the "genesis effect"—terraforming a forsaken planet—in *Star Trek II: The Wrath of Khan*. The scene was conducted by Industrial Light & Magic.

The principle of fractals though, i.e. repeating a pattern at finer and finer scales, is used in some of the visual methods further described in this section. For detailed reviews of fractals, the reader is referred to [8] and [21].

Most papers describing visual methods refer to Hill, who analysed tortuosities[1] of several digitised strokes and concluded that the angles between short segments have a Gaussian distribution with a mean of 16°. Furthermore, he reported in his paper[14], that tortuosity depends neither on segment length nor on the position within the stroke—simply speaking, that the characteristics of the lightning bolt's wiggling remain the same all the way from the cloud to the ground.

By the study of photographs, Reed and Wyvill[30] come to the same conclusions as Hill—that the angle between lightning branches forms a normal distribution centred around 16° and use a variance value of $\sigma^2 = 1$. In their method, using this assumption, they generate underlying shapes which they then replicate using fractal scaling. They state that their method suffers from the use of a pseudo-random number generator which makes it hard for the user to control branching consistently—a feature certainly required by an artist.

Kruszewski builds on this and attempts to overcome this ambiguity of random number generation by using the *random split tree method*[6] which effectively controls the way branching is performed[18]. On top of that, Kruszewski adds tortuosity (or as they call it *electrification*) to the generated straight lines—this is, once again, based on fractals: in this case a two dimensional midpoint replacement algorithm. If a line segment AB is longer than a set l_{max}, it is subdivided into two segments AD and DB, where D is randomly selected within a radius of point C which is the midpoint of line segment AB. This is performed recursively until all line segments are shorter than l_{max}. Additional tortuosity is a very important feature for visual methods, as it allows to separate actual branching from, let's say "visual properties of electricity". In addition to that, it adds a separate and very intuitive user control for the artist to use.

Glassner uses statistical data gathered from the analysis of lightning images to randomly choose line segments with various properties (length, angle between new and previous segment and whether or not to create a new branch)[11]. Like Kruszewski, Glassner adds additional tortuosity. We would like to mention the name Glassner chose for the project: *Digital Ceraunoscope*. A ceraunoscope is "an in-

1 "Tortuosity is a property of curve being tortuous (twisted; having many turns)"[36]

strument or apparatus employed in the ancient mysteries to imitate thunder and lightning"[37]. Until coming across this article, we were unaware of the fact that there actually is a single English word to describe a lightning simulator.

In his master's thesis [2], Bergl combines Reed and Wyvill's method with Lindenmayer systems (L-systems). Lindenmayer systems were originally developed as a way of modelling plants[28]—although seemingly unrelated, the natural phenomena are surprisingly similar, so L-systems can also be adopted to model lightning branching. Vice versa, also physically based methods by Kim et al. were used to generate plants, or used as a bias when modelling with L-systems[17]. Lindenmayer systems can be thought of as *rewriting systems*—starting with an axiom (a symbol or string of symbols) and recursively using a set of replacement rules to replace all occurrences of symbols in the current string, a complex structure can be quickly generated. Symbols may define a line of certain length, branching, a turn, a polygon or any other feature required to model the desired phenomenon, making it a very versatile system. It may seem that L-systems demonstrate a lot of similarity to formal grammars: the main difference is that rules are not applied sequentially, but in parallel, for all symbols in the string at once. In order to generate sequences with a degree of randomness, multiple rules for replacement for a single symbol have to be allowed—this variant, dubbed Stochastic L-systems, is used in Bergl's thesis.

3.1.3 *Physically Based Methods*

In its essence, nature is continuous and, with today's technologies, it is impossible to simulate entirely and precisely. In most cases, some sort of discretisation has to be applied in order to feasibly calculate such a simulation. In physics, this is called a *lattice model*. The phenomenon that physically based lightning generating methods attempt to simulate is that of *electrical breakdown*. All the physically based methods presented below are centred around a 2D or 3D grid, together with parameters defining the electrical field they attempt to simulate—boundary conditions. A starting point on the grid is chosen (by the user), and in subsequent iterations new cells neighbouring the present shape are picked (by the algorithm) to expand it. The process of new growth cell selection and the way of possible grid potential recalculation differs between methods. An example of such grid is depicted in figure 6.

The first endeavour in simulating dielectric breakdown was Diffusion-limited aggregation[8]. Inspiration for this method comes from the electrolysis of copper sulphate ($CuSO_4$), where Cu^{2+} ions drift along Brownian paths towards the cathode, where they build up copper deposit creating a fractal pattern. In the simulation, a "particle" is

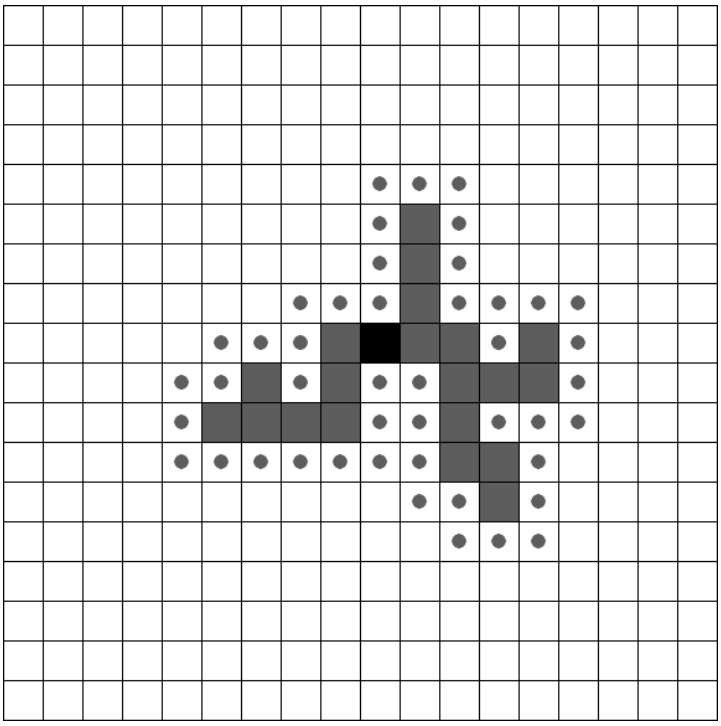


Figure 6: An example of a grid configuration of a physically based method after several iterations. Black cell: origin, gray cells: growth sites now part of shape, gray circles: candidate sites.

released from a random point on a circle of radius r centred around the origin. This particle performs a random walk through the lattice, either ending up in one of the candidate sites, thus adding itself to the shape as a growth cell, or leaving the circle and not contributing. The particle moves through the lattice with an equal probability of $\frac{1}{4}$ in each allowed direction (north, east, south, west). As is, this method would not be directly applicable to lightning as it e.g. lacks control over direction—the generated shape spreads uniformly towards the boundaries.

A more general approach was introduced by Niemeyer et al.: the Dielectric Breakdown Model[24] builds on calculating the electric potential of grid cells by solving Laplace's equation (see section 3.3); then, based on potential differences, probability values are assigned to candidate sites and a growth site is randomly chosen, weighted by the probabilities. The growth site is added as a boundary condition of the electric field and Laplace's equation has to be re-solved. As this was selected as our method of choice, the method itself as well as its underlying concepts, such as solving Laplace's equation, will be described in much more detail further in this section. Kim and Lin use this method in their research on physically based lightning generation in [15] and further improve its performance by the use of an adaptive grid[16]; Bickel et al. do so as well in a similar fashion in [3].

Solving Laplace's equation is costly, so researchers attempted to refrain from doing so: Kim et al. derive an approximation for the case when boundary conditions are set to be $\phi = 1$ on a circle of infinite radius around the generated shape, growth sites of which set boundary conditions of $\phi = 0$. Calculating initial potential in a point is rather expensive (involving the calculation of Euclidean distances between the point and all boundary conditions; the infinite radius is replaced by a user-settable value which influences the amount of branching), but recalculating it after a new growth site is selected is much faster—briefly stated, it is done by merely contributing the influence of the new growth site/boundary condition. Furthermore, the method does not require the calculation of potential in all grid cells—only candidate cells, making it very efficient, though iteration time greatly depends on the number of candidates—as the simulation progresses, iteration times increase significantly. Fortunately, its nature allows efficient adoption of GPU acceleration and even after 12000 iterations, GPU iteration times remain constant while CPU times increase nearly tenfold[7]. In the software/research project leading to this thesis[7], we experimented with this method, attempting to apply it to different charge configurations, specifically to generate a cloud-to-ground lightning. Although the results could lead to usable images, we experienced rather erratic behaviour when setting parameters controlling branching—leading to either too much or too little branching. We finally opted to drop the method and concentrate on calculating the potential across the grid by solving Laplace's equation as mentioned above.

Sosorbaram et al., instead of solving Laplace's equation, use Coulomb's law for a system of discrete charges—introduced by boundary conditions, and use it to calculate the potential in the grid cells[32]. To help randomness, they also add noise to the field. In contrary to other methods, growth sites do not contribute to the field and so the potential field does not get recalculated. In their approach, Sosorbaram et al. choose random points around a leader, and subsequently select N of those with the largest progression probability (N is a settable parameter influencing the amount of branching). Candidate points which fall within a radius R of any of the N points are removed, the remaining become new leaders. R is another user parameter changing the character of branching—increasing N increases branching, whereas increasing R decreases branching.

3.1.4 *Existing Products*

Two main products are available on the market: Adobe After Effects[2] come with a built-in "Advanced Lightning" effect. The Sapphire plug-in collection by Genarts consists of over 250 different effects plug-ins

2 a compositing suite by Adobe Systems Inc.

and is well known known in the post-production industry—one of the plug-ins, the "Zap" effect, also simulates lightning.

It is unclear what methods commercially available applications use, as they do not openly present their work in order to sell their products. It is quite apparent though, that the methods used are based on the visual approaches presented earlier, probably combining several of them together. Both products seem to generate a very simple underlying shape with no tree-like branching and add most of the "lightningness" via tortuosity (e.g. using the methods presented in [11] or [18]). If more branching is set, more instances of these shapes appear to be added to the "parent" jolt at predefined positions (with some degree of randomness, but still following certain rules so that their distribution is more or less even), optionally with more decay if desired. This way, it is relatively easy to create lightning with a lot of branching towards the target, which can be desirable in many sci-fi situations. On the other hand, doing this in a physically based method is rather complicated or even impossible when only relying on physically based simulation. From our research, the only approach which seems to intuitively achieve this to some extent is that of Sosorbaram et al. It would be unfair though to present the lack of such a feature as an a priori negative aspect of physically based methods: physically based methods attempt to simulate lightning occurring in the *real world*—in the atmosphere—whereas science fiction applications usually have no real physical background, but rather aesthetic *inspired* by real-world occurrences of lightning put into situations which would be impossible to achieve in (at least today's) scientific experiments.

Sapphire do a good job animating jolts, but this apparently comes from their way of creating the shape—the original underlying curve (which could just be a simple Bézier curve) is modified slowly and gradually, and only added tortuosity varies greatly in each frame, thus creating an appealing continuous animation of "sci-fi" lightning. Unfortunately, if branching is set, the points where branches leave their parent jolt stay the same, so this feature is effectively usable only with no or very little branching.

Apart from these, major post-production studios are likely to have their own, in-house developed lightning generators: many science fiction movies or TV series feature such a large amount of scenes including lightning it makes sense to develop an in-house tool fitting into the production pipeline and delivering results tailored to the needs of the picture being produced.

3.2 THE DIELECTRIC BREAKDOWN MODEL

For the project, we chose to experiment with the Dielectric Breakdown Model, first described by Niemeyer et al. in [24], but expanded and applied to practical simulation of natural phenomena including

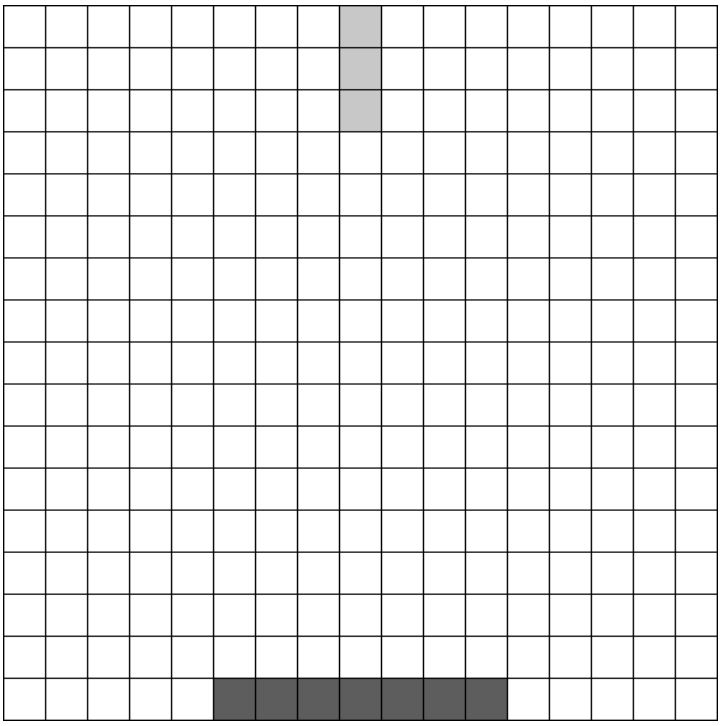


Figure 7: Typical initial grid configuration for cloud-to-ground lightning. Light gray: origin/growth sites, dark gray: destination sites.

lightning by Kim et al. in a series of papers: [17], [15] and [16] as well as Bickel et al. in [3].

Briefly, DBM is an iterational algorithm to simulate pattern growth of an electric discharge. The algorithm uses a 2D grid of electric potential values—figure 7 depicts a typical initial configuration. As a result of every iteration, a new growth site is selected, gradually creating a shape in a manner already illustrated in figure 6. If the shape grows as far as the destination set, generation terminates.

First, in each iteration, the potential in every cell has to be calculated, based on the boundary conditions incurred by growth and destination sites. The potential of growth sites is set to $\phi = 0$ and destination sites to $\phi = 1$. Determining potential across the grid is achieved by solving Laplace's equation:

$$\nabla^2 \phi = 0. \tag{1}$$

How that is done is explained in section 3.3. For now, let's assume we know how to do so: each grid cell has a value of electric potential ϕ reflecting the current grid situation.

Next, candidate sites have to be determined: candidate sites are all sites adjacent to growth sites (and logically not growth sites themselves). Because the bolt grows by one pixel (growth site) per iteration, new candidates are found only within the 8-neighbourhood of the new growth site.

Having candidates, we determine the probability p_i of a candidate being chosen:

$$p_i = \frac{(\Phi_i)^\eta}{\sum_{j=1}^{n} (\Phi_j)^\eta} \tag{2}$$

where the sum in the denominator is a sum over potentials of all candidate sites and η is a "branching" constant—the higher the value, the faster the bolt reaches the destination, incurring less branching. Setting $\eta = 0$ would not respect the values of electric potential across the grid and therefore would result in complete randomness, creating a so-called *eden set*. On the other hand, already $\eta = 4$ results in a straight line, effectively leaving a reasonable domain of $\eta \in [1, 3]$. In the project, we use two values, $\eta = 1$ for more branching and $\eta = 2$ for less branching.

Based on a random value $\nu \in [0, 1]$, we choose a candidate according to its probability—cycle through candidates until the sum of all probabilities of previous candidates is greater than ν.

Finally, the chosen candidate has to be removed from the candidate set, added to the grown set and the grid updated. To effectively store the shape and its branching in order to render it afterwards, we do so in a graph, explained in section 4.1.

3.3 LAPLACE'S EQUATION

The primary problem in the field of electrostatics is to determine properties of an electrical field based on a given charge distribution. We will start with Gauss's law which, in its differential form, states the divergence of an electrical field as follows:

$$\nabla \cdot \mathbf{E} = \frac{\varphi}{\varepsilon_0}. \tag{3}$$

where $\nabla \cdot \mathbf{E}$ denotes the divergence, φ charge density and ε the permittivity of vacuum. According to Faraday's law, an electric field has zero curl and therefore we can define electric potential ϕ as the gradient

$$\mathbf{E} = -\nabla\phi. \tag{4}$$

Putting that together we write

$$\nabla \cdot \nabla\phi = \nabla^2\phi = -\frac{\varphi}{\varepsilon_0} \tag{5}$$

which in fact is so called *Poisson's equation*, which we will later use for animation in section 3.6. In a charge-free region—which is our case for now—the right hand term $-\frac{\varphi}{\varepsilon_0}$ equals to zero which leaves us with Laplace's equation

$$\nabla^2\phi = 0. \tag{6}$$

For analytical solutions and more context in the field of physics, the reader is referred to corresponding literature, such as [12]. Here we will describe the derivation of the discrete form of the equation that we later use to come up with a numerical solution. The derivation uses the *method of discrete differences*. We show the derivation in two dimensions, but an analogical approach can be applied in 3D. As we have already mentioned in section 3.2, the simulation runs on a grid, and we are required to determine the potential for each of the grid cells. First, we rewrite Laplace's equation in Cartesian coordinates

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0. \tag{7}$$

Let's now recall the definition of a first order derivative

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{8}$$

Using the central difference scheme, we approximate the derivative

$$f'(x) \approx \frac{f(x+\frac{1}{2}\Delta x) - f(x-\frac{1}{2}\Delta x)}{\Delta x} \tag{9}$$

where Δx corresponds with the size of a grid cell in the simulation. By applying this once more, we receive an approximation for the second order derivative

$$f''(x) \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2}. \tag{10}$$

Now let's define our simulation grid: an M by N grid, a grid cell ϕ_{ij} referring to the potential $\phi(x_i, y_j)$, $i \in \{1,\dots,M\}$, $j \in \{1,\dots,N\}$. Without loss of generality we will also define $\Delta x = \Delta y = 1$. So, applying our approximation on equation 7 for a single grid cell, we get

$$(\nabla^2\phi)_{ij} \approx \phi_{i+1,j} - 2\phi_{ij} + \phi_{i-1,j} + \phi_{i,j+1} - 2\phi_{ij} + \phi_{i,j-1} = 0 \tag{11}$$

therefore, the value of each grid cell is determined as

$$\phi_{ij} - \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right) = 0. \tag{12}$$

which defines a system of $M \times N$ linear equations.

In order to actually define the electrical field we determine the potential values for, we have to introduce *boundary conditions*, i.e. grid cells, the potentials of which are known and do not change. A destination site introduces a boundary condition $\phi_{i,j} = 1$, whereas a growth site $\phi_{i,j} = 0$.

The system of linear equations is best represented in matrix form $Ax = b$, where each row/column represents a grid cell (matrix A is symmetric). We show how such a matrix is constructed on a simple, small-scale example—a 3×3 grid shown in figure 8. A will be

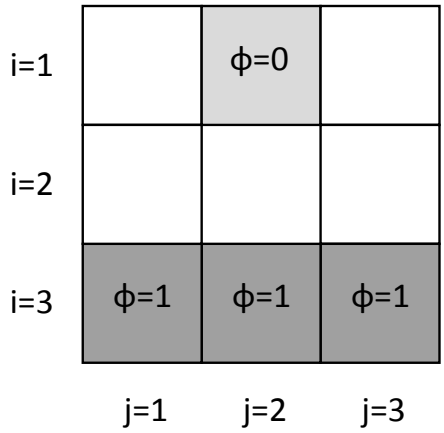


Figure 8: An illustrative 3×3 grid configuration to show the creation of matrix A. Gray cells mark boundary conditions.

a square 9×9 matrix, vectors b and x 9×1 column vectors. But first, let's consider a 3×3 grid with no boundary conditions: this would of course not give any useful results, but serves as a good illustration as to how matrix A is constructed:

$$\left(\begin{array}{ccc|ccc|ccc} 1 & -\frac{1}{4} & 0 & -\frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{4} & 1 & -\frac{1}{4} & 0 & -\frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{4} & 1 & 0 & 0 & -\frac{1}{4} & 0 & 0 & 0 \\ \hline -\frac{1}{4} & 0 & 0 & 1 & -\frac{1}{4} & 0 & -\frac{1}{4} & 0 & 0 \\ 0 & -\frac{1}{4} & 0 & -\frac{1}{4} & 1 & -\frac{1}{4} & 0 & -\frac{1}{4} & 0 \\ 0 & 0 & -\frac{1}{4} & 0 & -\frac{1}{4} & 1 & 0 & 0 & -\frac{1}{4} \\ \hline 0 & 0 & 0 & -\frac{1}{4} & 0 & 0 & 1 & -\frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{4} & 0 & -\frac{1}{4} & 1 & -\frac{1}{4} \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{4} & 0 & -\frac{1}{4} & 1 \end{array}\right) \begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{12} \\ x_{22} \\ x_{32} \\ x_{13} \\ x_{23} \\ x_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{13}$$

Now we will introduce boundary conditions as shown in figure 8. In boundary condition cells, we do not want to calculate anything: their corresponding rows and columns, as well as their entry in the b vector will be zero. Next, let's consider cells that neighbour boundary condition cells: the boundary condition cell with a set value of ϕ_{ij} will always influence such a cell by $\frac{1}{4}\phi_{ij}$. Therefore, the value of b_{ij}

has to be increased by $\frac{1}{4}\phi_{ij}$. For the grid configuration in figure 8 we therefore get

$$\left(\begin{array}{ccc|ccc|ccc} 1 & -\frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{4} & 1 & 0 & 0 & -\frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{4} & 0 & 0 & 1 & 0 & 0 & -\frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & -\frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{4} & 0 & -\frac{1}{4} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{12} \\ x_{22} \\ x_{32} \\ x_{13} \\ x_{23} \\ x_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{4} \\ 0 \\ 0 \\ \frac{1}{4} \\ 0 \\ 0 \\ \frac{1}{4} \\ 0 \end{pmatrix} . \quad (14)$$

3.4 METHOD OF CONJUGATE GRADIENTS

The discretised version of Laplace's Equation leaves us with a system of linear equations $Ax = b$ we are required to solve. It is important to note on the structure and size of the system: on a $N \times N$ simulation grid we construct a $N^2 \times N^2$ matrix. Even for small grid configurations such as 64×64, it would be infeasible to work with an explicit representations of matrix A. From the analysis in the previous section, it can easily be seen, that each column or row contains a maximum of 5 entries, thus being a nice sparse matrix. Several ways of representing sparse matrices exist, more on that topic can be found in section 3.7 below. Many methods exist for solving sparse systems of linear equations—fortunately, matrix A is also symmetric and positive-definite so we can apply the method of conjugate gradients[31]—an effective iterative numerical method. In each iteration, the solution to the system x_k is refined until a user-defined tolerance or a maximum number of iterations is reached. Pseudocode for the algorithm is offered below:

$k := 0$
$\vec{r_k} := \vec{b}$
$\vec{d_k} := \vec{b}$
$\vec{x_k} := \vec{0}$
while $\|\vec{r_k}\| >$ tolerance **and** $k <$ max_iter **do**
    $s_k := A\vec{d_k}$
    $\alpha := \frac{r_k \cdot r_k}{d_k \cdot s_k}$
    $x_{k+1} := x_k + \alpha d_k$
    $r_{k+1} := r_k - \alpha s_k$
    $\beta := \frac{r_{k+1} \cdot r_{k+1}}{r_k \cdot r_k}$
    $d_{k+1} := r_{k+1} + \beta d_k$
end while

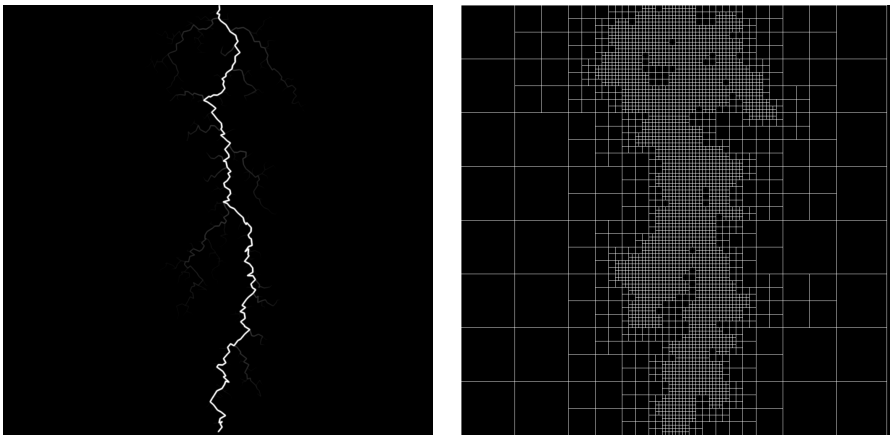
The algorithm maintains three vectors in each iteration k: $\vec{x_k}$ represents the approximate solution—after the while loop terminates, $\vec{x_k}$ is the solution. Vector $\vec{r_k}$ denotes the *residual*, which is sought to be minimised and $\vec{d_k}$ is the search direction. Coefficients α and β ensure the choice of a good search direction, minimising $\vec{r_k}$ and thus increasing precision of the solution $\vec{x_k}$—for proof thereof and more theoretical background, the reader is referred to Shewchuk's report[31], where he provides a complex analysis of the method.

It is quite evident that the computationally most expensive step is the matrix-vector multiplication on line 6; therefore it is crucial for this step to be implemented efficiently and in parallel. More on that can be found in section 3.7.

When solving large sparse systems of linear equations, it is a common practice to use *preconditioning*. Knowing the structure of matrix A, it is possible to rearrange the elements in the matrix in such a way to allow much quicker convergence of iterative solving methods. Kim and Lin use preconditioning in their work but note that for smaller matrix sizes, the computational expense of preconditioning is larger than the actual speed-up gained by the solver's faster convergence. In our project, we attempt to achieve running times which would not distract the artist from their work: the largest grid configuration we use is 256×256, on which it takes around 5 seconds to generate a lightning bolt (more benchmarks in section 3.8)—much more often though, an artist would choose lower resolution grid configurations on which it is possible to achieve generation times even under 500 milliseconds per frame. In their work, Kim and Lin do not concentrate on this restraint and in turn run their simulations on much larger grid configurations including 3D grids. In this relation, it is apparent, that preconditioning would not speed-up the generation in the case of our *relatively* low-resolution grid configurations. Furthermore, in the next section, we describe the use of an adaptive grid (which itself on the other hand offers significant speed-up). Unfortunately, the patterns found in A matrices corresponding to the adaptive grid configuration make it much trickier to implement preconditioning, both on theoretical grounds and computationally. Therefore, we decided not to implement preconditioning at all as we did not expect any speed-up. For details on preconditioning, including the adaptive grid case, the reader is directed to Kim and Lin's paper [16].

3.5 ADAPTIVE GRID

So far, we calculate potential values in all grid cells. However, the number of cells of which that value is ever used in the simulation is very low. Suppose a 256×256 grid: simulation might be done in say 1200 iterations—resulting in 1200 growth sites and about twice as many candidate sites. Together, that is only about 5% of the total. An



(a) (b)

Figure 9: A generated lightning bolt (a) and a corresponding quadtee (b).

obvious first step is exploiting the nature of lightning and not using a square, but rectangular grid: but that only increases the proportion to 10%, which is still not great. Also, when using $\eta = 1$, the rectangular grid can get too narrow for some of the generated branching.

The solution is to use an adaptive grid: high spatial resolution is only required in those areas, where the new shape is evolving and around boundary condition cells. We choose to implement a *quadtree* data structure(figure 9) a *quadtree* is a rooted tree, where each node represents a square area. A node is either a leaf, or has four children: SW, SE, NW, NE. In our implementation, we also require the tree to be *balanced*: the size of neighbouring cells differs at most by a factor of 2. This makes neighbour searches much easier—an operation we do very often when constructing the A matrix for the simulation. For details on quadtrees themselves, the reader is referred to [5]. Here, we will concentrate on how quadtrees change the simulation grid and matrix A.

Three situations can occur within the simulation grid: a neighbour can be of the same size, 2 times larger or half the size. The first case is analogical to the regular case: the influence of such a cell remains the same. In the other two cases, the influence is set to $\frac{1}{2}$ of what it would be in the first case. The value of the matrix entry on the diagonal though (which was 1 in the regular grid case) has to respect these relations and be set to the opposite of the sum of cell influences—the sum of each row and column of matrix A equals to zero. Surprisingly, setting different coefficients does not greatly impact the simulation[16], at least not to an extent that the influence could be visually characterised.

Apart from the fact that now a cell can theoretically use up to nine matrix entries per row (as opposed to 5 in the regular grid case), more implementation issues are introduced: in the regular case, ma-

trix A only changed when a growth site newly became a boundary condition—the corresponding row and column would be set to zero, as well as the entry in vector b. With an adaptive grid, the number of required operations can greatly increase: suppose we find a new candidate and due to quadtree balancing we have to subdivide the quadtree. Apart from the newly subdivided cells, all of their neighbours have to be updated in the matrix as the former parent cell is not valid any more. This is revised further in section 3.7.

3.6 ANIMATION

Real-world lightning is very quick: when compositing such lightning into a scene, it will be shown on one frame only, possibly two. However, science fiction set-ups demand a dancing lightning bolt that can keep on going even as long as seconds—therefore tens of frames. A first assumption would be: just generate new frames and put them in sequence. Unfortunately, this does not create pleasing results—in its nature, the physically based shape generation method creates greatly randomised patterns—putting these shapes in sequence just makes an impression of the bolt unpleasantly oscillating. Hence we would like to make every successive shape follow a similar general path as its predecessor. Fortunately, this can be achieved by depositing a small charge along the main channel of the preceding shape in the new simulation grid. This slightly attracts the generated shape towards that of last frame's, whilst maintaining a certain degree of randomness.

In section 3.3, when deriving Laplace's equation, we came across Poisson's equation: Laplace's equation was only a special case of Poisson's equation in a charge free region. Let's recall Poisson's equation

$$\nabla^2 \phi = -\frac{\varphi}{\varepsilon_0}. \tag{15}$$

The interpretation for our case is decreasing the value in vector b by a small amount if the corresponding cell neighbours one which was part of the main channel in the previous frame.

This approach does introduce a slight performance issue: the quadtree structure described in the section above cannot be used as effectively, as we require to keep the subdivision created in the previous frame. We have found though, that it is possible to slightly lower the precision of the conjugate gradient solver without visual impact on the results, thus reducing the influence on performance.

3.7 IMPLEMENTATION

For the shape generation process, we implemented four different versions of the methods described above, differing mainly by the solver

used. First, we used a linear algebra library which included its own conjugate gradient solver for the regular grid version, later on we expanded it for use with an adaptive grid; for comparison we also wrote our own conjugate gradient solver for an adaptive grid, and finally we created a GPU-accelerated version using CUDA.

At the beginning, for simplicity or even proof-of-concept, we tried to find a library for linear algebra which would:

- support BLAS[3] operations,
- support sparse linear algebra,
- include an efficient conjugate gradient solver
- and offer parallelisation.

Such a library appeared to be Eigen[4] so the first implementation attempts focused on utilising Eigen. We started with the regular grid version, which did not pose great difficulties after we got accustomed to Eigen's sparse matrices. Sparse matrices require great attention, as their improper usage can introduce serious performance issues. In Eigen's implementation, random access operations require a rather expensive binary search, so modifying a matrix after it has been created is not a great idea. In the regular grid case, this is not a big deal—in every iteration, we only have to set row and column entries of one cell—which is 9 matrix random access operations in total. Even though still very expensive, these are negligible compared to the solver. With an adaptive grid on the other hand, entries for many more cells may require changes—if a cell gets subdivided in the quadtree, the entries for all of its parent's neighbours become invalid, as they now neighbour the child nodes, not the parent node. This can even occur recursively, if the factor-of-two neighbourhood rule happens to be broken after a subdivision (see section 3.5 for details).

Eigen introduces so-called *triplets*, tuples holding i, j indices and a corresponding value, defining a non-zero element. From a list of those (or any iterable container), Eigen can construct a sparse matrix in $O(n)$ time. Even though it still seems inefficient, building the matrix from scratch before every solver run only requires a fraction of time of what the solver itself does. Although we have a reasonable way of building the sparse matrix, we still haven't solved the problem of actually *altering* the values: for that, we created a supporting data structure—a triplet list together with a vector of row beginning indices. This allows efficient row removal and insertion—the triplet list passed to Eigen does not have to be ordered. Removing a row means deleting successive items in the linked list of triplets (accessed

3 Basic Linear Algebra Subprograms
4 http://eigen.tuxfamily.org/

in $O(1)$ time due to the row beginning index), inserting can simply be done to the end of the linked list. Also, neither our own CPU nor the CUDA implementation, which we discuss further on, pose the requirement on list ordering: in both versions we access the sparse matrix by rows, therefore maintaining row order is sufficient (and is done per-se in the row beginning indices vector). Therefore, this representation served as the *actual* sparse matrix format. (For CUDA, we also have to construct another sparse matrix in a defined format, but once again, efficient row access is sufficient.)

Being put off by the heavy-handedness of Eigen, we decided to implement our own conjugate gradient solver for comparison. The implementation is dubbed "prog2", because it was inspired by a programming assignment[27], which was simply the second in the Parallel and Distributed Computing class at the University of San Francisco. For Level-1 BLAS operations (vector-vector) we stayed with Eigen's dense vectors; however matrix-vector multiplication follows our sparse matrix representation as presented above. We use parallelisation via OpenMP, parallelising the `for` cycle iterating over matrix rows—each matrix row–vector multiplication is de-facto an independent dot product.

The last to implement was a GPU-accelerated version using CUDA. Nvidia offer two handy libraries, cuBLAS[25] and cuSPARSE[26]; the first, as the name suggests, implements BLAS routines as CUDA kernels, whereas cuSPARSE offers functions that manage operations with sparse matrices on the GPU. Their usage is relatively intuitive: both libraries follow the BLAS specification—for us, the most important were cuBLAS Level-1 routines `axpy()`, `copy()`, `dot()` and `nrm2()`, and cuSPARSE Level-2 `csrmv()` (matrix–vector multiplication $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$, where A is a sparse matrix in CSR[5] format and $\vec{x}$ and $\vec{y}$ are dense vectors). A matrix in CSR format is defined by 3 arrays and an integer specifying the number of nonzeros (`nnz`). The first array is of `nnz` length and stores the non-zero values themselves; the second, of length $m + 1$, where m is the number matrix rows, stores indices of each row's first element in the last (and first) array. The last element of the second array stores the value of `nnz`. The last array contains column indices. Of course, creating the matrix and copying it to GPU memory creates certain overhead, however it is still always disproportionate with the time required by the conjugate gradient solver—this can be seen in figure 10. What is a larger issue are the computational kernels themselves and their CPU overhead: as our goal is to achieve running times usable for compositing, the simulation grids we use are relatively small, as well as the resulting matrix A—by far not small enough to represent it as a dense matrix, but definitely on the smaller side when it comes to use of sparse matrices today. As it can clearly be seen in figure 10, on 128×128 grid configurations, the GPU spends

5 Compressed Sparse Row Format

more time waiting for the kernels to be run than actually performing the calculations, resulting in GPU utilisation only around 17% and suboptimal performance. Fortunately, at least with larger grid configurations, the amount of required calculations increases and running GPU times come close to those of the CPU on a 256×256 grid, outperforming it on a 512×512 and larger grids as can be seen on the performance chart in figure 11 and the tables in section 3.8. On the 512×1024 grid, GPU utilisation even reaches 80%. More performance analysis will be presented in the next section.

(a)

(b)

Figure 10: Screenshots from nVidia Visual Profiler of one conjugate gradient solver iteration running on a 128×128 (a) and 512×512 (b) grid.

Originally, we expected better performance of the GPU version, greatly optimistic from the results obtained in the project [7] leading to this thesis; however, the method finally chosen does not offer such a degree of parallelisation. Furthermore, there is no way of making operations asynchronous—both calculations and memory copies—in the algorithms, every step requires the result from the previous. This is true for both the solver (to perform another iteration, the result of the previous has to be known) and the Dielectic Breakdown Model (based on the result of the solver, we have to choose a growth site, and only then can new candidates be found, matrix A updated and the potential re-solved).

The respective parts are implemented in several source files: the main logic of the shape generation algorithm, as described in section 3.2, can be found in `laplacianGrid.cpp`; all three CPU variants of the conjugate gradient solver (section 3.4) in `cg_solver.cpp` and the CUDA implementation in `cg_solver_cuda.cu`; finally the quadtree data structure is implemented in `quadtree.cpp`. Code documentation generated by Doxygen[6] is available on the enclosed CD in `/doc/perun/` both as a PDF file and HTML.

3.8 BENCHMARK

To compare performance, we ran all of the four solvers on a scale of grid configurations starting at 64×64 and ending at 512×1024. It was apparent that the regular grid Eigen solver would take too long on grid configurations 256×512 and larger so these tests were not run; the same applies for the Eigen adaptive grid version for the largest 512×1024 configuration.

Because of the shape generation process being randomised, the simulation can end after an arbitrary number of iterations. Even with the same grid configuration and η, the total number of iterations can vary greatly. Whilst an average total time could be representative after many executions, we propose an alternative metric. We run a smaller number of tests whilst measuring the average time *per iteration* for each solver. In addition to that, we also determine an average number of iterations across all solvers for the given grid configuration. Finally, these values are multiplied to express the average time per frame. As such we believe we obtain a much fairer comparison, as times per iteration are independent of the total number of iterations during the simulation and vary much less.

All benchmark tests were run on a Fujitsu Celsius W530 Power with the following configuration:

- Windows 7 64-bit
- CUDA release 7.5
- Intel Xeon E3-1231 v3 @ 3.40 GHz (4 cores, 8 threads)
- 16.0 GB system RAM
- NVIDIA GeForce GTX 750 Ti, 4096MB RAM

Benchmark results are presented in tables 1 through table 4. In figure 11 we also include a plot of the values in table 2.

6 http://www.stack.nl/~dimitri/doxygen/

Table 1: Average times per iteration in milliseconds, $\eta = 1$

Grid conf.	64×64	64×128	128×128	128×256	256×256	256×512	512×512	512×1024
REGULAR	1.500	2.668	7.011	13.494	44.540	–	–	–
EIGEN	0.407	0.916	1.781	5.473	10.138	26.322	62.575	–
PROG2	0.148	0.296	0.520	1.368	3.252	6.741	18.121	28.341
CUDA	2.203	2.323	2.543	3.801	5.364	8.259	16.644	26.026

Table 2: Average times per iteration in milliseconds, $\eta = 2$

Grid conf.	64×64	64×128	128×128	128×256	256×256	256×512	512×512	512×1024
REGULAR	2.544	3.982	10.207	26.435	61.923	–	–	–
EIGEN	0.580	1.043	1.868	6.268	11.192	26.917	69.984	–
PROG2	0.300	0.501	0.886	1.973	3.961	7.262	21.117	35.325
CUDA	3.894	3.125	3.716	4.521	6.178	8.970	18.009	28.290

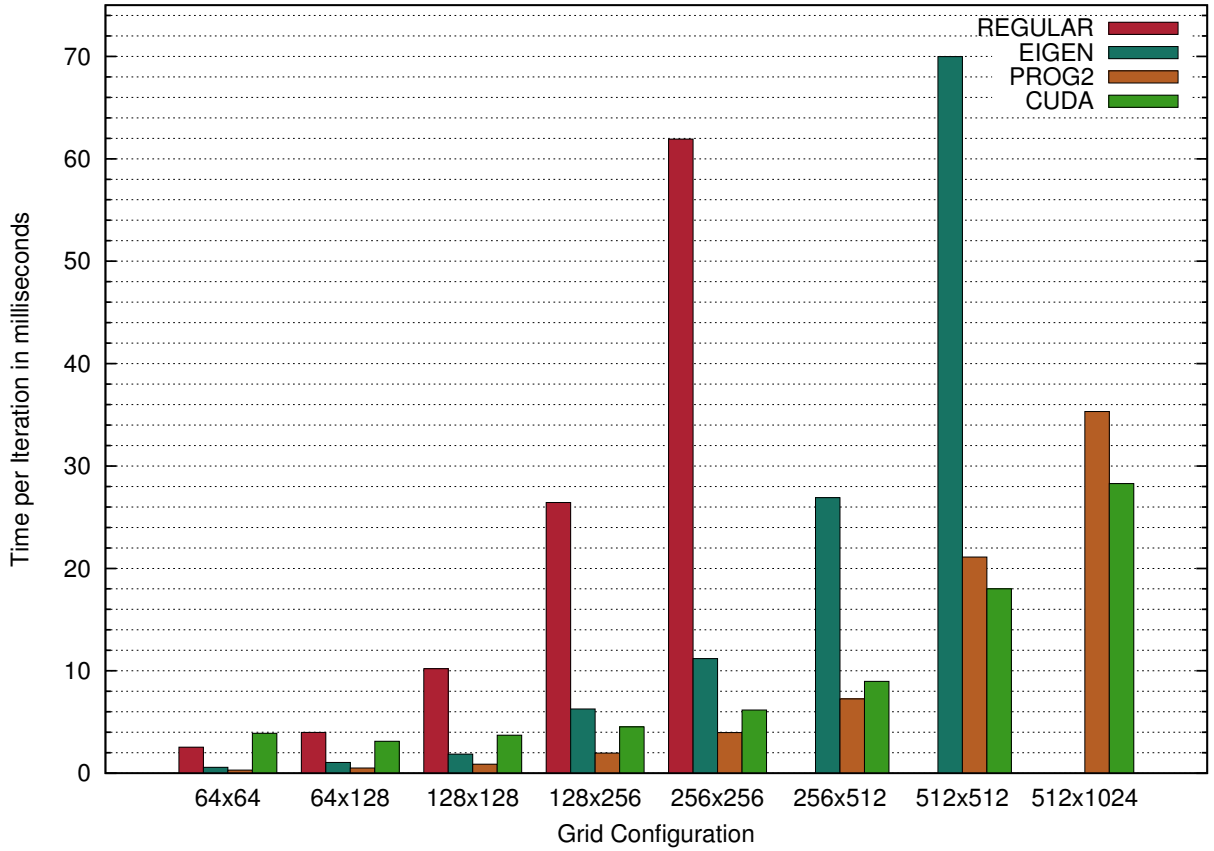


Figure 11: A plot of benchmark results in table 2—i.e. iteration times with $\eta = 2$.

Table 3: Shape generation running times per frame in milliseconds, $\eta = 1$

Grid conf.	64×64	64×128	128×128	128×256	256×256	256×512	512×512	512×1024
Av. # of it.	800	1,600	1,800	2,000	2,400	7,000	9,000	24,000
REGULAR	1,200	4,269	12,620	26,987	106,896	–	–	–
EIGEN	325	1,465	3,206	10,946	24,331	184,256	563,171	–
PROG2	119	473	937	2,735	7,805	47,190	163,092	680,177
CUDA	1,762	3,717	4,577	7,603	12,874	57,815	149,792	624,629

Table 4: Shape generation running times per frame in milliseconds, $\eta = 2$

Grid conf.	64×64	64×128	128×128	128×256	256×256	256×512	512×512	512×1024
Av. # of it.	200	350	400	900	1,000	2,000	2,700	6,000
REGULAR	509	1,394	4,083	23,792	61,923	–	–	–
EIGEN	116	365	747	5,641	11,192	53,833	188,958	–
PROG2	60	175	354	1,775	3,961	14,524	57,015	211,951
CUDA	779	1,094	1,487	4,069	6,178	17,940	48,625	169,739

From the results it can be clearly seen how the inefficiency of matrix construction for Eigen's conjugate gradient solver severely affects the performance on larger grid configurations. Also, it is apparent that the CUDA implementation only starts to become interesting with larger grid configurations where the advantages of massive parallelisation can be exploited. It is possible that transferring more of the computational process to the GPU would make the CUDA version even more favourable compared to the CPU prog2 version, especially for grid size even larger than those that have been tested. Because the goal of this thesis was to create a handy compositor's tool based on physical simulation rather than concentrating on an as precise as possible simulation, we did not pursue in verifying this assumption.

The benchmark results also show an interesting observation: times per iteration are shorter with more branching ($\eta = 1$) than when less branching is set ($\eta = 2$). We assume the reason for this could be the influence the number of growth sites has on the solver: the more growth sites there are, the less work the solver has to do—growth sites correspond to zero rows/columns in the matrix and no solving is required for them. Nonetheless, total running times are still of course higher when more branching is set ($\eta = 1$) because the total number of iterations increases significantly more than the decrease of time per iteration.

4

RENDERING

In the previous chapter, we constructed the shape of the lightning bolt which was so far represented as a grid carrying the information on whether a grid cell is a growth site or not. While this is sufficient for the Dielectric Breakdown Model, it is not enough to convincingly render the generated lightning. On the grid, we do not know the relationships between growth cells—e.g. which belong to the main channel or which merely lie on a side branch. To overcome this, we construct a graph of the lightning branching and later render the graph edges as line segments. Finally, we use convolution to add glow to the bolt. All rendering that we perform results in floating-point image data: in its nature, lightning is an effect introducing very high luminosity, much higher than even the human eye can distinguish, let alone a digital sensor. Therefore it seems logical to output high dynamic range data exceeding the visible range, letting the compositor determine tone mapping within a given scene. The renders from the standalone application are saved as EXR images.

4.1 BOLT SHAPE GRAPH

In our implementation we use a *rooted tree* to record the branching. Each node contains a pointer to its parent, an `std::vector` of children, an id, value of the x and y coordinate of the vertex, branch type and assigned luminance. We do not pose any restrictions on the number of children; from the grid configuration we use, a parent could theoretically have up to 7 children, but practically 1, 2 or 3 children mostly occur. In the shape generation algorithm, when a new growth site is selected, its parent has to be determined by searching the cell's 8-neighbourhood. There is no explicit way of knowing which of the neighbouring sites should be the parent if more than one growth sites are incident, but due to the lightning shape we seek to produce, north, northwest and northeast parents are given priority. In the standalone version (independent of Nuke), we also introduce the *resolution multiplicator*—a crucial parameter that influences rendering. Originally, growth sites bear their simulation grid x and y coordinates, but we would like to stretch the simulation grid and instead of rendering the cells, draw lines between the cells' coordinates. Therefore, the original coordinates are multiplied by this parameter. A reasonable multiplicator is 4, but even 8 produces feasible results in some cases. At this point we also introduce *jitter*—just magnifying the simulation grid would create visual artefacts based on its uniformity. Therefore

Figure 12: A sample render, shape generated on a 256×256 grid, $\eta = 1$, resolution multiplicator 4.

instead of taking only the original coordinates, we modify them in each direction by a random value between -0.5 and 0.5; in relation to the simulation grid, this can be considered as randomising the graph node's position inside the original grid cell.

4.2 FINDING BRANCHES

Within the graph, we would like to assign different luminance values to different branches. Obviously, the main channel should have a much larger luminance than the side branches, even by an order of magnitude. We divide branch types into 3 categories: the first being nodes along the main channel, the second nodes along side branches that fulfil branch length criteria (discussed further) and category 3 for all other nodes. Determining graph nodes along the main channel is straightforward. Once a destination site is reached in the simulation and the simulation is terminated, all that has to be done is to traverse the graph back up to the root from that site via the nodes' parents.

Setting side branches requires a more elaborate approach. Photographs of lightning show that some side branches are much stronger than others. Kim and Lin state that it is not clear as to how these brighter branches are selected in reality, yet they propose that in the simulation the longest side branches should be selected. Furthermore, we add a user modifiable cutoff variable which determines

a minimum length of such a type 2 branch. However, finding longest branches actually involves performing a blind depth-first search of the whole branching tree. As inefficient as it may seem, the graphs generated are actually relatively small: on a 256×256 grid the tree might contain around 1200 vertices. In any case, finding side branches does not pose as a computationally expensive step compared to the shape generation itself.

Finally, we introduce luminance decay. It can be seen even through the study of photographs, that side channels fade away as they get further away from the main channel. While this might not be physically accurate, we set a linear decay for every branch—starting with the luminance set for the given branch type or the luminance of the parent branch at the branching point (whichever is smaller) and gradually ending at zero. Adding decay greatly increases believability and we are surprised that this feature hasn't been discussed in any of the papers referenced in section 3.1.

4.3 DRAWING LINES

In the real world, the diameter of the lightning bolt's main plasma channel is only several centimetres[29]. Kim and Lin suggest that humans perceive the channel of light as much thicker because its luminosity greatly exceeds the capabilities of the human eye. Another factor could also be light scattering within the atmosphere, which is bound to happen in almost all cases of thunderstorms.

Nevertheless, we still choose to begin the render with thin lines. For that, we use Xiaolin Wu's line drawing algorithm which produces one pixel wide anti-aliased lines. From our experiments we conclude that it is not necessary to draw thicker lines: in most scenes, the composited lightning stroke will only be several pixels wide at a maximum; that can easily be attained by rendering the lines with a higher luminance and later adding blur.

4.4 ADDING GLOW

Both Bickel et al. and Kim and Lin use the so called Atmospheric Point Spread Function (APSF) to add glow to their renders. Interestingly, Narasimhan and Nayar introduced the APSF as a method to *remove* glow around light sources in images, induced by multiple scattering of light in bad weather such as fog or rain[22]. Their goal was to recover the shapes and depths of light sources in the image and in turn clear the image of these undesired atmospheric effects, which could greatly help e.g. in automated image processing. They did however suggest that the method could also be used to simulate weather effects and show use thereof in their paper.

Based on a set of weather parameters, the Atmospheric Point Spread Function produces a convolution kernel to be convolved with the original image. (In our case, where we aim to *add* glow; in the original case, the kernel would be used to *deconvolve* the original image.) As physically accurate as the APSF may claim to be, we did not gather any benefit in its use. At first we experimented with its application but did not achieve significantly better results opposed to applying a series of simple Gaussian blurs. In principle, the APSF kernel is nothing more than a sophisticated blur. Furthermore, in order for the APSF to provide visually pleasing results, rather large kernels have to be used, which is computationally expensive concerning the convolution. Also, the input parameters of the APSF are not as intuitive as they may seem at first glance.

Therefore, we stayed with simple Gaussian blurs and merges with the original image. Motivation for this was also the fact that Nuke offers a straightforward efficient implementation of Gaussian blurs: not exploiting it for the plug-in would be regretful. But most importantly, such blurs introduce very intuitive control over the result—for every blur, only two settings are used: the size and subsequent mixing with the original. More on this will be presented in the next chapter on Nuke.

Another important feature that has not been mentioned in the papers researched in section 3.1 is adding grain. Even when creating full-CGI movies, grain gets added to the renders in post-production, because we are just used to seeing it there. Even with the best digital sensors today, noise is always present in the pictures acquired, both still or moving. With moving pictures this is even more profound as we clearly notice the noise as it changes in each frame.

4.5 SCENE ILLUMINATION

Many of the methods discussed in section 3.1 set their purpose to accompany computer generated imagery, in which case it is at hand to exploit scene information (usually 3D) to attempt to introduce accurate and at the same time efficient scene lighting. While this is of course a valid path and a field of research seeking innovation, it is still a somewhat idealistic presumption that such methods would be used in any non full CGI scenes in today's post-production pipelines. There are several arguments to support this proposition.

First and foremost, such scene information is not available in most cases. Many assets used in 2D compositing are in fact CGI and their renders may also include depth passes; Nuke itself can even work with the original mesh through the 3D framework it ships with. However, most shots an artist will be working with will be shots captured by a cinematic camera, which of course do not include any other information apart from pixel intensities across the captured image plane.

In such cases, traditional compositing techniques have to be applied, all based on experience and know-how of the compositor.

This may possibly change in the future with the proposed Lytro Cinema[1]—Lytro have been offering light field still-image cameras for several years and last year announced their entry into the cinematic world. At this year's NAB show, they presented a prototype of their camera and a teaser for their short film *Life* they are filming to exhibit the technology. Light field cameras, or plenoptic cameras, capture both the intensities and directions of light rays incident to the image sensor. This in turn makes it possible to extract depth information from the recorded data, or even make a basic 3D reconstruction of the scene. Lytro showcase this on creating a depth-matte, showing how their technology could even obsolete the use of greenscreens. It is of course still to be seen how this technology will influence the film industry and to what extent it will be adopted in the future.

Another interesting novelty is deep image compositing: instead of just accompanying CGI renders with depth maps, pixels also carry render information at more given depths along the z-axis. Such information could of course help realistically illuminate a scene by lightning. Although Nuke readily supports deep image compositing operations, it is still an emerging field, today adopted only by large VFX houses in their production pipelines.

With the lack of scene information, masking and keying is still king. As far as automatic methods and algorithms may go, they still rely on the artist's input, at least in film post-production: this may be as simple an operation as visually checking the results of a render and modifying the settings for a next attempt. Analysing the artistic properties of an image is just something still impossible to achieve with today's computer technology.

But back to lightning. Lightning as such is very quick. Concerning real-life atmospheric lightning, it is an effect that will find its place on just a few frames of the shot. The viewer will register the lightning bolt, and that it illuminates certain parts of the scene, but will never question the details of that illumination. They will concentrate much more on the dynamics and timing, and on the mood the effect creates in the scene.

If we take into account the science fiction use, lasting much longer, the desired effect on scene illumination is actually a relatively rapid variation of lighting from frame to frame. Because of the longer time the scene is exposed to such illumination, much greater care has to be taken when creating the masks (and possibly animating and transforming them by tracking elements in the original scene) as the viewer could quickly notice imperfections—but that is a basic necessity in any compositing work. In many cases though, it is even sufficient to avoid masking whatsoever, only varying gain over the

1 https://www.lytro.com/cinema

whole image (e.g. with a ColorCorrect node in Nuke), simulating the effect of the lightning illumination spilling over the entire image sensor. Such a simple approach was used in the sci-fi example included on the CD in `/video/scifi/`.

In the other example scenes shown here and on the enclosed CD, we use a combination of masking and keying to determine the area that is to be illuminated. In the case of real-life lightning, the scenes into which lightning is to be added in post-production will usually include dramatic clouds with varying luminosities—which is ideal for pulling some sort of key to influence cloud illumination.

5 THE FOUNDRY'S NUKE PLUG-IN CREATION

Today, The Foundry's Nuke is de-facto the industry standard in digital compositing. The Foundry's Nuke is commercial software, selling at £2,570 per year for the basic version, going up to £5,655 for Nuke Studio. Since April 2015, Nuke can also be run for free in non-commercial mode with certain restrictions. In respect to this project, one of these is unfortunately no 3rd party plug-in support. For development and testing, access to commercial Nuke licence has been made possible by R.U.R. studio in Prague.

Nuke projects (actually called scripts, as they are saved as human readable, and even alterable, plain-text files) consist of node-based workflows (unlike e.g. Adobe After Effects which is layer-based). In Nuke, *Nodes* are objects visible to the user that represent image processing operations and build up the Node Graph/Direct Acyclic Graph (DAG) of image operators. Nodes have input and output arrows to connect to other Nodes in the DAG to achieve the desired visual effect. An example of a very simple node graph can be seen in figure 13. Figure 18 shows a zoomed out view of a slightly more complex node graph, one actually used to composite lightning generated by the developed plug-in into a scene. (Practical use of the plug-in is teared down in the next chapter.) Still, that node graph is relatively small—it is rarely the case in post-production that adding such an effect would be the only operation necessary.

A *Node* consists of one or more *Operators* (Ops). Operators do the actual image processing, these operators can be seen as the classes implemented by a plug-in developer and loaded into Nuke as shared libraries.

The settings that appear in the properties panel in Nuke for each node are called *knobs*. Apart from obviously being used as controls to set parameters for the Ops, they can also be used just as simple data storage. (How we make use of this is further explained in section 5.3.)

In this chapter we describe how we develop the plug-in and an accompanying *gizmo* (a *gizmo* is a node which internally packs a whole part of the node graph, explained in section 5.4).

5.1 OVERVIEW

Several different options exist for plug-in creation. The most straightforward would be Python scripting, ideal for short tasks that do not require much processing power. On the other side of the spectrum lies the Nuke Developer Kit, a C++ API allowing developers

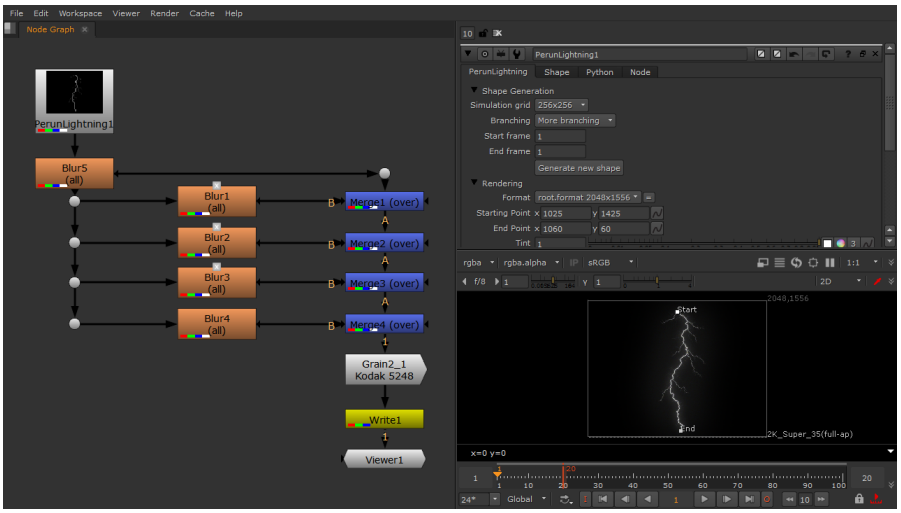


Figure 13: A screenshot of Nuke and the PerunLightning plugin in a configuration generating a similar render as shown in figure 12.

to create Nuke plug-ins of far larger complexity and allowing to exploit the inner workings of Nuke. Another interesting approach is the OpenFX API[1]—an open C standard for visual effects plug-ins, making it possible for plug-ins written to the standard to be used in other applications supporting OpenFX—apart from Nuke, these are Blackmagic Fusion or Assimilate Scratch, to name a few. A recent addition to the mix is Blink script, another scripting interface available in Nuke.

For this project, the Nuke Developer Kit seems to be the most obvious option and we indeed pursue in that direction, motivated by the tight integration with Nuke as Nuke is our target platform. We discuss details about NDK in the next section; it would be unfair though not to investigate other available options.

A popular choice for commerical VFX plug-in developers is OpenFX, as it is then possible for them to sell their products to a much larger audience with very little extra effort. A leading example can be the effects bundle Sapphire by Genarts[2]. OpenFX was originally developed by The Foundry and released under a BSD licence. Since 2009, it has been managed by The Open Effects Association, a non-profit organisation, whose goal is to "develop and promote open standards across the visual effects community" [1]. There is no doubt that the plug-in could also be developed as an OpenFX plugin. The API reference [23], although somewhat crude at first sight, does provide the necessary information and The Open Effects Association add a Programming Guide for successful plug-in development. Apart from that, some tutorials do exist online to aid OpenFX plug-in creation.

1 http://openfx.sourceforge.net/
2 http://www.genarts.com/sapphire/ofx

A new addition to Nuke 8.0 was the Blink API and Blink script—this is a very interesting approach for users with non-programming backgrounds, as it efficiently parts the user from the underlying technical concepts while still using their full potential. Blink scripts are written in a C-like style and because of the relatively small application space—essentially image processing—Blink kernels are parallelised and compiled on the fly and depending on their type and the availability of resources (multi-core CPU or an OpenCL enabled GPU). Because this is a relatively new feature, the user base and information regarding Blink script is rather sparse, but, apart from the Blink API reference document [9], The Foundry do offer a collection of sample kernels[3] to get started with. With the recent wide availability of CUDA/OpenCL capable GPUs, GPU parallel computing is finally finding its way into common commercial software, a notable example being NUKE itself, or the Cycles rendering engine, part of the free and open-source 3D animation suite Blender. Therefore, we believe Blink scripts may gain on popularity, because achieving parallelisation with a Python script is near-to-impossible, and programming plug-ins in C++ requires much more extensive programming knowledge (even more so if parallel computing is in mind) and recompilation for each platform and new version of Nuke.

The last to be mentioned is the Nuke Python scripting API. Undoubtedly the Python scripting API is a very powerful tool, but its main application domain lies elsewhere: automatised node graph manipulations, altering nodes' settings or as import/export scripts. Although probably not impossible, it would be rather ineffective to create a rather complex image generating plug-in utilising the Python API.

5.2 NDK C++ API

For a comprehensive overview of all options the API offers, the user is of course directed to the NDK Developers Guide[10]. The API also comes with a collection of example plug-ins which are crucial for better understanding of the workings. In this section, we attempt to summarise the parts of the API that are important for our implementation. Implementation details are then provided in section 5.3

Work of operators can be typically broken down to four main phases: *creation*, *validate*, *request* and *engine*. In the *creation* phase, an instance of the op is created, the `knobs()` function is called to construct user knobs. These operations are inherited from the `Op` class and no action has to be taken by the programmer except for defining the knobs.

When a user connects the node in the node graph, the op's `_validate()` function is called. The main goal of an implemented `_va-`

3 https://www.thefoundry.co.uk/products/nuke/developers/90/BlinkKernels/

`lidate()` function is to set up `IopInfo`: records such as the image format, bounding box, channels etc.

Nuke implements a so called "pull system"—operators' functions are only called when requested by nodes further down the node graph—there is no need to do any image processing when no *Viewer* or *Write* node (that would actually be rendering at the time) is connected later in the graph.

Such nodes will call `_request()` on their inputs, and these are in turn responsible for recursively calling `_request()` on their inputs—but possibly with different parameters. The purpose of the `_request()` function is to define a region of interest—a *Viewer* node may request a single image row of its input, that being e.g. a *Blur* node. In order for the *Blur* node to provide correct output though, it requires several more rows above and below the image row that is to be rendered, depending on the size of the convolution kernel. Therefore it would also request these rows from its input.

Finally, if we are to actually produce image output, the `engine()` function of the op is called. If knob values change, the whole *validate-request-engine* sequence is repeated.

As indicated above, the fundamental image processing unit in Nuke is a *row*, simply a row of pixels in an image plane. This has been chosen by Nuke developers to make implementation of multi-threading easy—in a standard scenario, the `engine()` function of an op would be called in many worker threads for different rows.

As we have described in chapter 4, the heart of the rendering process for our lightning generator is *line drawing*, which does not really correspond with Nuke's concept of using rows as processing units, as using such an approach would be extremely ineffective. Fortunately Nuke offers more base classes with specific configurations from which ops can be derived, in our case we derive the op class for the plug-in from `PlanarIop`. The `PlanarIop` class replaces the `engine()` function with `renderStripe()` which gives the developer access to the whole image plane.

Furthermore, for the plug-in, we require an option to run shape generation independently on rendering—only when the user chooses to do so. For that, the NDK offers the `Executable` class to derive from. When a button is pressed in the UI, the execution process commences. The developer then has three important functions to implement: `beginExecuting()` which is called first, `execute()` which is called once for every frame in the desired frame range (setting the frame range is best done with a Python script—using a *PyScript knob* to retrieve other knob settings), and finally `endExecuting`, called before the end of execution.

The Perun plug-in was developed with the Nuke 9.0v8 API; as of May 2016 Nuke 10.0v1 has been released—fortunately, according to the developers guide, nothing changes with respect to plug-in de-

velopment. Although not tested as thoroughly, the plug-in can be compiled against and works in the new current version of Nuke.

Should the reader wish to try out the plug-in, a 15 day trial of Nuke with no limitations may be obtained from The Foundry.

5.3 IMPLEMENTATION

From the benchmarks presented in section 3.8 it is apparent that the CPU implementation of our own prog2 solver indisputably outperforms the others on smaller grids. The Cuda version does slightly precede prog2 with larger grids, but, from a practical perspective concerning the plug-in, we expect much more frequent use of smaller grids where computation time does not have to bother the artist and slow them down in their work.

Building Nuke plug-ins with Cuda acceleration is possible, but also absolutely undocumented. It is no surprise that this is not covered in the NDK documentation, but we would have expected to find at least a single post on the topic on a bulletin board elsewhere on the Internet—but no. Nevertheless, in the project leading to this thesis, we did look into this and managed to compile a plug-in using Cuda. The main complication lies in the fact that Cuda code has to be compiled with the Nvidia Cuda Compiler (`nvcc`) but Nuke plug-ins have to be built with Visual Studio 2010 on Windows or gcc 4.1 on Linux. In order to create a functional Cuda-accelerated plug-in, it is necessary to compile the Cuda part independently with `nvcc` as a separate shared library, functions from which are then called from the actual plug-in library, compiled to be compatible with Nuke.

Because we did not find any benefit in doing so, we decided not to pursue in adding GPU-acceleration to the actual Nuke plug-in and only use our prog2 solver for shape generation.

Most of the application structure remains similar to the standalone version, although several elements had to be reorganised. The main difference is the greater separation of the shape generation process and the rendering process. Shape generation is computationally expensive and is best only performed once, whereas changing render settings, such as channel luminance or tint, can comfortably be done in real-time for the same shape. Therefore only changing the grid configuration, branching setting or frame range requires new lightning shapes to be generated. (*Shape generation* section in the plug-in/gizmo UI as shown in figure 13 or 14.)

As indicated above in the previous section, our plug-in class is derived from two base NDK classes: `Executable` and `PlanarIop`. The former is used for shape generation and the latter for rendering.

We keep information about the generated shape internally within the Nuke script. For that we utilize the Table knob: as the name suggests, such a knob stores a table (2D array) with predefined columns.

We create two Table knobs, one for shape vertices, and the second for shape edges.

Storing shape information is very important: once a shape is generated and the artist is satisfied with it, there is no reason whatsoever to generate an alternative. Moreover, imagine the client approves the scene or perhaps just the shape of the lightning itself—it *has* to remain exactly the same as it was approved, even if some other subtle changes are requested to be made.

For rendering, instead of using our own class to manipulate with floating point image data, we render directly to an instance of Nuke's internal `ImagePlane` class.

Internally, Nuke makes heavy use of caching, so that image processing does not have to be repeated if nothing changes. Therefore, every op utilises a hash, which changes if the hash of any node connected to its inputs changes, or if any knob setting does. Special care has to be taken with knobs that only provide data storage such as a Table knob, as they do not automatically affect the hash—this has to be done by the developer with the `hash.append()` function.

A feature we added to the plug-in as opposed to the standalone version was the selection of a start and end point for the lightning. For that, the NDK offers a practical *XY knob*, which also shows the points in the viewer and allows the user to move them—a screenshot is shown in figure 18. Implementation-wise though, this required us to construct our own 2D transformation matrices to transform the generated shape to the position set by the user.

5.4 NUKE GIZMOS

Nuke also allows users and/or developers to create so-called *gizmos*, collections of nodes with a specifically defined sets of controls. In principle, a gizmo is a cut-out of a Nuke script/node graph to be replicated within the same script or be used in other scripts. The creator can define which of the aggregated nodes' controls should appear in the gizmo's toolbar. From a developer's point of view, this can be regarded as a restriction to artists in order for them not to misuse the intended features of the gizmo.

Creating a gizmo is pretty straightforward: first, the user selects the nodes in the node graph which are to form the gizmo and makes them a group (Other > Group from the Toolbar of Ctrl-G). This substitutes the selected nodes with a new node. The group of nodes this node represents is in fact exactly the same as the gizmo—but creating a gizmo makes it possible to also replicate it in different scripts. It is of course important to add the desired controls to the gizmo's UI: the user can select any of the controls any of the contained nodes offer, or even add new ones, and regroup them as desired in their new UI.

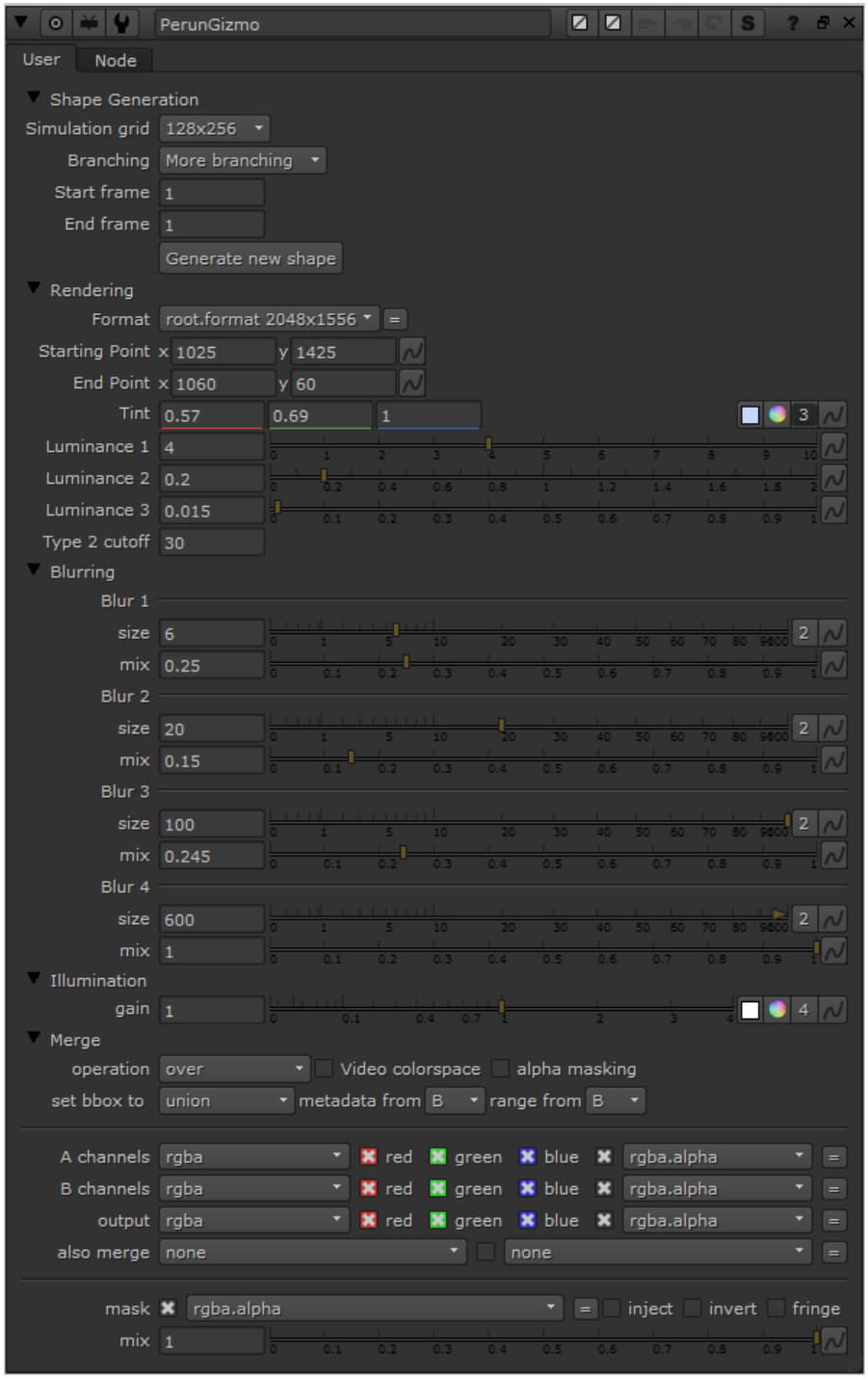


Figure 14: A screenshot of the gizmo controls.

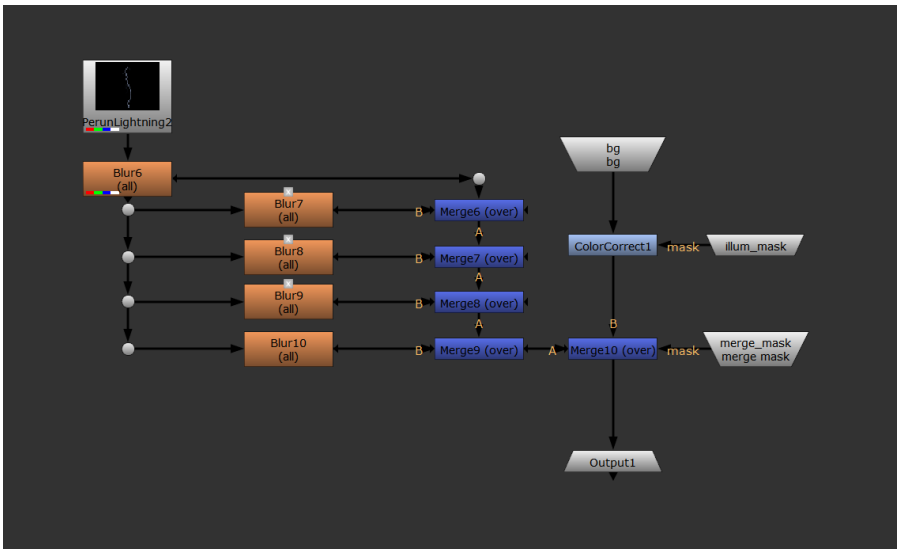


Figure 15: The internal structure of the gizmo.

Figure 15 shows the internal structure of the gizmo created for the plug-in. The main pipeline for the plug-in itself is the series of Gaussian blurs with various settings. Their default settings are shown in the *Blurring* section in figure 14—such settings produce renders similar to the one presented in figure 12 in the previous section. The user can of course modify these to suit their scene. The gizmo also offers three optional inputs: a background image, a background illumination mask and a merge mask. As can be seen in the gizmo's node graph in figure 15, these are of use only when a background image is connected. Concerning illumination, the user can increase or decrease gain on the background image—the *gain* slider in the *Illumination* section of the gizmo controls maps to the master *gain* slider of the *ColorCorrect1* node. If the background illumination mask input is connected, it is automatically applied. Finally, the *Merge10* node controls merging with the background image, again with an optional mask. Because this is a crucial operation, we leave all the *Merge* node's settings available to the user in the *Merge* section of the gizmo's UI.

Although the gizmo packs many convenient features in one node, we still prefer to work with the sole plug-in, but that probably stems from our detailed knowledge of the inner workings: we feel more than comfortable modifying the settings in our scripts directly to suit our needs. In addition to that, in general work with Nuke, we tend not to use integrated node features that incorporate functionality of other nodes available in Nuke; we prefer to stick to a workflow that clearly exposes the process in the node graph. It is of course just a personal preference though. An artist using the plug-in for the first time may on the other hand appreciate the basic functionality of the gizmo.

5.5 USABILITY TESTING

After creating a working prototype of the plug-in, we presented it to several other compositors at the R.U.R. studio for comments. A few points on what could be improved or added emerged from the discussion.

Probably the most crucial feature not present at the time was setting the start and end point of the bolt. This feature was added and the implementation has already been described above. When using the plug-in in the compositing process, this indeed proved to be a very important and practical element.

In Nuke, it is common practice that even image generating nodes have an optional input and basic control of its merging with the original image. Implementation of such a feature was attained by adding a merge node and its options to the accompanying gizmo—also already described in the previous section.

Inspired by the capabilities of the Sapphire Zap lightning plug-in, it was suggested that apart from a start and end point, the lightning shape could follow a designated curve. One approach to do so would be to actually incorporate the curve into the simulation grid to attract the shape being generated; a probably much simpler approach would be just warping the generated image—which also seems to be the case within Zap effect. This suggestion was not deemed as crucial so it has not been implemented.

Users also expressed concern working with frame ranges offered by the plug-in—if the frame range changes, lightning shapes for all frames are generated anew. This behaviour stems from the standalone application; it is of course not always necessary to generate new shapes, even if the frame range is expanded—we can still use the last frame's lightning's shape graph as a basis for subsequent frames. This modification though has not been incorporated into the plug-in yet.

Instead we at least included the feature that the generated shape stays valid outside the set frame range—the shape of the last valid frame becomes frozen. This particularly comes in handy when compositing classic atmospheric lightning, where it is sufficient to generate no more than one shape and only animate the luminance knobs within the few frames the bolt is to be shown in. Example use thereof is presented in the next chapter.

6 RESULTS

In order to demonstrate the results that can be obtained by using the plug-in, we selected three different scenes to prove the plug-in's wide scale of application. In the first scene, *Sky*, presented in section 6.1, we exhibit the use of the plug-in to simulate atmospheric cloud-to-ground lightning. The second scene, *Sci-Fi*, teared down in section 6.2, shows the plug-in in a science fiction set-up, adding artificial lightning to a scientific experiment. Finally, we show how the plug-in could be used in a company jingle[1]. That is described in section 6.3.

The actual Nuke scripts used to make the compositions are included on the enclosed CD in `/nuke/` together with the original footage, the resulting videos are in `/video/`.

6.1 SKY

As the first example, we choose to present artificial lightning in its original form: atmospheric cloud-to-ground lightning. We use footage of an actual storm recorded on video and attempt to composite in lightning generated by the Perun plug-in. The *Sky* scene shows footage where two lightning strokes occur—the first is the one generated by the plug-in, the second stays as recorded on video. In figure 17 we show a frame by frame breakdown of both the original and the composited effect.

In the original footage it is interesting to see how the stroke is visible at some intensity for as long as 7 frames, although that is only true for the most prominent lightning occurrences captured. During this storm, we recorded more than an hour's worth of video and captured several lightning strikes; most of them though were shrouded by clouds—only the main channel would be visible and it would mostly show up on just one frame. Even though 7 frames seem very long, it does correspond with the physics research presented in chapter 2, where we state that the duration of the flash can be as long as 200–300 ms. The footage was recorded at 30 frames per second, which corresponds to roughly 233 milliseconds for 7 frames. Concerning the bolts which were visible for a shorter amount of time, two explanations are at hand: they did not consist of that many return strokes, therefore being visible for a shorter time, or, the intensity

1 A short sequence used in advertisements to present the company; although a *jingle* usually refers to sound, in the VFX industry it is also used to describe an analogical video sequence, usually incorporating the company logo

Figure 16: Composited generated lightning (left) next to lightning captured on a digital camera (right).

of the return strokes was not high enough for the light to propagate through the clouds present between the stroke and the camera.

This example does reveal a limitation of the plug-in—as such, it is not possible to generate very long side branches. In some cases of real-world lighting, such as the one presented, side branches span even wider than the lightning's visible height. Yet, not all lightning bolts include side branches which are so long; furthermore, on video, these would only be visible for one frame anyway—the viewer would have to concentrate remarkably to actually knowingly witness them.

Moreover, Perun generates new random shapes in every run, figure 16 depicts yet another generated lightning bolt, this time seeming even less artificial compared to the real bolt captured on video.

(a)

(b)

Figure 17: A frame by frame breakdown of real-world lightning (a) and CGI lightning generated by Perun and composited into the scene (b).

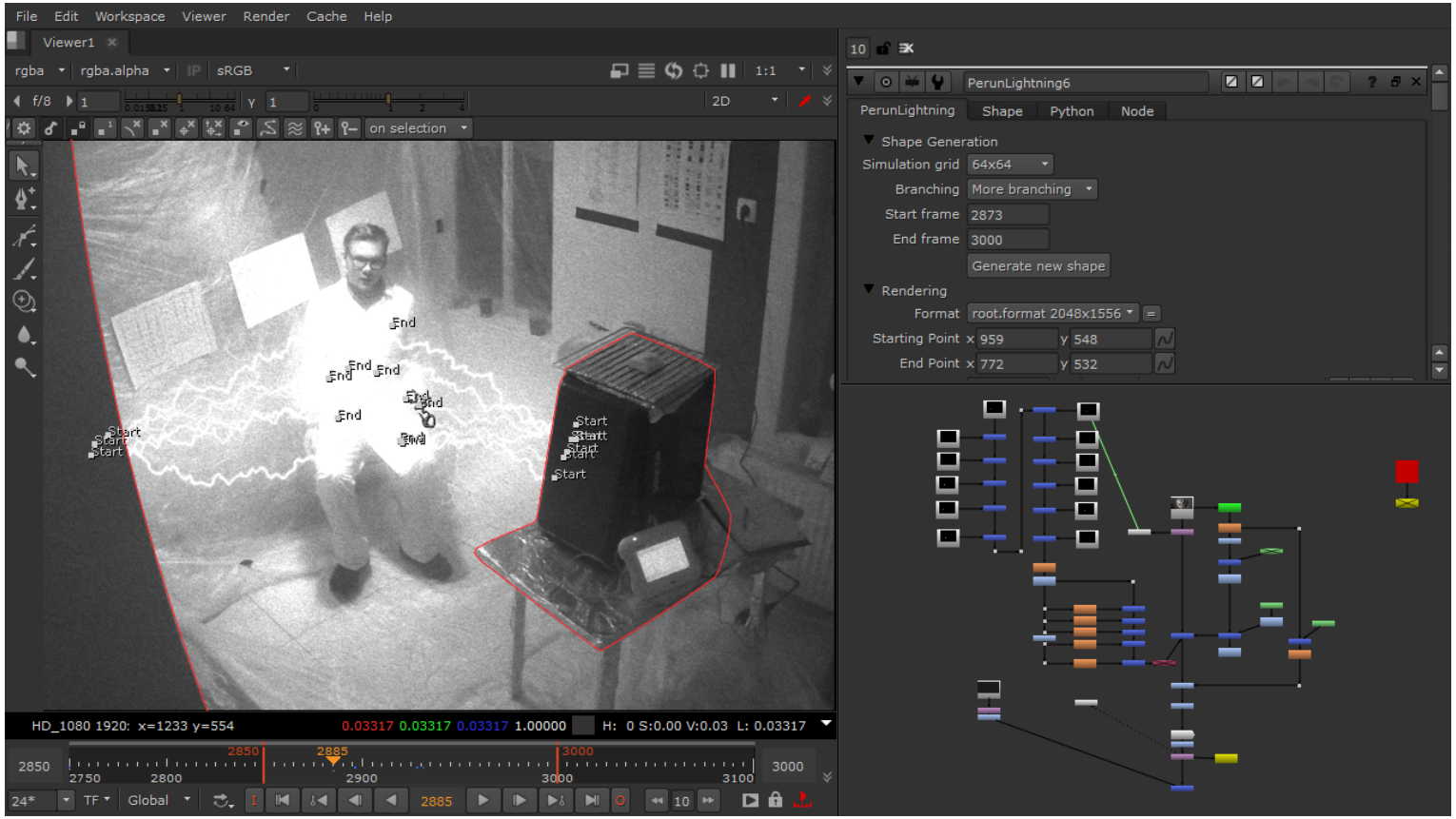


Figure 18: A screenshot of the compositing process for the sci-fi scene. Notice the use of masks and start-end points for the lightning.

6.2 SCI-FI

It is amazing to see how many today's film productions actually use lightning-like effects in science-fiction settings. It is indisputable that such an effect has its place in the science-fiction world and therefore it is imperative also to seek use for the plug-in therein.

The showcased scene itself originates from a boy scouts' camp, where it was recorded as part of the symbolic framework. Within the narrative, it attempts to depict a scientific experiment aiming to genetically enhance the subject. The children did not see the experiment itself, but just the footage as if from a security camera in the laboratory.

At the time of making, the Perun project was not production ready, therefore an alternative approach had to be chosen: an obvious choice was Advanced Lightning in Adobe After Effects. Now we seek to recreate the lightning effect with the use of the Perun plug-in. Both versions of the video are included on the enclosed CD in `/video/sci-fi/`: `scifi_render_ae.mp4` is the original After Effects version, `sci-fi_render_perun.mp4` is the new comp in Nuke with Perun lightning.

In the composition, several instances of the plug-in are used, as depicted in figure 18. Two simple masks are used to designate the area where the lightning effect is to be seen. Although the lightning itself induces some glow to the scene, it is enhanced by varying gain across the whole scene, as discussed in section 4.5.

Apart from illuminating the whole scene, it is also desirable to depict the effect of the subject being "electrified" in the experiment. For that, we used a simple luminance key on the subject's white lab coat together with a rough garbage matte to mask out the surroundings. This serves as a mask to randomly increase and decrease gain on the subject resulting in glow, supposingly originating from the energy being "pumped" into the body.

The After Effects version may seem deliberately scamped, but this is definitely not the case. The video sequence was created prior to the Perun project being usable in production, but already with the knowledge of and experience with compositing, and with respect to research on lightning and its simulation.

It shows the application of the Advanced Lightning feature of Adobe After Effects, attempting to exploit the branching features it offers. The amount of branching can be set, but its animation cannot be controlled that well—therefore the effect seems to end in the middle of nowhere in some frames. It is true that more distinct bolts with less branching could also have been generated in After Effects in a similar way as the Perun plug-in was used; nonetheless this proves that facilitating branching towards the lightning's target (a feature that cannot be easily simulated by physically based methods as discussed in section 3.1.4) is not necessarily such a useful setting.

Figure 19: A screenshot of the plug-in being used for a company jingle.

6.3 JINGLE

Lightning symbolises energy, electricity in particular. Therefore we also attempted to incorporate the generated lightning into a jingle for a company in the field of energetics. For that, we chose to experiment with the logo of ČEZ, a. s.[2]. In their current jingle, used mainly as the finale of their advertisements, ČEZ simply fade in their logo and accompany it with text, also faded in.

As it is visualised—an integral square—the logo can be seen as a sign to be illuminated—and that is what we seek to do with the generated lightning. In the composited scene, the glow induced by the lightning actually illuminates the logo, as if lighting it up from behind. In some of our original attempts, we also tried a version where the lightning would reach the white "E" from the front, going over the orange background, but instead decided to pursue further with the version a snapshot of which we present in figure 19.

2 The logo and trademark are the property of ČEZ, a. s.; the logo was created by Studio Marvil, s. r. o. We do not hold any affiliation with either of these companies and have chosen their graphics solely as a case study.

7 DISCUSSION

We created a functional lightning plug-in for Nuke which we believe could even be used in professional production as it does not show significant visual shortcomings compared to other commercially available products.

Its physically based nature though does make it lack a certain degree of flexibility that visual methods may offer—controlling branching specifics, such as where we would like the branches to go (this may be a very rare requirement though), is rather difficult and the user would have to generate new shapes until one that satisfies their expectations appears.

Also, compared to visual methods, another downside of a physically based approach is computational complexity. Although we manage to achieve running times that do not distract the artist ever so much, these can still not be compared to visual simulations which can, in some cases, be close to real-time, if not even being real-time on modern hardware.

An interesting experiment would be to combine our physically based method with elements used in visual methods (section 3.1.2). Our proposition is to generate basic shapes with little branching on small grids (for the process to be fast) which could later be combined together to create more branching, possibly dependent on user input where the user could effectively control the characteristics of the branching. Simulation on small grids though would not generate sufficiently detailed "electrical tortuosity", but this could be easily added via the fractal methods also discussed in section 3.1.2. Like this it may be possible to generate even very complex shapes, but with much faster running times. However, such an approach could not be considered as completely physically based.

In the thesis, we do not mention sound although it is an inseparable effect accompanying lightning. The Perun project simply focuses on the visual side of lightning—mainly because sound and video post-production would be dealt with in completely different stages of the post-production pipeline: compositors rarely have sound data available to them. Another reason is that with atmospheric lightning, the accompanying thunder is all but synchronised with the lightning due to the differences in speed of sound and light. Science fiction lightning on the other hand would not use a thunder sound effect at all, but instead some sort of an a electrical "buzzing" sound.

Although Nuke is the industry leading solution for compositing, its pricing puts it into the solely professional segment of the market—

many smaller studios or freelancers may rely on cheaper solutions that also allow them to do compositing work. Smaller studios or freelancers would probably be the main target audience for a plug-in such as ours, as they would not commonly have access to other means of generating lightning. Therefore also adopting Perun as an OpenFX plug-in or as plug-ins for other compositing programs could significantly widen the potential audience that could make use of the lightning generator.

8 CONCLUSION

In the thesis, we created a physically based lightning generator to be used in film/video post production, both as a standalone application which produces EXR images and a plug-in for the Nuke compositing program.

First, we presented a theoretical insight into how lightning is formed in the real world—in the atmosphere—and described its properties and types. We continued with a survey of existing methods of lightning simulation in the field of computer graphics, both visual and physically based.

As a result of this research, we chose the Dielectric Breakdown Model to build our lightning shape generation process on and provided an in depth description of the approach, together with the specifics of our implementation. We implemented four variants of a conjugate gradient solver, including a CUDA version running on the GPU, and performed benchmark tests to analyse their effectiveness.

Next we discussed rendering of the generated electrical bolt taking into account atmospheric effects. We further dealt with a practical approach to scene illumination induced by generated lightning when scene information is not available.

We provided an insight into The Foundry's Nuke itself as a compositing program as well as an introduction into Nuke plug-in creation, focussing on the particularities of the Perun lightning plug-in's implementation. After developing a prototype plug-in, we performed an informal usability test based on which we added several more features to the current version.

Finally, to demonstrate practical use of the developed plug-in, we used it to composite lightning into three different scenes showing the plug-in's adaptability. We also provided a video comparison with real atmospheric lightning.

A IMPLEMENTATION DETAILS

A.1 STANDALONE APPLICATION

A.1.1 *Files and Classes*

- `cg_solver.cpp` and `cg_solver.h`
  - class `CGSolver`, the class which implements solvers using the method of conjugate gradients
- `cg_solver_cuda.cu` and `cg_solver_cuda.cuh`
  - class `CudaSolver`, the Cuda implemetation of the method of conjugate gradients
- `common.h`
  - definitions of terms and constants used within the whole program
- `graph.cpp` and `graph.h`
  - class `DAG`, the node of the graph data structure to store the generated lightning shape
- `imageFP.cpp` and `imageFP.h`
  - class `ImageFP`, the class for manipulation with with floating-point images.
- `laplacianGrid.cpp` and `laplacianGrid.h`
  - class `LaplacianGrid`, the main class which holds all the neccessary methods and variables to simulate the Dielectric Breakdown Model / Laplacian Growth
- `perun.cpp` and `perun.h`
  - main project file and header with `main()` function.
- `quadtree.cpp` and `quadtree.h`
  - class `QuadTree`, class to hold information about the quatree itself with a pointer to the root node
  - class `QuadTreeNode`, the actual quadtree node class.

A.1.2 *Installation*

The project is written in C++ and was developed and tested in Microsoft Visual Studio 2010, project files for which are included for compilation within Visual Studio. Apart from that, a Makefile is provided for compilation with gcc on Linux.

The program uses several libraries: Tinyexr[1], a header only library for EXR image manipulation located in the ./include directory; Eigen, a library for linear algebra, left for the user to install; and finally Cuda, also left for the user to install.

Compiled 64-bit binaries are included for Microsoft Windows and Linux. Tested on Windows 7 and Ubuntu 14.04 LTS.

A.1.3 *Usage*

Perun can be run without any parameters: as such it will generate one frame on a 64x128 simulation grid with low branching using the prog2 solver. Furthermore, the user can customise the output through user runtime parameters:

PARAMETER	OUTCOME
`-e ETA`	Branching factor. `ETA = 1` more branching, `ETA = 2` less branching.
`-f FRAMES`	Number of frames to animate.
`-g`	Do not generate new shape.
`-h HEIGHT`	Simulation grid height.
`--help`	Display this help listing.
`-l1 LUMINANCE`	Luminance of main channel. Default = 4.0.
`-l2 LUMINANCE`	Luminance of secondary channels. Secondary channels are those longer than `TYPE_2_CUTTOFF`. Default `LUMINANCE = 1.0`.
`-l3 LUMINANCE`	Luminance of ternary channels. Ternary channels are all others.
`-r RESOLUTION`	Resolution multiplicator of rendered image. Resulting image width will be `WIDTH * RESOLUTION`. Default `RESOLUTION = 8`.
`-s SOLVER`	Solver to use. Valid options `regular`, `eigen`, `prog2` or `cuda`.
`-t2 TYPE_2_CUTOFF`	Secondary channels cuttof. Default `TYPE_2_CUTOFF = 20`.
`-w WIDTH`	Simulation grid width.

1 https://github.com/syoyo/tinyexr

A.2 NUKE PLUG-IN

A.2.1 *Files and Classes*

- `cg_solver.cpp` and `cg_solver.h`
  - class `CGSolver`, the class which implements the prog2 conjugate gradient solver
- `common.h`
  - definitions of terms and constants used within the whole program
- `graph.cpp` and `graph.h`
  - class `DAG`, the node of the graph data structure to store the generated lightning shape
- `laplacianGrid.cpp` and `laplacianGrid.h`
  - class `LaplacianGrid`, the main class which holds all the neccessary methods and variables to simulate the Dielectric Breakdown Model / Laplacian Growth
- `PerunLightning.cpp` and `PerunLightning.h`
  - class `PerunLightning`, the actual Op/Node class for Nuke
- `quadtree.cpp` and `quadtree.h`
  - class `QuadTree`, class to hold information about the quatree itself with a pointer to the root node
  - class `QuadTreeNode`, the actual quadtree node class.

A.2.2 *Installation*

The project is written in C++ and was developed and tested in Microsoft Visual Studio 2010, project files for which are included for compilation within Visual Studio. In order to create Nuke plug-ins for Nuke 10.0v1 on Windows, plug-ins have to be compiled in Visual Studio 2010. Apart from that, a Makefile is provided for compilation with gcc on Linux, however gcc 4.1 is required to compile plug-ins for Nuke 10.0v1 on Linux. Other gcc versions are not guaranteed to work.

To compile the plug-in, the target version of Nuke has to be installed. This version uses Nuke 10.0v1.

The program uses the Eigen library for linear algebra which has to be installed and the path set in Visual Studio or in the Makefile.

A compiled 64-bit DLL is included for Microsoft Windows.

The gizmo is platform-independent and also included on the CD.

A.2.3 *Usage*

The plug-in (and the gizmo if desired) has to be copied to a directory set in Nuke to search for plug-ins, by default the .nuke directory in c:/Users/Username is the easiest choice on Windows. On Linux, this would be /.nuke/. To bring up the plug-in in Nuke, with focus on the node graph, pressing x will show a dialog to enter a TCL command—typing "PerunLightning" will create a new node. Typing "PerunGizmo" will bring up the gizmo.

B

ENCLOSED CD

The CD as part of the thesis is enclosed with the printed written part. It has the following directory structure:

- `/bin/` contains compiled versions of both the standalone application and the Nuke plug-in. Standalone application binaries compiled for both Windows and Linux can be found in `/bin/standalone/`; the `/bin/nuke/` folder contains the Nuke plug-in compiled for Nuke 10.0v1 on Windows—both the DLL library (`PerunLightning.dll`) and the gizmo (`PerunGizmo.gizmo`).
- `/doc/` contains reference guides for the code generated by Doxygen. A HTML version and a generated PDF are included for both the plug-in and standalone version code.
- `/nuke/` contains the Nuke scripts used for demonstration scenes as well as the original footage/elements.
- `/pdf/` contains the electronic version of this document.
- `/src/` contains the source code for both projects.
- `/video/` contains rendered videos that demonstrate the use of the plug-in.

C

LIST OF SYMBOLS AND ACRONYMS

C.1 LIST OF SYMBOLS

SYMBOL	MEANING
kA	kiloampere, a unit of current
°K	degrees Kelvin, a unit of temperature
p_i	probability of candidate i being chosen
$\mathbf{E}$	Electric field
ε_0	Permittivity of vacuum
η	a constant that influences brachng in DBM
μs	microseconds, a unit of time
∇	gradient of a vector field
$\nabla\cdot$	divergence of a vector field
∇^2	Laplace operator, $\nabla^2 = \Delta = \nabla \cdot \nabla$
ϕ_i	electric potential in candidate site i
φ	charge density

C.2 LIST OF ACRONYMS

ABBREVIATION	MEANING
API	application programming interface
APSF	Atmospheric Point Spread Function
BLAS	Basic Linear Algebra Subprograms, a specification defining a set of routines performing common linear algebra operations
C++	A programming language
CG, CGs	cloud-to-ground lightning
CGI	computer generated imagery
CPU	central processing unit, the processor of a computer
CSR	compressed sparse row format, a format for storing sparse matrices
CUDA	a parallel computing platform by NVIDIA
DAG	Directed Acyclic Graph
DBM	Dielectric Breakdown Model, a physically based method of electric discharge simulation
DLL	Dynamic-link library
EXR	a high dynamic range image format
GPU	graphics processing unit, the graphics card of a computer
HDR	high dynamic range
IC, ICs	icntracloud, intercloud or cloud-to-air lightning
NDK	Nuke Development Kit
SDK	software development kit
VFX	visual effects

REFERENCES

[1] The open effects association homepage. URL http://openeffects.org/.

[2] Tomáš Bergl. Modelování a vizualizace blesků. Master's thesis, Czech Technical University, 2008.

[3] Bernd Bickel, Martin Wicke, and Markus Gross. Adaptive simulation of electrical discharges. In *Vision, Modeling, and Visualization*, pages 209–216, Aachen, Germany, 2006. IOS Press.

[4] Loren C. Carpenter. Computer rendering of fractal curves and surfaces. *SIGGRAPH Comput. Graph.*, 14(3):109–, July 1980. ISSN 0097-8930. doi: 10.1145/965105.807478. URL http://doi.acm.org/10.1145/965105.807478.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry, Algorithms and Applications*. Springer-Verlag, Berlin, Germany, second, revised edition, 2000. ISBN 3-540-65620-0.

[6] Luc Devroye. A note on the height of binary search trees. *J. ACM*, 33(3):489–498, May 1986. ISSN 0004-5411. doi: 10.1145/5925.5930. URL http://doi.acm.org/10.1145/5925.5930.

[7] Bc. Richard Dobřichovský. Perun, a lightning plug-in for the foundry's nuke compositor, a4m39svp—software or research project, 2015.

[8] Kenneth Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1990. ISBN 0-471-92287-0.

[9] *Blink API Reference*. The Foundry Visionmongers, Ltd., 2014. URL https://www.thefoundry.co.uk/products/nuke/developers/90/blinkreference/.

[10] *Nuke binary plugins Documentation*. The Foundry Visionmongers, Ltd., 2014. URL http://docs.thefoundry.co.uk/nuke/90/ndkreference/Plugins/.

[11] Andrew Glassner. The digital ceraunoscope: Synthetic thunder and lightning, part 1. *IEEE Comput. Graph. Appl.*, 20(2):89–93, March 2000. ISSN 0272-1716. doi: 10.1109/38.824552. URL http://dx.doi.org/10.1109/38.824552.

[12] David J. Griffiths. *Introduction to Electrodynamics*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, third edition, 1999. ISBN 0-13-805326-X.

[13] Karel Havlíček-Borovský. *Křest sv. Vladimíra*. Jiří Chvojka, Havlíčkův Brod, 1948.

[14] R. D. Hill. Analysis of irregular paths of lightning channels. *Journal of Geophysical Research*, 73(6):1897–1906, 1968. ISSN 2156-2202. doi: 10.1029/JB073i006p01897. URL http://dx.doi.org/10.1029/JB073i006p01897.

[15] Theodore Kim and Ming C. Lin. Physically based animation and rendering of lightning. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, PG '04, pages 267–275, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2234-3. URL http://dl.acm.org/citation.cfm?id=1025128.1026050.

[16] Theodore Kim and Ming C. Lin. Fast animation of lightning using an adaptive mesh. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):390–402, March 2007. ISSN 1077-2626. doi: 10.1109/TVCG.2007.38. URL http://dx.doi.org/10.1109/TVCG.2007.38.

[17] Theodore Kim, Jason Sewall, Avneesh Sud, and Ming C. Lin. Fast simulation of laplacian growth. *IEEE Comput. Graph. Appl.*, 27(2):68–76, March 2007. ISSN 0272-1716. doi: 10.1109/MCG.2007.33. URL http://dx.doi.org/10.1109/MCG.2007.33.

[18] Paul Kruszewski. A probabilistic technique for the synthetic imagery of lightning. *Computers & Graphics*, 23(2):287 – 293, 1999. ISSN 0097-8493. doi: http://dx.doi.org/10.1016/S0097-8493(99)00038-2. URL http://www.sciencedirect.com/science/article/pii/S0097849399000382.

[19] prof. RNDr. Petr Kulhánek, CSc. *Blýskání, aneb, třináctero vyprávění o plazmatu*. Aldebaran Group for Astrophysics, Praha, 2011. ISBN 978-80-904482-3-9.

[20] Irving Langmuir. Oscillations in ionized gases. *Proceedings of the National Academy of Sciences*, 14(8):627–637, 1928.

[21] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, New York, 1982. ISBN 0-7167-1186-9.

[22] Srinivasa G. Narasimhan and Shree K. Nayar. Shedding light on the weather. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'03, pages 665–672, Washington, DC, USA, 2003. IEEE Computer

Society. ISBN 0-7695-1900-8, 978-0-7695-1900-5. URL http://dl.acm.org/citation.cfm?id=1965841.1965928.

[23] Bruno Nicoletti. *The OFX Image Effect Plug-in API, 1.3, Programming Reference*, 2012. URL http://openfx.sourceforge.net/Documentation/1.3/Reference/index.html.

[24] L. Niemeyer, L. Pietronero, and H. J. Wiesmann. Fractal dimension of dielectric breakdown. *Physical Review Letters*, 52:1033–1036, 1984.

[25] *cuBLAS Library User Guide*. NVIDIA Corporation, 2015. URL http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.

[26] *cuSPARSE Library*. NVIDIA Corporation, 2015. URL http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf.

[27] Peter Pacheco. Programming assignment 2: The conjugate gradient method, parallel and distributed computing class, university of san francisco, 2014. URL http://www.cs.usfca.edu/~peter/cs625/prog2.pdf.

[28] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996. ISBN 0-387-94676-4.

[29] Vladimir A. Rakov and Martin A. Uman. *Lightning: Physics and Effects*. Cambridge University Press, Cambridge, first edition, 2003. ISBN 0-521-58327-6.

[30] Todd Reed and Brian Wyvill. Visual simulation of lightning. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 359–364, New York, NY, USA, 1994. ACM. ISBN 0-89791-667-0. doi: 10.1145/192161.192256. URL http://doi.acm.org/10.1145/192161.192256.

[31] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[32] Batjargal Sosorbaram, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba. Visual simulation of lightning taking into account cloud growth. In *Computer Graphics International 2001*, CGI '01, pages 89–98, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1007-8. URL http://dl.acm.org/citation.cfm?id=647781.735220.

[33] Michal Téra. *Perun – bůh hromovládce: sonda do slovanského archaického náboženství*. Mervart, Červený Kostelec, 2009. ISBN 978-80-86818-82-5.

[34] Lars Wåhlin. *Atmospheric Electrostatics*. John Wiley and Sons Inc., New York, NY, 1986. ISBN 0-471-91202-6.

[35] Pao K. Wang. *Physics and Dynamics of Clouds and Precipation*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK, 2013. ISBN 978-1-107-00556-3.

[36] Wikipedia. Tortuosity — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=Tortuosity&oldid=716717413. [Online; accessed 8-May-2016].

[37] Wiktionary. ceraunoscope — wiktionary, the free dictionary, 2016. URL https://en.wiktionary.org/w/index.php?title=ceraunoscope&oldid=36780126. [Online; accessed 8-May-2016].