CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Example-based Stylization of 3D Renderings on the GPU
Student:	Bc. Martin Dzurenko
Supervisor:	doc. Ing. Daniel Sýkora, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2017/18

Instructions

Explore state-of-the-art in example-based stylization of 3D renderings [1,2]. Given existing CPU implementation of these algorithms identify time consuming parts that are suitable for parallelization on currently available GPUs. Focus preferably on the problem of randomized nearest neighbor retrieval with constraints on equitable utilization [3,4]. Using NVIDIA CUDA implement a parallel version of the algorithm [2] based on [4] and evaluate its efficiency on a real data set provided by the thesis supervisor. Verify that the GPU implementation achieves comparable visual quality as the original CPU version of [2].

References

[1] Bénard et al.: Stylizing Animation by Example. ACM Transactions on Graphics 32(4):119, 2013.
[2] Fišer et al.: StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings, ACM Transactions on Graphics 35(4):92, 2016.
[3] Barnes et al.: PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing, ACM Transactions on Graphics 28:3(24), 2009.
[4] Kaspar et al.: Self tuning texture optimization. Computer Graphics Forum 34(2):349-360, 2015.

[illegible]

Ing. Michal Valenta, Ph.D.
Head of Department

České vysoké učení technické v Praze
Fakulta informačních technologií
L.S.
-3-

Pavel Tvrdík

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 8, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Software Engineering

Master's thesis

Example-based Stylization of 3D Renderings on the GPU

Bc. Martin Dzurenko

Supervisor: doc. Ing. Daniel Sýkora, Ph.D.

10th January 2017

Acknowledgements

First of all, thanks to Daniel Sýkora for his phenomenal supervising skills and precise approach during the whole process. Thanks a lot to Ondřej Jamriška for his brilliant ideas, unmatched enthusiasm and keen willingness to help at all times. Finally, thanks a lot to my family for their unconditioned support and faith in me.

Declaration

In Prague on 10th January 2017 ....

Citation of this thesis

Dzurenko, Martin. *Example-based Stylization of 3D Renderings on the GPU.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Cílem stylizace vzhledu na základě ručně kreslené předlohy je co nejvěrnější reprodukce stylu umělce. Aplikováním stylizace lze ušetřit čas a námahu umělci díky eliminaci repetitivní práce, např. při tvorbě animací. Současné přístupy k stylizaci nedokáží zachovat všechny detaily uměleckého díla. Častým problémem jsou nedostatečné informace popisující vstupní data a algoritmy vytvářející viditelné artefakty. Cílem této práce je vytvořit alternativu moderního referenčního přístupu, který se zabývá řešením těchto problémů. Naše metoda je vhodná pro paralelní implementaci na grafickém akcelerátoru. Informace popisující vstupní data zobrazují různé části osvětlení modelové scény, čímž je umožněno přesnější zachování stylu. Algoritmus syntézy využívá minimalizování energie textury s termem energie rozšířeným o váhy pixelů pocházející z mapy výskytů za účelem dosáhnout rovnoměrného využití okolí pixelů z předlohy. Referenční implementace naší metody na GPU je detailně popsána. Kvalita výsledků referenční implementace je porovnatelná s moderní referenční metodou zatímco doba výpočtu je řádově kratší. V závěrečné části jsou předvedena další vylepšení referenční GPU implementace. Důraz je kladen na zvýšení výkonu bez viditelnějších dopadů na výsledky.

Klíčová slova stylizace, na základě předlohy, mapa výskytů, GPU paralelizace

Abstract

The aim of stylization by example is to faithfully reproduce artist's style. Application of the stylization saves a lot of time and effort by eliminating repeated work, e.g., when creating animations. Most of the current approaches are unable to preserve all the details provided by the artist due to insufficient guidance or algorithms producing visible artifacts. The goal of this thesis is to create an alternative to a reference state-of-the-art approach that alleviates these problems. Our approach is suitable for GPU parallelization. It is guided by illumination decomposed to multiple 3D renderings of the same scene that enable detailed preservation of the style. The synthesis algorithm utilizes texture energy minimization with an extended energy term that includes pixel weights from the occurrence map to encourage uniform distribution of patch assignments. A reference GPU implementation of our method is described in detail. Results of the implementation are of a quality comparable to those of the reference state-of-the-art method while the performance is in orders-of-magnitude faster. Finally, additional improvements to the reference GPU implementation are proposed that further increase the performance of the reference GPU method without visible impact on quality of the results.

Keywords stylization, example-based, occurrence map, GPU parallelization

Contents

List of Figures

Introduction

Creation of a complex hand-drawn art from scratch is a time-consuming and expensive process. In most cases, the overall result is a consistent application of the artist's personal style on a medium. The application often includes tedious and repeated work of craft that takes non-negligible time. While the creative part is currently irreplaceable with a machine, the repetitive part can be taken care of once the machine has "learned" the style. The stylization of 3D renderings by example is, therefore, desirable and applicable in a wide array of areas such as animation or video games. For this kind of computer aid to work properly and to create plausible results, one must consider many aspects of the artist's style. There can be various colors for different types of shadows and reflections (see Fig. I.1) or different strokes around edges and plain areas. Micro-scale details (e.g., painting methods such as watercolor, oil, chalk) are just as important as macro-scale ones (color transitions, an orientation of strokes, display of shapes). All this has to be taken into account and addressed properly [6].

There has been a lot of research in the stylization by example area in the past decades [17, 9, 5, 7, 6] that made many aspects of its current state possible. The problem being solved is usually the same, given an example guidance texture A and its stylized version A', apply the same stylization on a given target guidance texture B to produce the result B' (see Fig. I.2).

Results of the previous approaches to stylization by example differ in quality. Their guidance is often driven by color information [9, 5] in the guidance textures A and B which makes them unable to distinguish and reproduce different illumination effects properly. Some of the approaches apply greedy scan-line method [9, 5] in the algorithm which causes unpleasant artifacts and unnatural looking seams. Algorithms that use texture optimization [12, 19, 7, 4] give better results overall, but are prone to excessive patch usage resulting in a distinct wash-out effect [13] present in the stylized target. Recent research [10, 11, 6] alleviate the wash-out effect problem by encouraging uniform distribution of example patches.

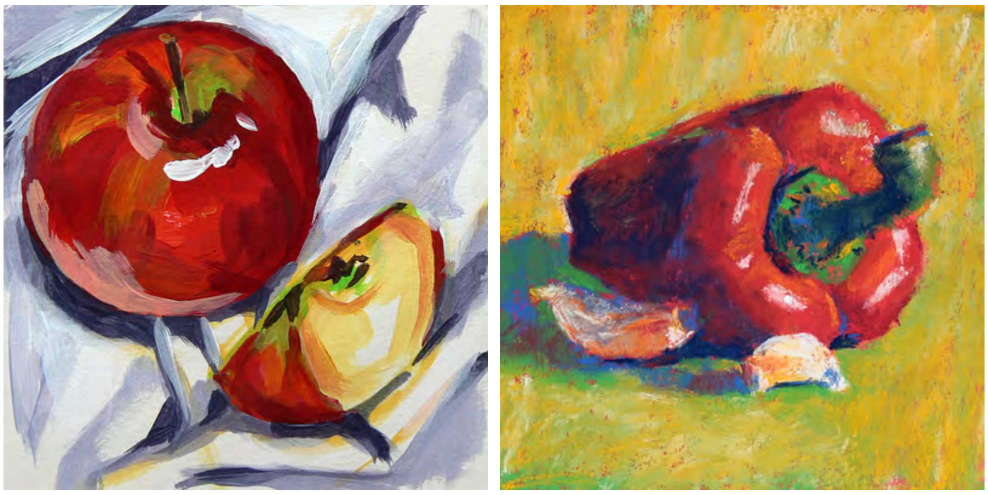

Figure I.1: The style of an artist in a real painting. It conveys many specific details, e.g., both pink and dark shadows of the apple (left), or a hint of blue in the pepper's shadows (right). Image courtesy © *Kerry Daley* (left) and *Gail Sibley* via howtopastel.com (right). Source of images Fišer et al. [6]

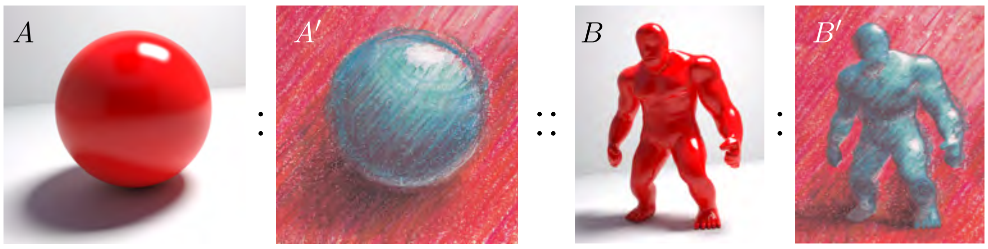

Figure I.2: Demonstration of the problem of stylization by example. A is the example guidance texture and B is the target guidance texture. The artist draws stylization of the example A' following the exact outlines and illumination areas of the guidance. Given A, A' and B the program synthesizes stylization of the target B'. Source of images Fišer et al. [6]

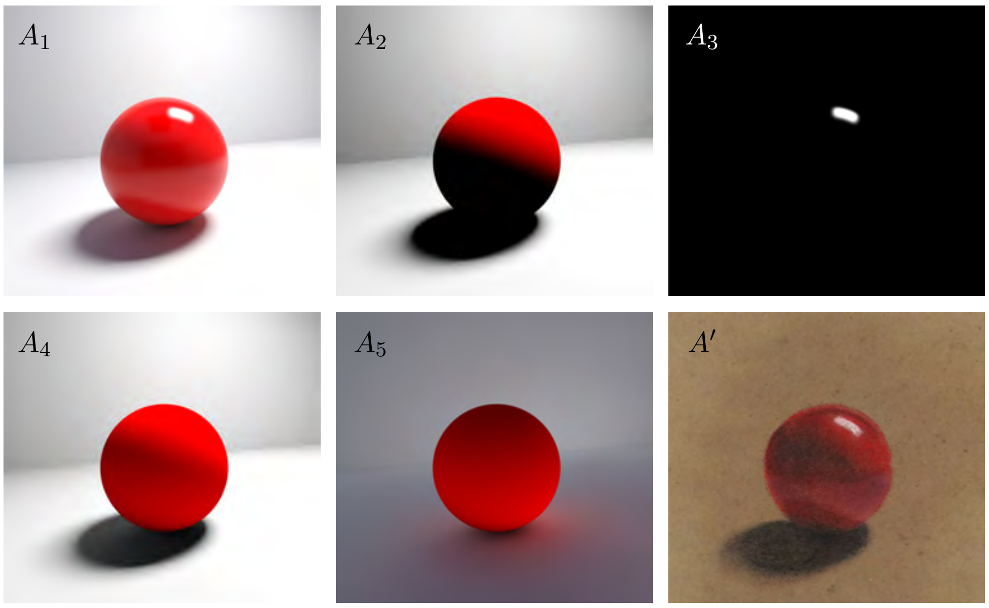

Figure I.3: Five guidance textures of decomposed illumination using LPEs along with the stylized example texture A' © *Daichi Ito*. A_1 full global illumination render, A_2 direct diffuse, A_3 direct specular, A_4 first two diffuse bounces, A_5 diffuse inter reflection. Images source Fišer et al. [6]

These publications are basis for the very recent work called *StyLit* [6] that produces state-of-the-art example-based stylization by introducing two important improvements. First is illumination-enabled guidance. Illumination of the guidance textures A and B is now decomposed to five textures using light path expressions (LPEs) in order to display the effects separately (see Fig. I.3). They enable synthesis algorithm to match areas of the same illumination effect with better accuracy which in turn applies the stylization on the target as the artist would have intended. Second is an adaptive use of enforced uniform patch assignments [10] in the algorithm. It plays an important role in equal distribution of the whole style A' on the result B' that preserves high-frequency details and does not leave disturbing artifacts or repetitive patterns.

The motivation of our efforts is to improve the performance of the current stylization algorithms. The closer the evaluation of stylized texture gets to real-time the more productive and efficient the artist can be. In this regard, we aim our approach to be fully and effectively parallelizable on GPU since its hardware architecture is much better suited for this kind of problems and can decrease time consumed multiple times [1]. Our solution builds upon the existing *reference GPU implementation* (from now on referred to as “Ref”) which is based on the idea of additional guidance textures of StyLit [6] and energy minimization [12, 19] with its energy term extended by an occurrence term [11].

Goal

In our work, we will first look into the research background by analyzing numerous state-of-the-art solutions with a focus on Fišer et al. (StyLit) [6]. Next, a new problem of energy minimization will be described along with our approach to its solution and followed by reviewing the most important algorithmic parts of Ref. Results of Ref will be compared to those of StyLit in quality as well as time-consumption and the benefits of the Ref's GPU solution will be presented. As our main contribution, we will then propose *further improvements* of Ref. Method, implementation and comparison of our results to those of Ref and StyLit will be provided for each improvement. Our aim will be to improve Ref's performance without notably decreasing the quality of its results.

Chapter 1

Background

In this chapter we will talk about different and widely used approaches to texture synthesis and the problem of stylization by example.

1.1 Image Analogies

Hertzmann et al. [9] presents the problem of stylization by example by providing the two guidance textures A and B and the stylized version A' of the texture A (see Fig. I.2, note that Hertzmann et al. allows arbitrary textures A and B, e.g., two different photographs). The algorithm generates the result B' which is a stylization of B that is similar to the stylization A' of A by matching similar areas of A and B and then transferring their corresponding pixel values from A' to B'.

Guidance is mostly based on the color information of the textures A and B. As a result, the algorithm is not able to match areas of the same illumination effect and thus fails in proper application of the style (see how background area of the style is incorrectly applied to the target area that should contain direct light reflection in the image (d) in Fig. 1.1). Another issue of this approach is its greedy nature when synthesizing as it creates seams and other visible artifacts which lead to wrong and synthetic looking results.

The synthesis algorithm is of greedy nature. Each pixel of B' is processed in a scan-line order, its surrounding area is matched to the most suitable example area (with combination of global [2] and coherent [18] search) and its corresponding stylization pixel value is used as a resulting value of the processed pixel.

1.2 Texture Optimization

Alternative approach to texture synthesis algorithm as opposed to the greedy one [9] is *texture optimization* [12, 19]. Instead of filling the result pixel-by-pixel, the method refines the entire texture from coarse to fine quality as it iteratively minimizes the *texture energy* term in an expectation-maximization fashion. The value of a single pixel of B' is now evaluated by averaging multiple pixels' values of A'. Overall, the method gives better results in simple synthesis scenarios [7, 4], but when additional guidance channels are included the iteration often creates wash-out effect in the result (see the "smoothed" look of the image (f) in Fig. 1.1). The effect is caused by excessive usage of example patches in assignments with low-frequency content [13].

To alleviate this issue, a specific research was conducted to find ways to encourage uniform patch utilization [10, 11]. Jamriška et al. [10] uses reversed nearest-neighbor retrieval that enforces the uniformity, but at the same time fails to realistically synthesize StyLit [6] scenarios due to the enforcement. The size of areas of a matching illumination effect often differ in the example and target guidance and once all correct patches have been used, wrong ones are forced to the remaining parts of the area (see how the direct reflection area of the toroids is forced to incorrectly match background style in the image (g) in Fig. 1.1)

Kaspar et al. [11] introduces an improved patch similarity term that counts each example pixel's usage in assignments to dynamically adjust its patch usage penalization during the synthesis process. This way uniformity of assignments is more flexible and can adjust to a wider range of scenarios. The drawback is the term includes unknown value of λ which must be specifically tailored for each scenario as it controls the strength of pixel usage penalization.

1.3 PatchMatch

Barnes et al. [3] proposes a randomized algorithm for *approximate nearest-neighbors* retrieval. The retrieval is a core component in texture synthesis and many other image processing applications (e.g., noise reduction) and is often their performance bottleneck. Brute force solution that evaluates non-approximated exact nearest-neighbors would be extremely slow in most applications.

The approximation is iterative, its convergence is very fast and the resulting difference of patch distances from the exact solution becomes negligible with enough iteration steps. The algorithm consists of two interleaved procedures. The *propagation* spreads coherent areas of assignments in the synthesized texture as they often lead to best-perceived synthesis results. The *random search* searches for improving assignments randomly through the example texture.

Compared to previous approaches to the retrieval, PatchMatch achieves 20-100$\times$ decrease in time consumption.

1.4 Stylizing Animations by Example

Bénard et al. [5] further improves Image Analogies [9] by making it possible to render stylized animations by providing every 10-20th stylized frame. The most important and relevant contribution on top of the Image Analogies work is improved synthesis guidance.

The improved guidance ensures temporal coherence of frames, preservation of edges in the stylized target, ability to match rotated patches, etc. Additional guidance textures are used as well, but they are derived from the initial guidance textures A and B which causes the whole approach to depend on their color information and is, therefore, unable to create plausible results for the same reason the approach of Hertzmann et al. [9] cannot. (see the image (e) in Fig. 1.1).

Searching for patch assignments is evaluated with PatchMatch [3], but the algorithm also applies the greedy nature of Hertzmann et al. [9] which results in visible artifacts and disturbances in the result and is, therefore, insufficient for our needs.

1.5 StyLit

Fišer et al. [6] introduces illumination-guided stylization of 3D renderings that better preserves the richness and detail of a hand-made art. The artist draws the stylization A' of the example rendering A following its outlines and StyLit synthesizes similar stylization B' of the target rendering B (see Fig. I.2).

To enable illumination guidance, the most important illumination effects are decomposed to separate guidance textures using LPEs (see Fig. I.3). This way, the matching of example and target patches includes and considers all the provided effects. For instance, the white area in the guidance texture A_3 causes most of its patches to be matched to patches of the same white area in the guidance texture B_3. Otherwise, the distance metric would yield the patches as very distant. This way the additional guidance textures ensure closeness of all illumination effects at the same time in the assignments and as a result preserves their stylizations correctly (see the image (h) in Fig. 1.1).

Greedy synthesis algorithm [9] is replaced by optimization technique [12, 19]. Excessive patch assignments are dealt with the enforced uniform patch distribution [10]. As already mentioned, the enforced uniformity alone causes visible artifacts in the StyLit's scenario as many of the assignments are forced to incorrect positions. It is, therefore, used in an adaptive iterative fashion wherein each step only the correctly enforced assignments are kept. The iteration runs until no more unassigned patches remain. The drawback is that the iteration might require a lot of steps and the estimation of correct assignments in each step includes sorting which makes GPU implementation complicated and ineffective.

1.6 A Neural Algorithm of Artistic Style

An alternative approach of Gatys et al. [8] applies style of a painting A' on a given photograph B to create a stylized version of the photograph B'. The approach does not consider the example guidance texture A at all.

The method is based on parametric texture synthesis [14] where the Gabor filters are replaced with VGG-Network [16]. First, response from a specific layer in the network is evaluated for A' such that it represents its "style" (smaller-scale patterns). Another response from a different layer in the same network is evaluated for B that represents its "content" (larger-scale features). The result B' is initialized as white noise. By evaluating both the responses on texture B' and comparing their closeness to the responses from A' and B, a minimization problem is obtained. Iterative gradient descent is applied to obtain the resulting stylization B' that contains both "style" of A' and "content" of B. VGG-Network [16] is a publicly available convolutional neural network trained for human object recognition.

While the results on natural photographs and paintings are very impressive, the approach fails in our case as it does not work well in such synthetic scenarios (see the image (c) in Fig. 1.1).

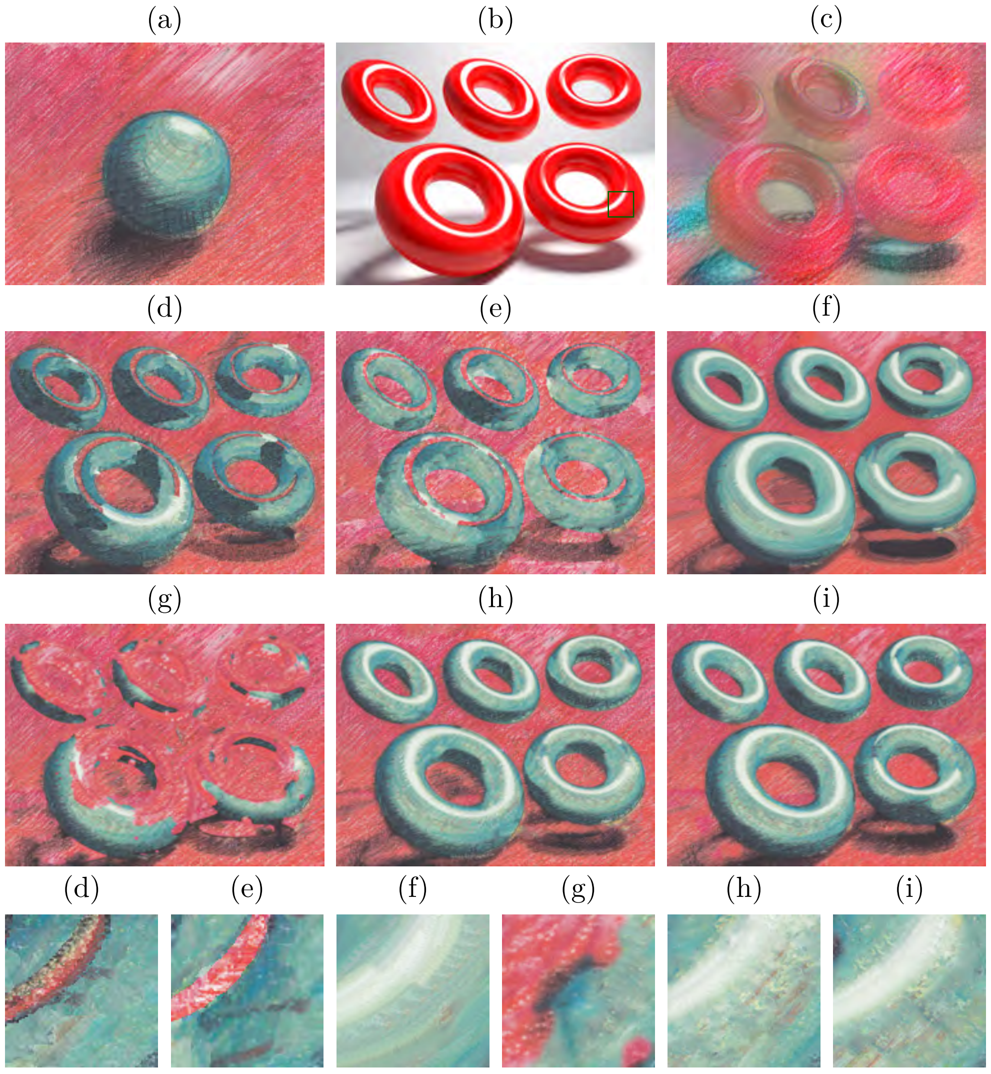

Figure 1.1: Quality comparison of results of background approaches. (d) and (e) used only A_1 and B_1 as guidance. (a) Example stylization © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) Gatys et al. [8]. (d) Hertzmann et al. [9]. (e) Bénard et al. [5]. (f) Wexler et al. [19]. (g) Jamriška et al. [10]. (h) Fišer et al. [6]. (i) Ref with $\lambda = 10000$. Source of images (a)…(h) Fišer et al. [6].

Chapter 2

Method

In this chapter, we will formulate a new synthesis problem and describe its solution which is based on the relevant background research [12, 19, 11] and StyLit [6]. We will start by providing formal definitions of often used terms.

2.1 Definitions

Texture

Texture I is of specified width w_I, height h_I, has n_I channels and a function f^I:

$$f^I : \langle 0, w_I - 1\rangle \times \langle 0, h_I - 1\rangle \rightarrow H_1 \times H_2 \times \cdots \times H_{n_I}$$

where the pixel values are vectors of individual channel values. Typically, a single texture consists of RGB color channels.

In this text, for simplicity reasons, the guidance and stylization textures (RGB) are all included in single textures $\bar{A}$ and $\bar{B}$ which means their pixels consist of total $(1 + 5) \times 3 = 18$ channels. We will exclusively refer to the stylization "subtexture" B' of $\bar{B}$ with notation $\bar{B}$.style. Note that $\bar{B}$.style is also a texture in which the result will be evaluated. Likewise, we will exclusively refer to the guidance "subtexture" $B_1, B_2, \ldots, B_5$ of $\bar{B}$ with notation $\bar{B}$.guidance. The same notation applies to $\bar{A}$.

Patch

Patch P is a small square of pixels of odd width w_P in texture I and its coordinates $(u_P; v_P)$ are given by its center pixel (see Fig. 2.1). The patch has a function f^P:

$$f^P : \langle 0, w_P - 1\rangle \times \langle 0, w_P - 1\rangle \rightarrow H_1 \times H_2 \times \cdots \times H_{n_I}$$

$$f^P(x, y) = f^I(u_P - r_P + x, v_P - r_P + y)$$

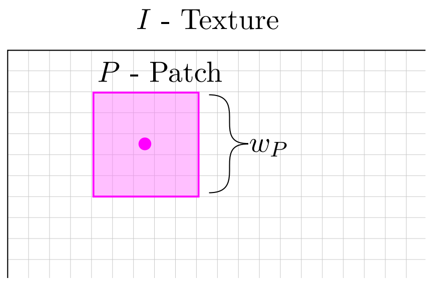

Figure 2.1: Patch P of texture I. Width of the patch is w_P and the small pink circle represents its coordinates $(u_P; v_P)$.

where radius $r_P = \frac{w_P - 1}{2}$. All patches have the same width in a single instance of the problem, which means it is considered as a global parameter in the algorithm. Set of all patches of a texture I is labeled as $\mathcal{N}_I$.

Patch Translation

Patch translation ($\odot$) is an operation applied on some patch P and vector $(x; y)$. It returns new translated patch Q:

$$\odot : \mathcal{N}_I \times (\mathbb{Z} \times \mathbb{Z}) \to \mathcal{N}_I$$

where $u_Q = u_P + x$, $v_Q = v_P + y$ and $w_Q = w_P$. We will use the translation operation in infix notation.

Patch Similarity

Patch similarity is a distance metric δ applied between patch P from $\bar{B}$ and patch Q from $\bar{A}$. We use L_2-norm which is a sum of squared differences (SSDs) of their corresponding channel values. It is a widely used metric to measure the exact distance between two images:

$$\delta : \mathcal{N}_{\bar{B}} \times \mathcal{N}_{\bar{A}} \to \mathbb{N} \cup \{0\}$$

$$\delta(P, Q) = \sum_{x=0}^{w_P-1} \sum_{y=0}^{w_P-1} \sum_{i=1}^{n_{\bar{B}}} (f_i^P(x,y) - f_i^Q(x,y))^2$$

Nearest-neighbor Field

Nearest-neighbor field (NNF) is a relation between textures $\bar{B}$ and $\bar{A}$ that describes closest patch assignments based on some patch metric φ (e.g., patch similarity δ):

$$\text{NNF} : \mathcal{N}_{\bar{B}} \to \mathcal{N}_{\bar{A}}$$

$$\mathrm{NNF}(P) = \operatorname*{arg\,min}_{Q \in \mathcal{N}_{\bar{A}}} \varphi(P, Q)$$

Note that such rigid specification of NNF is expensive to obtain. Therefore, the condition is relaxed by searching for *approximate NNF* (the assigned $\bar{A}$ patches are only approximately closest). From now on, the abbreviation NNF will refer to the approximate nearest-neighbor field.

Occurrence Map

Occurrence map Ω [11] stores weights of pixels of the example texture $\bar{A}$. The weight corresponds to the number of assigned $\bar{A}$ patches that contain the pixel's coordinates:

$$\Omega : \langle 0, w_{\bar{A}} - 1\rangle \times \langle 0, h_{\bar{A}} - 1\rangle \to \mathbb{N} \cup \{0\}$$

$$\Omega(x, y) = |\{P \in \bar{B} : (x; y) \in \varkappa(\mathrm{NNF}(P))\}|$$

where $\varkappa(Q)$ denotes a set of all coordinates of pixels covered by the patch Q.

Patch Penalization

Patch penalization term ρ [11] penalizes repeated usage of example patches in order to encourage their uniform distribution in NNF. The term is based on weights of pixels of an example patch Q in the occurrence map Ω:

$$\rho : \mathcal{N}_{\bar{A}} \to \mathbb{R}$$

$$\rho(Q) = \frac{\omega(Q)}{\omega_{\mathrm{best}}}$$

where ω_{best} is the number of times each pixel of texture $\bar{A}$ would be used in a perfectly uniform scenario:

$$\omega_{\mathrm{best}} = w_Q^2 \frac{w_{\bar{B}} h_{\bar{B}}}{w_{\bar{A}} h_{\bar{A}}}$$

and $\omega(Q)$ is the average weight of pixels of the patch Q:

$$\omega(Q) = \frac{1}{w_Q^2} \sum_{x=0}^{w_Q - 1} \sum_{y=0}^{w_Q - 1} \Omega(u_Q - r_Q + x, v_Q - r_Q + y)$$

2.2 Problem Formulation

Our solution is based on the *texture optimization* approach [12, 19] that minimizes *texture energy* of $\bar{B}$. The energy evaluates "closeness" of texture $\bar{B}$ to the texture $\bar{A}$, which implies “closeness” of the synthesized stylization B' to the example stylization A'. The energy term [12, 19] is as follows:

$$\sum_{P \in \mathcal{N}_{\bar{B}}} \varphi(P, \mathrm{NNF}(P))$$

where φ is a patch metric used in the evaluation of NNF.

In this regard, as proposed by Kaspar et al. [11], the metric φ is as follows:

$$\varphi(P, Q) = \lambda\ \rho(Q) + \delta(P, Q)$$

where the similarity term δ encourages the assignments to be as similar as possible while at the same time, the penalization term ρ encourages uniform distribution of example $\bar{A}$ patches in the assignments. Parameter λ controls the strength of the patch penalization. The energy of $\bar{B}$ is minimized in the same iterative fashion as in the texture optimization technique [12, 19].

Note that if NNF was exact and not approximated, the new metric φ would imply a fundamental issue stemming from the cyclic dependency of the definitions. Evaluation of exact NNF requires the state of Ω to correctly evaluate the closest patch assignments with metric φ. However, the newly evaluated assignments might cause different state of Ω, thus different weights of pixels in the metric, thus the possibility that some of the assignments are not longer closest. Our solution to evaluate approximate NNF searches for the equilibrium in an iterative fashion based on PatchMatch [3].

2.3 Texture Optimization

Texture optimization technique [12, 19] synthesizes the entire texture by refining it from coarse to fine quality. The refinement is achieved by minimizing the *texture energy* term in a way that is similar to expectation-maximization (EM) algorithm from statistics. Given two sets of unknown variables, i.e., assignments in NNF and pixel values in B', the iteration alternates between one set labeled as known and the other as unknown. Values of the known variables are used to evaluate new, more refined, values of the unknown variables. This way, the two sets are refined until a specific criterion is met, or until a required number of iterations are completed.

First, the M-step of the algorithm evaluates NNF using the current state of $\bar{A}$, $\bar{B}$ with PatchMatch [3]. In a case of the very first M-step, NNF assignments are initialized to random example patches. The occurrence map Ω changes as well during the NNF evaluation as the assignments change during the process. Second, the E-step of the algorithm evaluates B' using the current state of NNF in a procedure called *voting*.

In voting, new value of pixel $(x; y)$ is evaluated as an average of w_P^2 “votes” given by the patches of B' that cover the pixel $(x; y)$. Let there be patch

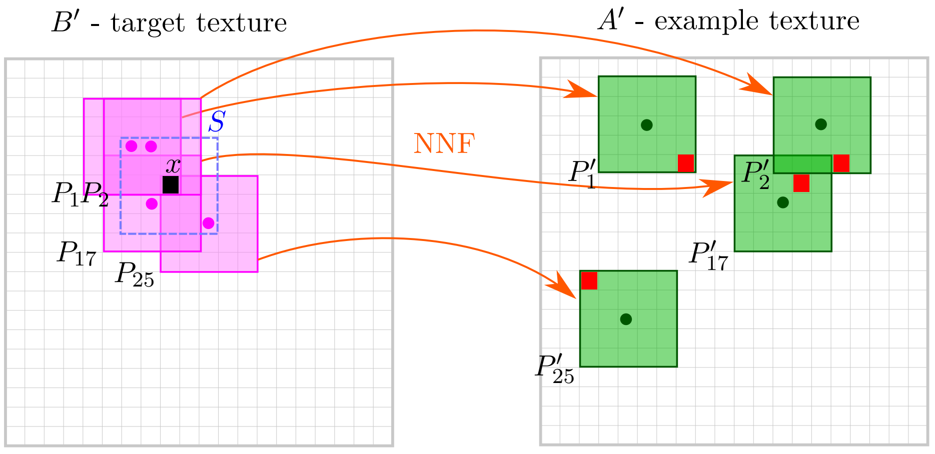

Figure 2.2: Voting for a new value of the pixel x in the synthesized texture B', patch width is 5. $P_1, P_2, \dots, P_{25}$ are the set of patches containing pixel x (dashed blue square S). These patches vote with values of pixels relative to x given by their assignments $P'_1, P'_2, \dots, P'_{25}$ in NNF (displayed as red pixels in A').

$P \in \mathcal{N}_{B'}$ so that $(x;y) \in \varkappa(P)$, then P votes with the value $f^{A'}(u_Q, v_Q)$ where $Q = \text{NNF}(P) \odot (x - u_P, y - v_P)$ (see Fig. 2.2).

Additionally, to further improve the refinement and to decrease impact of the randomly generated initial NNF, there are multiple EM iterations that are run in a pyramid-like resolution fashion. Textures $\bar{A}$ and $B_1, \dots, B_5$ are downsampled from their originals for each level. Starting from the lowest resolution, at the end of each level EM-refinement, both the NNF and B' are upsampled to the next level resolution where they are further refined (see Fig. 2.3). Since occurrence map Ω cannot be upsampled, its state is initialized using state of the upsampled NNF (procedure InitOccurrenceMap).

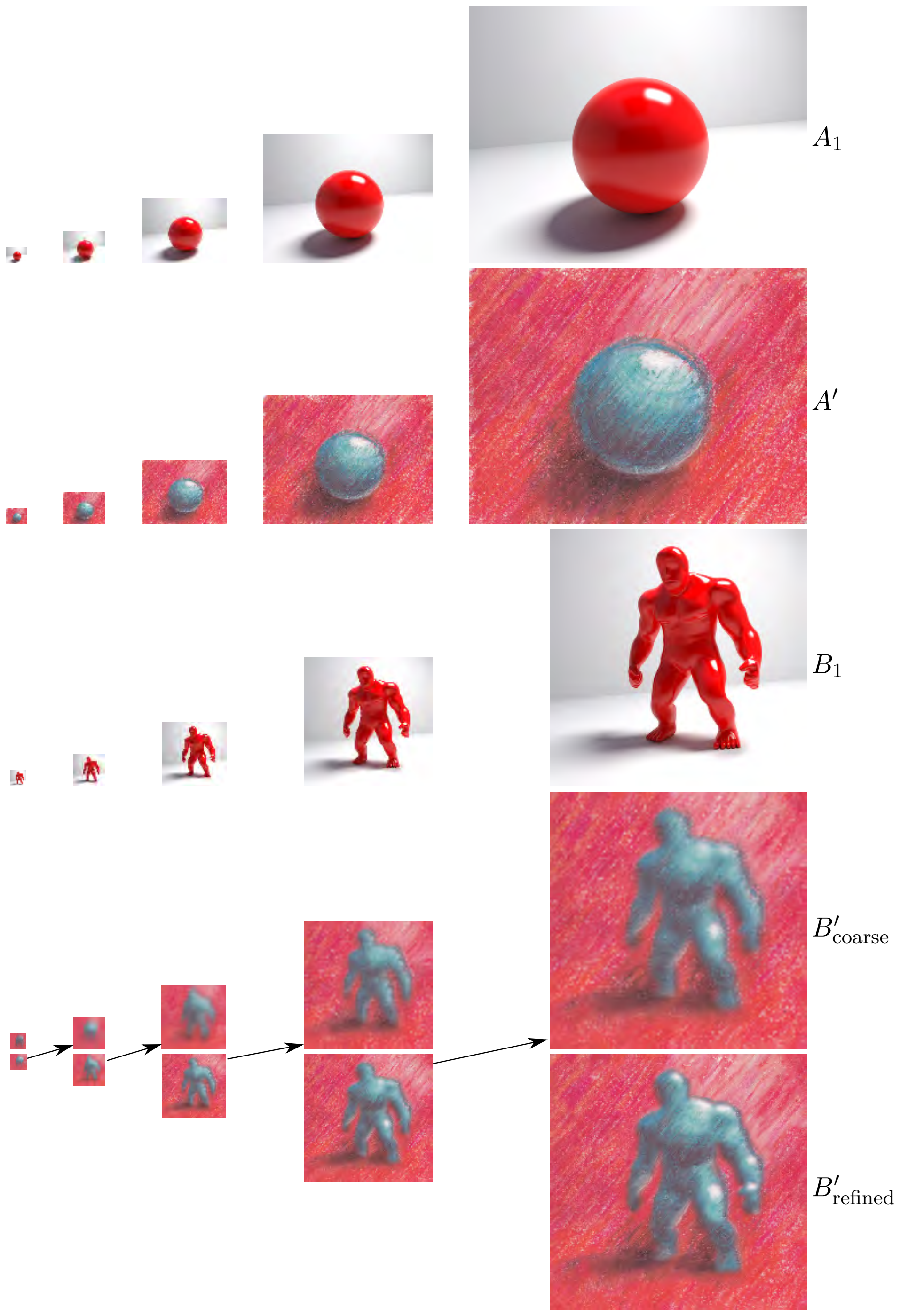

Figure 2.3: Five-leveled pyramid refinement of B'. $\bar{A}$ and $B_1, \ldots, B_5$ are downscaled from originals (only A_1 and B_1 are displayed). B'_{refined} are states after the final voting on the corresponding level and the black arrows display upsampling to the next level.

The whole approach of the texture optimization CPU algorithm is as follows:

```
function TextureOptimization(lvls, lvlIters, pmIters, $\bar{A}$, $\bar{B}$)
    $B'_{\uparrow} \leftarrow$ null
    NNF $\leftarrow$ RandomNNF$((w_{\bar{B}}, h_{\bar{B}}), (w_{\bar{A}}, h_{\bar{A}}))$
    for lvl $\leftarrow 1 \dots$ lvls do
        $\bar{A}_{\downarrow} \leftarrow$ Downsample$(\bar{A}, 2^{\text{lvls}-\text{lvl}})$
        $\bar{B}_{\downarrow} \leftarrow$ Downsample$(\bar{B}, 2^{\text{lvls}-\text{lvl}})$
        if $B'_{\uparrow}$ is not null then
            $\bar{B}_{\downarrow}$.style $\leftarrow B'_{\uparrow}$
        end if
        $\Omega \leftarrow$ InitOccurrenceMap$(w_{\bar{A}_{\downarrow}}, h_{\bar{A}_{\downarrow}}, \text{NNF})$
        for iter $\leftarrow 1 \dots$ lvlIters do
            Vote$(\bar{B}_{\downarrow}.\text{style}, \text{NNF})$
            PatchMatch$(\text{pmIters}, \bar{A}_{\downarrow}, \bar{B}_{\downarrow}, \text{NNF}, \Omega)$
        end for
        if lvl $<$ lvls then
            $B'_{\uparrow} \leftarrow$ Upsample$(\bar{B}_{\downarrow}.\text{style}, 2)$
            NNF $\leftarrow$ Upsample$(\text{NNF}, 2)$
        end if
    end for
end function
```

2.4 PatchMatch

PatchMatch [3] is a state-of-the-art randomized algorithm that evaluates NNF. The algorithm is iterative, it may start from scratch on random assignments, or further refine their current state. It runs propagation and random search procedures for each patch of $\bar{B}$ in a scan-line order (left to right, top to bottom). Additionally, even iterations examine patches in a reversed scan-line order (right to left, bottom to top).

Propagation

The propagation procedure is based on the observation that correctly synthesized textures usually consist of *coherent areas* which are continuous chunks of the example texture. To encourage these areas, the propagation tries to improve the current assignment P'_c of the patch P_c with candidate $P'_{ls} = \text{NNF}(P_l) \odot (1;0)$ where $P_l = P_c \odot (-1;0)$ and candidate $P'_{ts} = \text{NNF}(P_t) \odot (0;1)$ where $P_t = P_c \odot (0;-1)$. The candidates keep the coherent areas of assignments of P_l and P_t. (see Fig. 2.4).

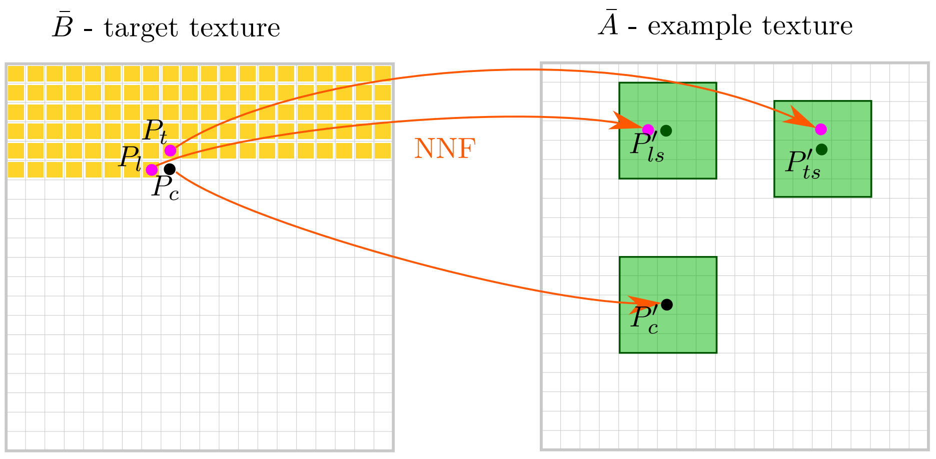

Figure 2.4: Propagation phase of PatchMatch where improving assignment for P_c (black dot in the target texture) is being searched. Assignments for previous target patches were already evaluated during the scan-line (displayed as yellow pixels in $\bar{B}$). The closest patch to P_c of the three green example patches P'_c, P'_{ls} and P'_{ts} will be its new assignment where P'_c is the currently assigned patch. P'_{ls} resp. P'_{ts} are candidate assignments that try to keep coherent assignment areas of P_l resp. P_t. Note that the image displays the phase in odd iteration, even iterations try to propagate assignments from bottom and right target patches.

Random Search

The stochastic part is based on the law of large numbers, where after a number of random assignments (that cover a reasonable fraction of the example texture) the probability of an incorrect assignment diminishes. Candidate assignments for patch P_c are randomly picked from square areas $A_i \in \{A_1, A_2, ...\}$ in the example texture $\bar{A}$. The areas are exponentially decreasing (halving) in width and are centered in the current assignment P'_c. The closest candidate to patch P_c that is also closer than the current assignment P'_c becomes the new assignment of P_c in NNF. The search starts in the largest area A_1 with radius set to $\max(B.\text{width}, B.\text{height})$ and continues while the area width is larger than zero (see Fig. 2.5).

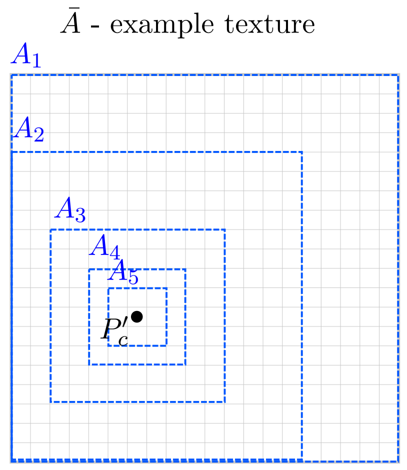

Figure 2.5: Candidate assignments for P_c are randomly picked from example texture areas A_i that shrink exponentially in radius and are centered in the current assignment P'_c. The closest candidate that is also closer than the current assignment P'_c will be the new assignment.

Both propagation and random search are used in the PatchMatch's CPU implementation as follows:

```
function PatchMatch(iters, $\bar{A}$, $\bar{B}$, NNF, $\Omega$)
    for $i \leftarrow 1 \dots iters$ do
        if $i$ is odd number then
            for $P \in \mathcal{N}_{\bar{B}}$ in left-right, top-bottom order do
                PropagateLeftTop($P, \bar{A}, \bar{B}, \mathrm{NNF}, \Omega$)
                RandomSearch($P, \bar{A}, \bar{B}, \mathrm{NNF}, \Omega$)
            end for
        else
            for $P \in \mathcal{N}_{\bar{B}}$ in right-left, bottom-top order do
                PropagateRightBottom($P, \bar{A}, \bar{B}, \mathrm{NNF}, \Omega$)
                RandomSearch($P, \bar{A}, \bar{B}, \mathrm{NNF}, \Omega$)
            end for
        end if
    end for
end function
```

Chapter 3

Implementation

In this chapter, we will describe the reference GPU implementation of our method. We will provide a pseudo code of NVIDIA CUDA kernels and the most important procedures. Before we start discussing the kernels in detail, the first section of this chapter will be a brief introduction to NVIDIA GPU architecture.

3.1 NVIDIA GPU Architecture [1]

Execution-wise, NVIDIA GPU architecture is designed to execute large amounts of *threads* in a parallel fashion. A single thread is an instance of a *kernel* which is a procedure operating on a single item and designed to be run in parallel on a large array of items. Execution of a kernel is always called from a code running on CPU. Threads are grouped into *blocks* and blocks are grouped into a *grid* that usually covers the whole array (blocks of size 8×8 are demonstrated in Fig. 5.1). The dimensions of a single block and grid are parameterizable (1D, 2D, 3D) which gives the user helpful level of abstraction. Each thread is able to evaluate its exact coordinates within the grid which are used to access items in the array. Each GPU consists of a number of *streaming multiprocessors* (SMs). SM employs *single instruction multiple thread* (SIMT) architecture model. During kernel's execution, all blocks are distributed among individual SMs in a uniform fashion. Each SM has *warp scheduler* that partitions blocks assigned to it into *warps* which it then executes. Warp is a group of 32 threads that execute the same instruction set (delivered per-instruction from warp scheduler) in a parallel lock-step fashion on a single SM. Therefore it is important to design kernels in a way that the threads of a single warp do not branch away from each other often as that causes thread divergence which decreases parallel execution efficiency.

Memory-wise, GPU has its own memory accessed by threads during kernel's execution that is not directly accessible from outside of GPU. Prior to kernel's execution, required data have to be copied into the GPU memory (in case they are not already present) and copied back once the execution is finished. This is a slow process and it is best to design the procedure that calls kernels in a way that minimizes these memory transfers. GPU consists of various memory types which differ in latency, bandwidth and size. *Register memory* is physically placed on a GPU chip. Every SM has its own set of 32-bit registers which are usually used for local variables of threads. Registers are the fastest memory available, but their number is very limited. Their usage directly implies how many warps can run in parallel on a single SM. *Global memory* has the highest latency as it is placed away from GPU chip and is not implicitly cached. Load requests to global memory that come from threads of the same warp are grouped into 32, 64, or 128-byte *transactions* under the condition that the requested memory segments are tightly consequential. This way of accessing global memory is also called *coalesced access* and it is a very important optimization part of GPU programming as it maximizes memory throughput by serving multiple load requests in parallel. *Shared memory* is in the middle (both in latency and size) between the former two. It is placed on the chip for each SM as well. Shared memory is distributed per-block which means that threads of the same block can communicate with each other using this memory. Due to bandwidth optimization access, shared memory is divided to *banks* which require special attention. The best case scenario is when every thread of a single warp accesses memory from a different bank which can be processed in parallel. In case multiple threads request different data from the same bank, the requests are processed sequentially. *Texture memory* is a part of global memory that is implicitly cached in the highest level cache. This memory is optimized for 2D spatially close access.

3.2 GPU Occurrence Map

The following kernels and GPU procedures describe handling of the occurrence map Ω during evaluation of NNF. *Procedure* in this context refers to a code run by GPU threads, unless otherwise stated. Note that a single patch of the target texture $\bar{B}$ is handled by a single thread that runs the kernel code.

Procedure *PatchOmega* evaluates the penalization term ρ of a single example patch Q.

```
function PatchOmega($Q$, $\Omega$)
    $\omega_{\text{average}} \leftarrow 0$
    $\omega_{\text{best}} \leftarrow w_Q^2 \frac{w_{\bar{B}} h_{\bar{B}}}{w_{\bar{A}} h_{\bar{A}}}$
    for $y \leftarrow -r_Q \dots r_Q$ do
        for $x \leftarrow -r_Q \dots r_Q$ do
            $\omega_{\text{average}} \leftarrow \omega_{\text{average}} + \Omega(u_Q + x, v_Q + y)$
        end for
    end for
    $\omega_{\text{average}} \leftarrow \frac{\omega_{\text{average}}}{w_Q^2}$
    return $\frac{\omega_{\text{average}}}{\omega_{\text{best}}}$
end function
```

Procedure *UpdatePatchOmega* increments or decrements elements in the occurrence map Ω that are part of the target patch Q. Collisions might emerge since this procedure writes to global memory and can be run on overlapping patches at the same time. Therefore, standard addition $+$ is replaced with atomic addition $\oplus$ to prevent undesirable race conditions between concurrent threads. The atomic solution decreases the performance, but the preservation of the occurrence map integrity is essential. The usual update values (parameter a) are $+1$ when new patch Q is assigned and -1 when removing old assignment Q.

```
function UpdatePatchOmega($Q$, $\Omega$, $a$)
    for $y \leftarrow -r_Q \dots r_Q$ do
        for $x \leftarrow -r_Q \dots r_Q$ do
            $\Omega(u_Q + x, v_Q + y) \leftarrow \Omega(u_Q + x, v_Q + y) \oplus a$
        end for
    end for
end function
```

The following procedure SSD evaluates the similarity term δ between patch P from the texture $\bar{B}$ and patch Q from the texture $\bar{A}$. The procedure slightly differs from the definition as the metric weights the guidance channels with *guidance influence* parameter μ, thus SSD is separated for guidance and style channels. Additionally, early termination (with the knowledge of the current assignment distance) can be applied in this procedure as well to improve its performance.

```
function SSD($P$, $Q$)
    $P_g \leftarrow \bar{B}$.guidance portion of $P$
    $P_s \leftarrow \bar{B}$.style portion of $P$
    $Q_g \leftarrow \bar{A}$.guidance portion of $Q$
    $Q_s \leftarrow \bar{A}$.style portion of $Q$
    $n_g \leftarrow n_{\bar{B}.\mathrm{guidance}}$
    $n_s \leftarrow n_{\bar{B}.\mathrm{style}}$
    res $\leftarrow 0$
    for $y \leftarrow 0 \ldots w_P - 1$ do
        for $x \leftarrow 0 \ldots w_P - 1$ do
            for $i \leftarrow 1 \ldots n_g$ do
                res $\leftarrow$ res + $\frac{\mu}{n_g}(f_i^{P_g}(x,y) - f_i^{Q_g}(x,y))^2$
            end for
            for $i \leftarrow 1 \ldots n_s$ do
                res $\leftarrow$ res + $\frac{1}{n_s}(f_i^{P_s}(x,y) - f_i^{Q_s}(x,y))^2$
            end for
        end for
    end for
    return res
end function
```

Finally, to envelop the whole metric φ, we use a procedure called *ImproveAssignment* that compares distance of the current assignment Q_{current} to patch P with distance of the candidate assignment Q_{new} to patch P. The closer assignment is returned. The procedure updates the occurrence map Ω in case the candidate is selected. This is the universal procedure that applies the metric φ between patches and accesses or makes any changes to the occurrence map Ω.

```
**function** ImproveAssignment($P$, $Q_{\text{new}}$, $Q_{\text{current}}$, $\Omega$)
    $\rho_{\text{current}} \leftarrow \text{PatchOmega}(Q_{\text{current}}, \Omega)$
    $\rho_{\text{new}} \leftarrow \text{PatchOmega}(Q_{\text{new}}, \Omega)$
    $\delta_{\text{current}} \leftarrow \text{SSD}(P, Q_{\text{current}})$
    $\delta_{\text{new}} \leftarrow \text{SSD}(P, Q_{\text{new}})$
    $\varphi_{\text{current}} \leftarrow \lambda\ \rho_{\text{current}} + \delta_{\text{current}}$
    $\varphi_{\text{new}} \leftarrow \lambda\ \rho_{\text{new}} + \delta_{\text{new}}$
    if $\varphi_{\text{new}} < \varphi_{\text{current}}$ then
        $\text{UpdatePatchOmega}(Q_{\text{new}}, \Omega, +1)$
        $\text{UpdatePatchOmega}(Q_{\text{current}}, \Omega, -1)$
        return $Q_{\text{new}}$
    else
        return $Q_{\text{current}}$
    end if
end function
```

3.3 GPU PatchMatch

The GPU implementation of PatchMatch is already included in the original publication [3]. Propagation and random search are now separated to individual kernels, which means they are no longer interleaved per-pixel, but rather per-texture. This change requires usage of redundant NNF$'$ in the propagation search since the kernel reads assignments of other patches which can become dirty due to concurrent updates. Therefore, the redundancy is necessary to preserve the integrity of assignments. The random search does not read assignments of other target patches and therefore does not require this redundancy. Propagation utilizes *jump flood scheme* [15], the distance to the neighboring target patches is halved in every iteration step until it is less than 1. In order for the information to be distributed most effectively, the jump distance should start as the larger dimension of the target texture $\bar{B}$. However, as stated by Barnes et al. [3], this is not needed and much smaller maximum jump is sufficient. The reference GPU implementation uses starting distance set to 4. The new CPU implementation of PatchMatch that launches propagation and random search kernels is as follows.

```
function GPU_PatchMatch(iters, $\bar{A}$, $\bar{B}$, NNF, NNF$'$, $\Omega$, $R$)
    for iter $\leftarrow 1 \ldots$ iters do
        Kernel_Propagation($\bar{A}$,$\bar{B}$,NNF,NNF$'$,$\Omega$,4) in parallel on $\bar{B}$
        Swap(NNF,NNF$'$)
        Kernel_Propagation($\bar{A}$,$\bar{B}$,NNF,NNF$'$,$\Omega$,2) in parallel on $\bar{B}$
        Swap(NNF,NNF$'$)
        Kernel_Propagation($\bar{A}$,$\bar{B}$,NNF,NNF$'$,$\Omega$,1) in parallel on $\bar{B}$
        Swap(NNF,NNF$'$)
        Kernel_RandomSearch($\bar{A}$,$\bar{B}$,NNF,$\Omega$,$R$) in parallel on $\bar{B}$
    end for
end function
```

Propagation

To overcome the inherently sequential propagation, Barnes et al. [3] implements the solution called *jump flood scheme* [15]. Jump flood is an iterative algorithm designed to broadcast information effectively across 2D data structure in parallel, i.e., to propagate patch assignments across NNF in order to encourage coherent areas. In a single iteration step, every target patch scans its four neighboring target patches that reside in the same distance d on horizontal and vertical axes. The candidates are selected in the same fashion as in the sequential version of the algorithm, the relative example patches are checked as candidates for improvements in order to try to keep the local assignment areas coherent. In case some candidate is closer than the current assignment, the occurrence map Ω is immediately updated based on this change using ImproveAssignment procedure. After iterating all candidates, the closest one (if it is closer that the current assignment) is stored in the redundant NNF′.

```
function KERNEL_PROPAGATION($\bar{A}$, $\bar{B}$, NNF, NNF′, $\Omega$, $d$)
    $P \leftarrow$ patch of this thread in $\bar{B}$
    $Q \leftarrow$ NNF($P$)
    $Q_c \leftarrow$ NNF($P \odot (d; 0)) \odot (-d; 0)$ clamped to $\bar{A}$ range
    $Q \leftarrow$ ImproveAssignment($P, Q_c, Q, \Omega$)
    $Q_c \leftarrow$ NNF($P \odot (0; d)) \odot (0; -d)$ clamped to $\bar{A}$ range
    $Q \leftarrow$ ImproveAssignment($P, Q_c, Q, \Omega$)
    $Q_c \leftarrow$ NNF($P \odot (-d; 0)) \odot (d; 0)$ clamped to $\bar{A}$ range
    $Q \leftarrow$ ImproveAssignment($P, Q_c, Q, \Omega$)
    $Q_c \leftarrow$ NNF($P \odot (0; -d)) \odot (0; d)$ clamped to $\bar{A}$ range
    $Q \leftarrow$ ImproveAssignment($P, Q_c, Q, \Omega$)
    NNF′($P$) $\leftarrow Q$
end function
```

Random Search

Random number generation works slightly differently in CUDA. Random numbers are generated using variables of a specific type called *curandState*, state of these variables changes after each generated number. Therefore, to ensure globally random numbers among all threads and kernel calls, each thread requires its own *curandState* instance initialized with a unique seed. These variables are stored in R. Similar as in the propagation, finding a closer candidate immediately causes updates to Ω by using ImproveAssignment procedure.

```
function KERNEL_RANDOMSEARCH($\bar{A}$, $\bar{B}$, NNF, $\Omega$, $R$)
    $P \leftarrow$ patch of this thread in $\bar{B}$
    $Q \leftarrow$ NNF($P$)
    $Q_0 \leftarrow Q$
    rand $\leftarrow R(u_P, v_P)$
    for $i \leftarrow 1 \ldots \infty$ do
        $r \leftarrow 2^{i-1}$
        if $r > \frac{\max(\bar{B}.\text{width}, \bar{B}.\text{height})}{2}$ then
            break
        end if
        $A_i \leftarrow$ square area centered in $Q_0$ with radius $r$ clamped to $\bar{A}$ range
        $Q_c \leftarrow$ GenerateRandomPatch($A_i$, rand)
        $Q \leftarrow$ ImproveAssignment($P, Q_c, Q, \Omega$)
    end for
    $R(u_P, v_P) \leftarrow$ rand
    NNF($P$) $\leftarrow Q$
end function
```

All the curandState variables must be initialized first with unique seeds, e.g., an id of the executing thread. The initialization is a slow process and is, therefore, run prior to the pyramid iteration using the following kernel Kernel_InitRandom.

```
function KERNEL_INITRANDOM($R$)
    $(x; y) \leftarrow$ coordinates of this thread in $R$
    $R(x, y) \leftarrow$ initialized $R(x, y)$ with thread-unique seed
end function
```

3.4 GPU Texture Optimization

Kernel_Vote is a kernel that handles voting in the M-step of the texture optimization technique.

```
function KERNEL_VOTE($B'$, NNF)
    $P \leftarrow$ patch of this thread in $B'$
    pix $\leftarrow$ zero vector of $n_{B'}$ channels
    for $y \leftarrow -r_P \dots r_P$ do
        for $x \leftarrow -r_P \dots r_P$ do
            $Q \leftarrow A'$ portion of NNF$(P \odot (x;y)) \odot (-x;-y)$
            for $i \leftarrow 1 \dots n_{B'}$ do
                $\text{pix}_i \leftarrow \text{pix}_i + f_i^Q(u_Q;v_Q)$
            end for
        end for
    end for
    $B'(u_P, v_P) \leftarrow \frac{\text{pix}}{w_P^2}$
end function
```

Now we put together all the described kernels and procedures to finalize the reference GPU implementation of our texture synthesis method. Source code for kernels Kernel_Downsample and Kernel_Upsample is not provided as it is out of the scope of this text. The textures are sampled with basic bilinear interpolation as it has proved to be sufficient in our scenario. GPU_TextureOptimization procedure is executed on CPU and launches proper kernels described earlier in this chapter. The procedure InitOccurrenceMap stays implemented on CPU as its parallel implementation would cause a lot of atomic concurrencies and the resulting contribution would be insignificant since it is run only once per level. To keep things simple, some of the constant parameters are omitted (e.g., λ, μ, w_P, etc.) and are considered as global.

```
function GPU_TextureOptimization(lvls, iters, pmIters, $\bar{A}$, $\bar{B}$)
    $R \leftarrow$ 2D array of curandState instances of size $\bar{B}$
    Kernel_InitRandom($R$) in parallel on $R$
    $B'_\uparrow \leftarrow$ null
    NNF $\leftarrow$ Kernel_RandomNNF($(w_{\bar{B}}, h_{\bar{B}}), (w_{\bar{A}}, h_{\bar{A}}), R$) in parallel on NNF
    NNF$'$ $\leftarrow$ Create redundant copy of NNF
    for lvl $\leftarrow 1 \ldots$ lvls do
        $\bar{A}_\downarrow \leftarrow$ Kernel_Downsample($\bar{A}, 2^{\text{lvls}-\text{lvl}}$) in parallel on $\bar{A}_\downarrow$
        $\bar{B}_\downarrow \leftarrow$ Kernel_Downsample($\bar{B}, 2^{\text{lvls}-\text{lvl}}$) in parallel on $\bar{B}_\downarrow$
        if $B'_\uparrow$ is not null then
            $\bar{B}_\downarrow$.style $\leftarrow B'_\uparrow$
        end if
        $\Omega \leftarrow$ InitOccurrenceMap($w_{\bar{A}_\downarrow}, h_{\bar{A}_\downarrow}$, NNF)
        for iter $\leftarrow 1 \ldots$ iters do
            Kernel_Vote($\bar{B}_\downarrow$.style, NNF) in parallel on $\bar{B}_\downarrow$
            GPU_PatchMatch(pmIters, $\bar{A}_\downarrow, \bar{B}_\downarrow$, NNF, NNF$'$, $\Omega$, $R$)
        end for
        if lvl $<$ lvls then
            $B'_\uparrow \leftarrow$ Kernel_Upsample($\bar{B}_\downarrow$.style, 2) in parallel on $B'_\uparrow$
            NNF $\leftarrow$ Kernel_Upsample(NNF, 2) in parallel on NNF
            NNF$'$ $\leftarrow$ Create redundant copy of NNF
        end if
    end for
end function
```

Chapter **4**

Results

In this chapter we will compare the reference GPU implementation (Ref) with the CPU implementation of StyLit [6]. The comparison will be done in both the quality of results and performance.

4.1 Configuration

The following configuration parameters are the same for both algorithms we are comparing and also for the further improvements algorithms in the next chapter:

Patch width	5
Pyramid levels	6
Votes per level	6
PatchMatch iterations	6
Guidance influence μ	2

The configuration of CUDA kernel launches and type of used GPU memory is important to mention as well. We have used texture memory for data that are often read from, but only once in a while written to as that is an ideal scenario to take advantage of the cache. Global memory is for data which require interleaved read and write operations.

Block size	32×32
Texture memory	$\bar{A}$, $\bar{B}$, NNF, NNF$'$
Global memory	R, Ω

StyLit [6] results were generated on CPU, both Ref and our further improvement results were generated on GPU which kernels were launched in CPU code:

StyLit CPU	3GHz, 4 cores
Our/Ref CPU	3.5GHz, 4 cores
Our/Ref GPU	GeForce GTX 660
OS	64-bit Windows 7

4.2 Comparison

The following figures show the results of Ref with increasing λ parameter and their comparison to the results of StyLit [6]. Stylization textures A' are various paintings created by two artists using different methods. Example guidance textures $A_1, \ldots, A_5$ are renderings of the same 3D model: a sphere. The simple shape of the sphere is used here so that the artist can draw the stylization without much effort. Target guidance textures $B_1, \ldots, B_5$ are renderings of various 3D models with the same light source positions and viewpoints as in the example renderings. Source of all guidance textures, as well as the results of StyLit, is Fišer et al. [6] All the results are of the same resolution 1200×912, the examples are of resolutions 1200×1100 or 1200×912.

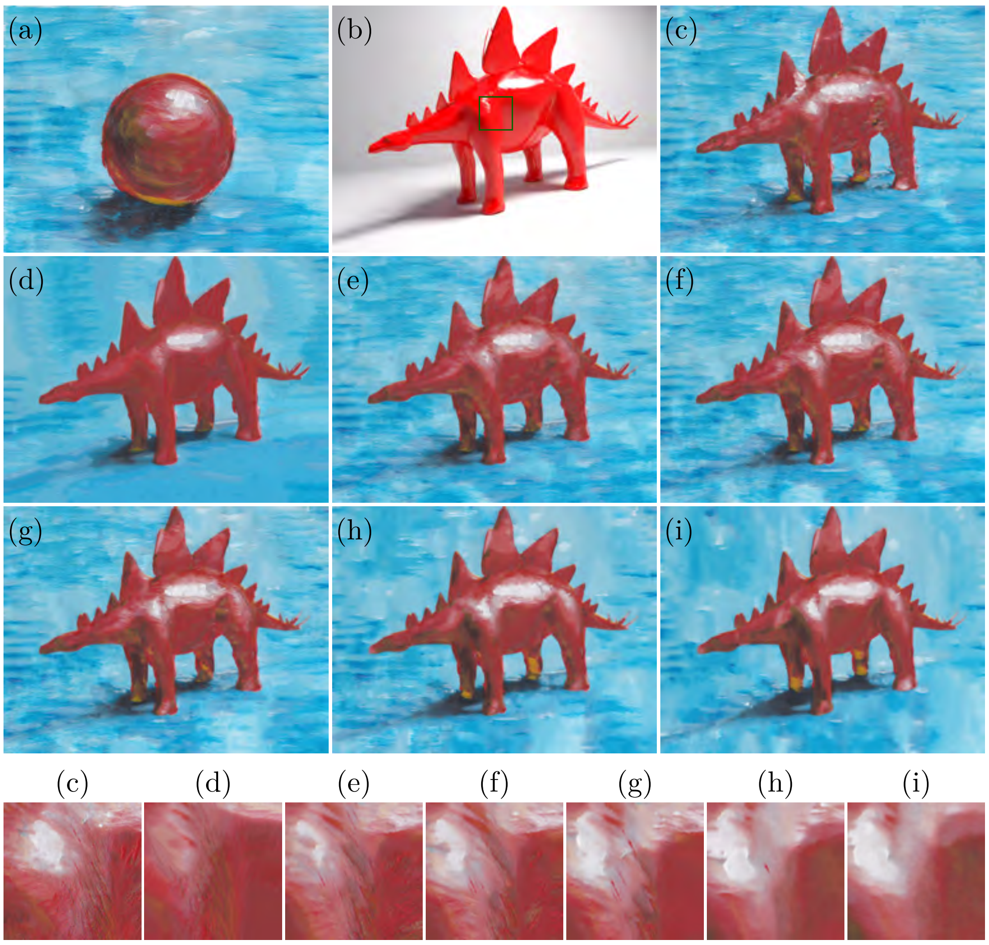

Figure 4.1: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) . . . (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2453	-
(d)	0	26.009	94.314
(e)	5000	46.890	52.314
(f)	10000	55.758	43.994
(g)	15000	64.957	37.763
(h)	50000	116.237	21.103
(i)	100000	143.980	17.037

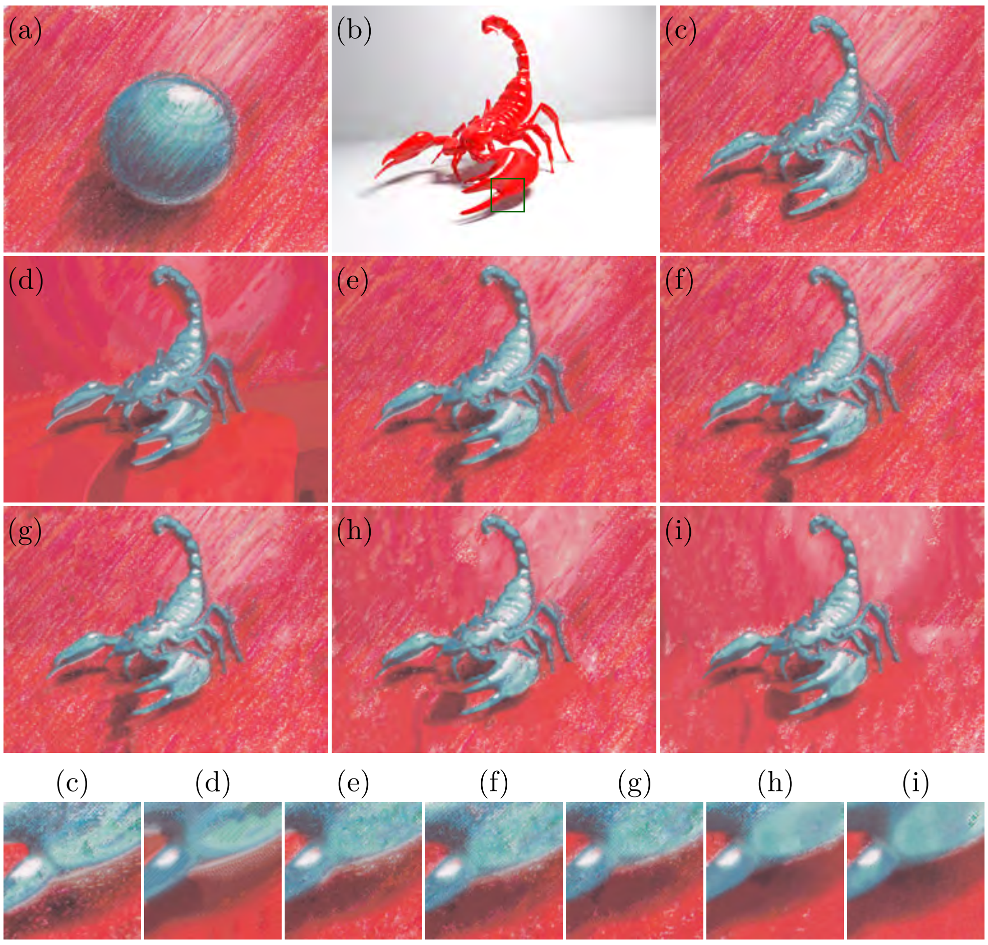

Figure 4.2: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) … (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	1628	-
(d)	0	25.791	63.123
(e)	5000	47.604	34.199
(f)	10000	50.310	32.359
(g)	15000	53.807	30.256
(h)	50000	106.620	15.269
(i)	100000	148.156	10.988

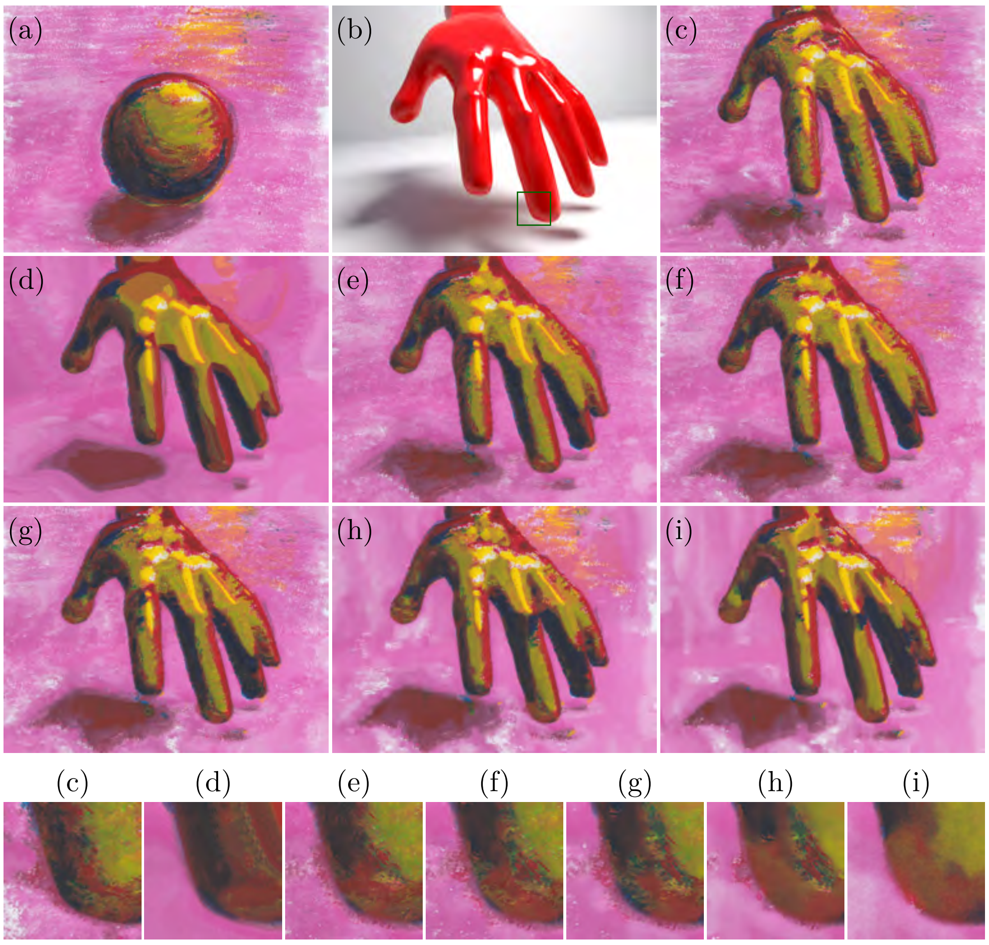

Figure 4.3: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) ... (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2139	-
(d)	0	25.259	84.683
(e)	5000	48.643	43.973
(f)	10000	59.543	35.923
(g)	15000	70.381	30.392
(h)	50000	118.645	18.029
(i)	100000	140.747	15.197

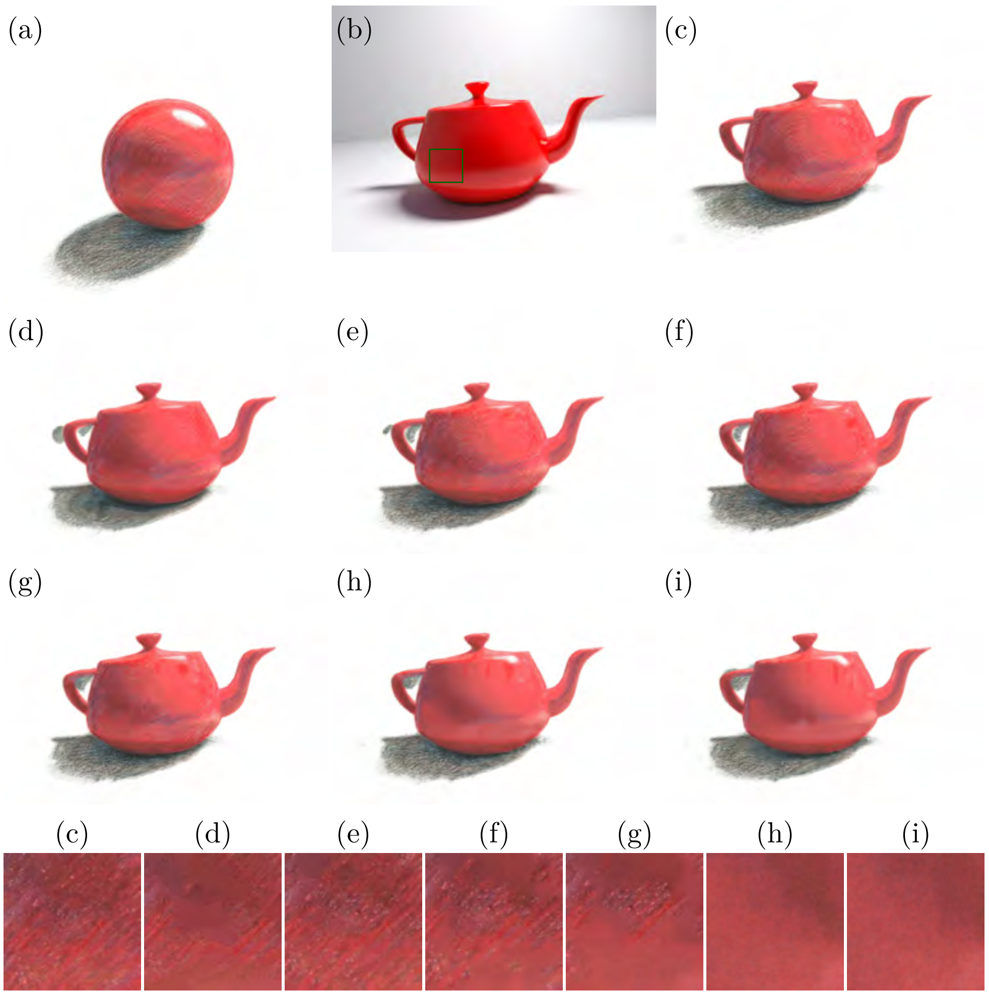

Figure 4.4: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) . . . (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	1714	-
(d)	0	27.396	62.534
(e)	5000	98.881	17.334
(f)	10000	113.362	15.120
(g)	15000	121.676	14.087
(h)	50000	144.630	11.850
(i)	100000	156.359	10.961

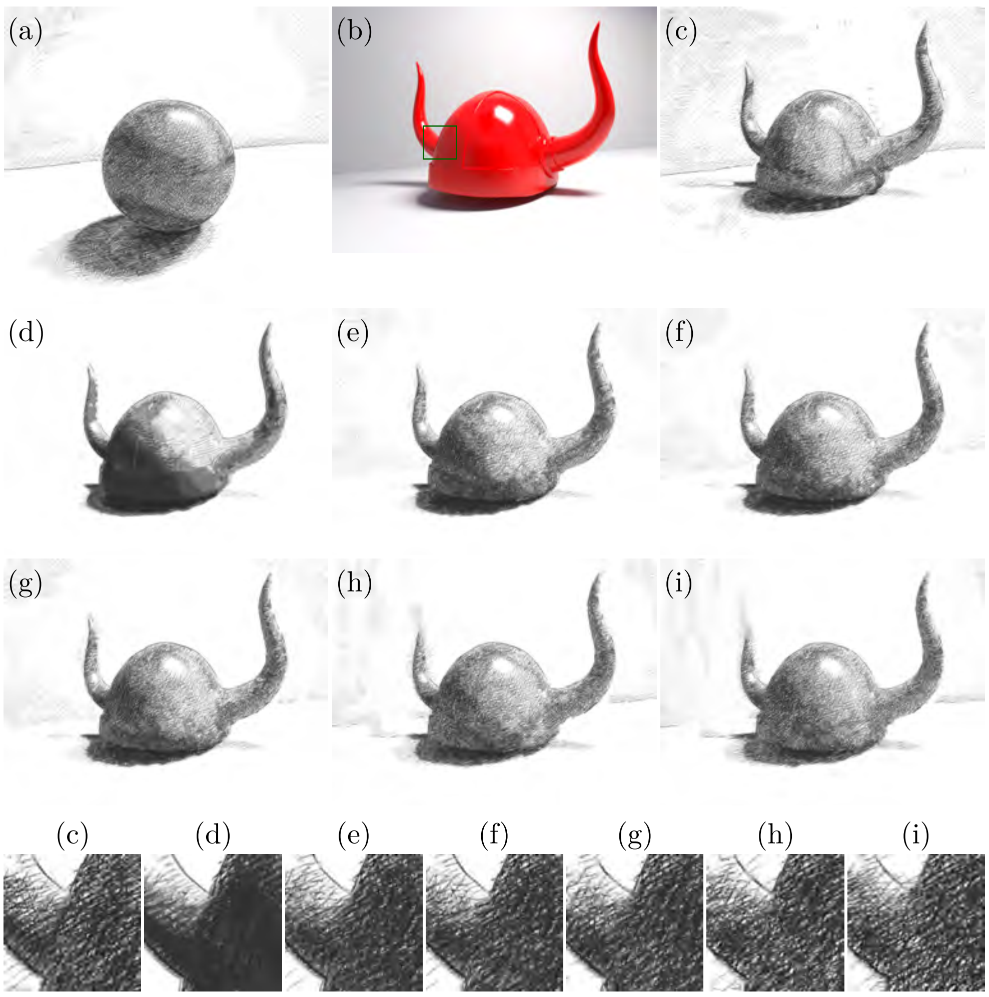

Figure 4.5: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) . . . (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2663	-
(d)	0	28.116	94.714
(e)	5000	96.276	27.660
(f)	10000	107.939	24.671
(g)	15000	114.586	23.240
(h)	50000	137.384	19.384
(i)	100000	148.154	17.975

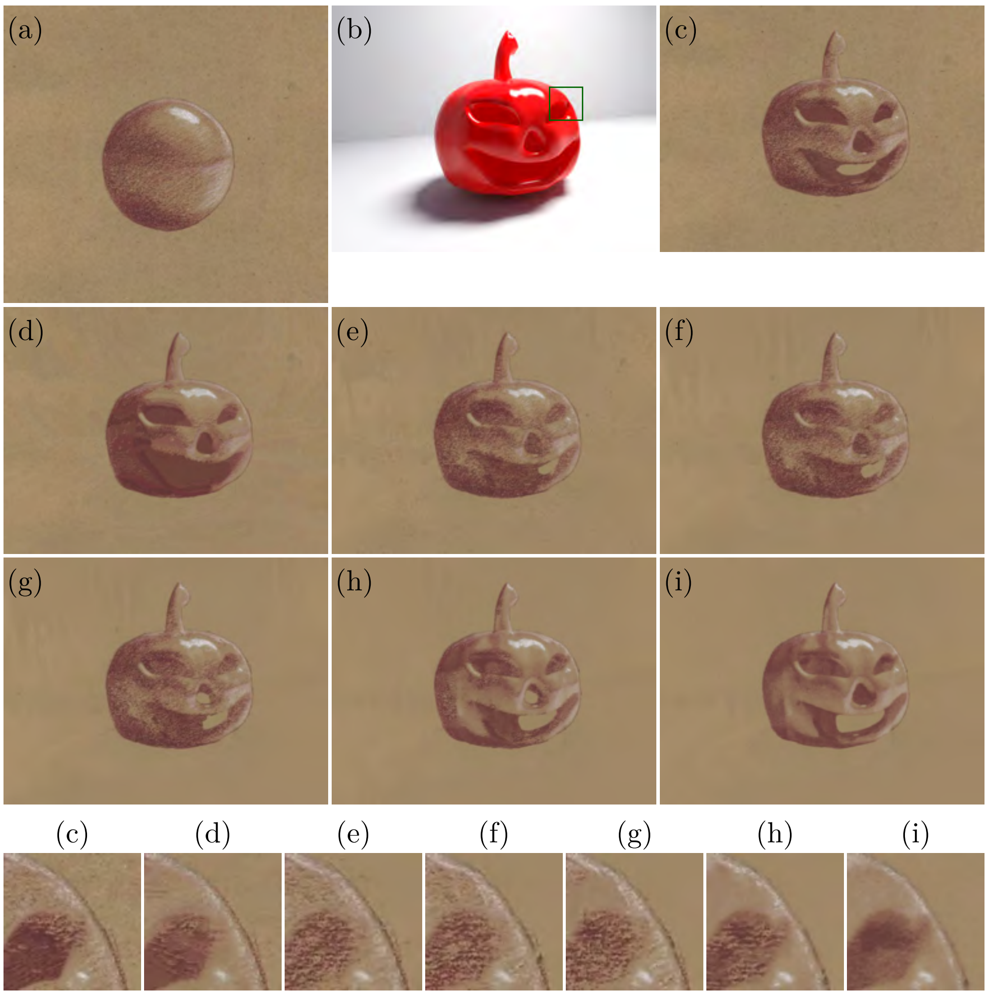

Figure 4.6: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) ... (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	1796	-
(d)	0	24.652	72.854
(e)	5000	87.600	20.502
(f)	10000	109.294	16.433
(g)	15000	118.872	15.109
(h)	50000	148.641	12.083
(i)	100000	162.076	11.081

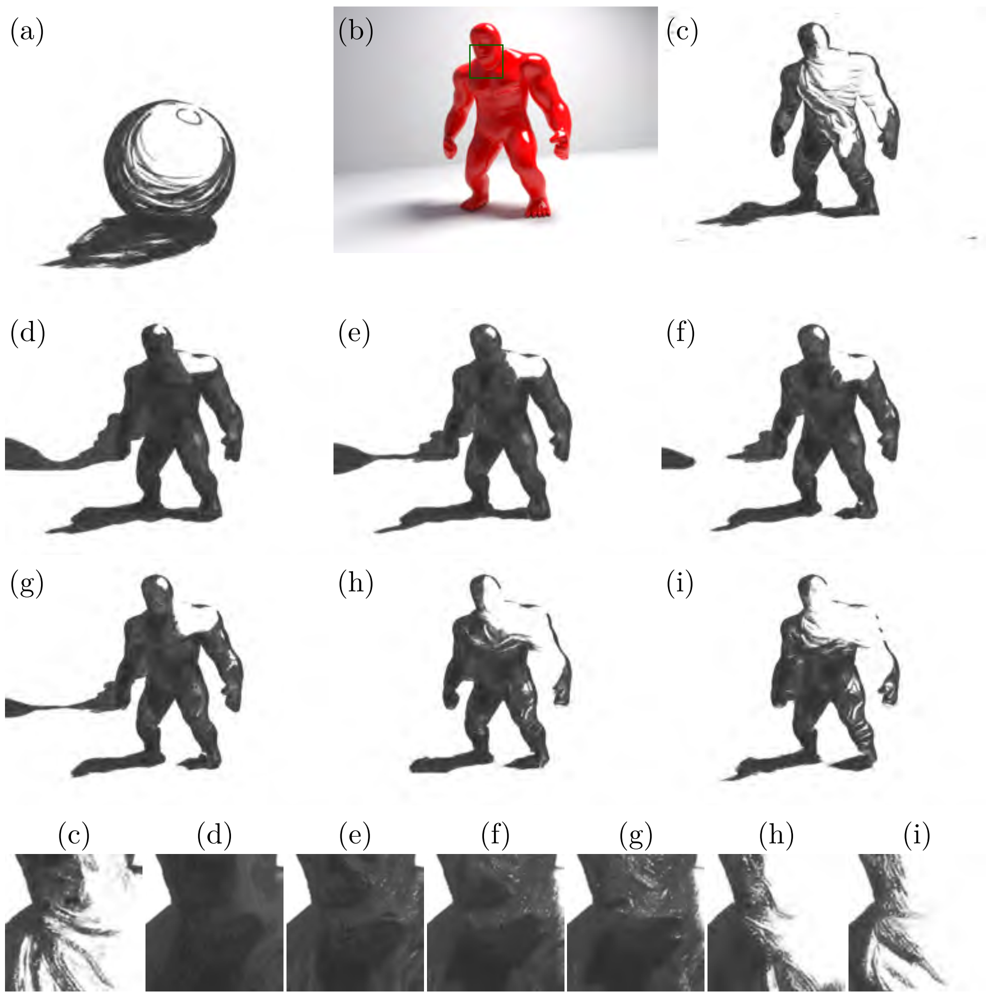

Figure 4.7: Comparison of results of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) . . . (i) Ref with increasing λ.

	λ	time [s]	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(x)}}$
(c)	-	2561	-
(d)	0	28.903	88.607
(e)	5000	107.389	23.848
(f)	10000	122.495	20.907
(g)	15000	130.712	19.593
(h)	50000	154.068	16.623
(i)	100000	163.317	15.681

4.3 Discussion

As we can see, the value of λ parameter has a great impact on quality of the results. The best results are usually achieved with λ in the range [5000; 15000] with evaluation times being $15 - 50\times$ faster than those of StyLit [6]. In quality comparison, the results of the reference GPU implementation come very close to those of StyLit. To the untrained eye, some of them might seem even identical on the first sight. However, the finest details are not preserved (e.g., notable trails left by the paintbrush bristles in Fig. 4.1, discontinuities of the wax paint in Fig. 4.3, or cardboard "grains" in Fig. 4.6). The differences are acceptable considering the performance benefits we gained. In practice, when using today's high-end desktop NVIDIA GPU (e.g., GTX 1080) the reference GPU implementation without our further improvements (see Chap. 5) is able to generate the same results in around 250ms. To our disadvantage, to achieve the best possible results, we would have to fine-tune the λ value for each input scenario separately.

The (d) results with $\lambda = 0$ are included only to demonstrate cases with the occurrence map [11] "turned off". They are what we would expect from the texture optimization technique alone [12, 19]. In fact, we can already see the mentioned wash-out effect [13] taking place. Note that in case $\lambda = 0$, the memory of Ω is still being accessed which slows down the algorithm. In case of the reference GPU implementation build purified of any occurrence map routines the performance would be even better (1.5 - 2.5$\times$ faster) with results nevertheless the same.

The (h) and (i) results with the largest λ values (50k and 100k) start to display problems as well. Performance notably decreases with growing λ as convergence to NNF is harder to reach. Any changes to NNF greatly affect pixel weights in Ω which leads to even more reshuffling of assignments. Also, in theory, as the value of λ increases, the behavior of the algorithm comes closer to the approach that enforces uniform patch assignments [10].

Chapter 5

Further Improvements

In this chapter we will describe our experiments that improve the performance of the reference GPU implementation (Ref) even further. The improvements are implementational as well as algorithmic.

5.1 Shared Memory

Texture memory has an advantage of being implicitly cached, but we cannot directly affect it or rely on its behavior. This improvement aims to create our own cache memory in order to eliminate cache misses or its possible occupation by unrelated data. For this purpose, we use shared memory.

5.1.1 Method

The PatchMatch [3] kernels are executed with blocks of 32×32 threads which means that each block takes care of a square of 32×32 assignments in NNF. During execution of PatchMatch, the target texture $\bar{B}$ and the example texture $\bar{A}$ are in read-only mode as they are used only to evaluate NNF's metric φ. Values of target patches that belong to a single block are repeatedly read many times by the threads in propagation and random search when evaluating the metric with candidate example patches.

We can easily determine a square area of 36×36 pixels of the target texture $\bar{B}$ accessed by all patches of a block (see Fig. 5.1). The square area of the target $\bar{B}$ with its underlying 18 channels can be fitted as a whole into the shared memory of a single block. Both the propagation and random search kernels will first start with synchronously filling the shared memory and then continue with their work no longer requiring access to $\bar{B}$, but instead using the shared memory which grants us faster access to the required data at all times without relying on the internal cache of the texture memory. Shared memory "simulates" the cache in a private way that never misses or gets dirty by different data. Unfortunately, this trick only applies to the target

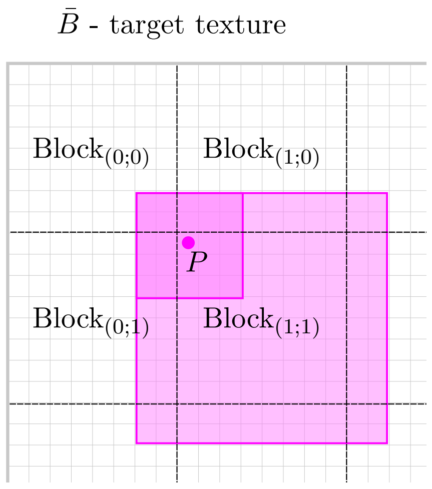

Figure 5.1: The large purple square of size 12×12 pixels displays area covered by all target patches that belong to threads from Block$_{(1;1)}$, in a case where the blocks are of size 8×8 threads and the patches are of width 5. The whole area of both stylization and guidance textures is stored in the shared memory. P is a top-left patch that demonstrates overlapping of the actual block area near its borders.

texture $\bar{B}$. In case of the example texture $\bar{A}$, the example patches assigned to the target patches of a single block can be randomly scattered all over the example texture (see how scattered the assignments in NNF can be in Fig. 2.2) and can also change a lot during kernel's execution. Of course, we want the assignments to be as much coherent as possible, but we are still unable to effectively determine which example pixels will be read more than once.

5.1.2 Implementation

The following algorithm shows the additional routine in the beginning of the random search and propagation kernels where all the threads of the block synchronously fill the shared memory. Line 8 fills the $(0;0) - (31;31)$ portion of the shared memory. Line 10 fills the $(0;32) - (31;35)$ portion, line 12 fills $(32;32) - (35;35)$ portion and finally line 16 fills $(32;0) - (35;31)$ portion of the shared memory. Line 18 synchronizes all the thread blocks to wait until all of them are finished. This way it is ensured that the shared memory has been filled as a whole with proper texture data and can be used by any thread of the block in the rest of the kernel's execution.

```
... *PatchMatch kernel with parameters including $\bar{B}$, $\bar{A}$, NNF,...*

$P \leftarrow$ patch of this thread in $\bar{B}$
$(x_r; y_r) \leftarrow$ coordinates of this thread relative to its block
$S \leftarrow$ shared memory of size $36{\times}36{\times}(n_{\bar{B}})$
$x \leftarrow u_P - 2$
$y \leftarrow v_P - 2$
$S(x_r, y_r) \leftarrow \bar{B}(x, y)$
if $y_r < 4$ then
    $S(x_r, y_r + 32) \leftarrow \bar{B}(x, y + 32)$
    if $x_r < 4$ then
        $S(x_r + 32, y_r + 32) \leftarrow \bar{B}(x + 32, y + 32)$
    end if
end if
if $x_r < 4$ then
    $S(x_r + 32, y_r) \leftarrow \bar{B}(x + 32, y)$
end if
__syncthreads()

... *rest of the kernel code, where S substitutes $\bar{B}$* ...
```

5.1.3 Comparison

Since the shared memory improvement is purely implementational as it only enables faster access to the same data of $\bar{B}$, there is no need to provide resulting textures B' as they would be the same as in the results of Ref. The input target and style pairs are the same. To keep the table short we only specify the targets in the table of results. The comparison is done purely by execution times to Ref with $\lambda = 10000$ and StyLit [6]:

target	Our [s]	Ref [s]	StyLit [s]	$\frac{\text{Ref}}{\text{Our}}$	$\frac{\text{StyLit}}{\text{Our}}$
stego	47.983	55.758	2453	1.162	51.122
scorpion	42.827	50.310	1628	1.175	38.013
hand	50.845	59.543	2139	1.171	42.069
teapot	104.231	113.362	1714	1.088	16.444
helmet	99.862	107.939	2663	1.081	26.667
pumpkin	100.550	109.294	1796	1.087	17.862
golem	114.372	122.495	2561	1.071	22.392

5.1.4 Discussion

The column $\frac{\text{Ref}}{\text{Our}}$ shows that this improvement decreased the time consumed compared to Ref's implementation only by $7 - 18\%$. The final percentage decreases even further as the performance comes closer to real-time (i.e., when running on today's high-end GPUs or when used on lower resolutions). Still, the usage of shared memory gives us some small advantage considering the resulting textures are not affected in any way.

5.2 Guide Maps

One of the most expensive parts of the reference GPU implementation (Ref) is overhead of reading from texture memory during evaluation of SSD between patches in PatchMatch [3] as each thread repeatedly reads w_P^2 pixel values from textures $\bar{B}$ and $\bar{A}$. Since the guidance textures $\bar{A}$.guidance and $\bar{B}$.guidance are synthetic renderings, the RGB values of their nearby pixels are often quite similar. We will try to estimate their SSD by using less pixel information to reduce required texture memory access.

5.2.1 Method

The idea of this improvement is to approximate guidance portions of SSD of two patches by using only their precomputed per-channel mean values.

$$\mathrm{SSD}(P,Q) = w_P^2 \sum_{i=1}^{n} (mean_i(P) - mean_i(Q))^2$$

$$mean_i(P) = \frac{1}{w_P^2} \sum_{x=0}^{w_P-1} \sum_{y=0}^{w_P-1} f_i^P(x,y)$$

where P is a patch of $\bar{B}$.guidance and Q is a patch of $\bar{A}$.guidance. Stylization textures $\bar{B}$.style and $\bar{A}$.style have to use the standard SSD that iterates through the whole patch as they are not synthetic renderings and RGB values of their nearby pixels often vary a lot.

However, we cannot use this technique on arbitrary patches as we would lose important information in patches containing edges or other areas of rapidly changing intensity. In order to distinguish these specific patches, we first evaluate sum of per-channel variances for all patches in the target guidance $\bar{B}$.guidance:

$$\mathrm{var}(P) = \sum_{i=1}^{n} \frac{1}{w_P^2} \sum_{x=0}^{w_P-1} \sum_{y=0}^{w_P-1} (mean_i(P) - f_i^P(x,y))^2$$

The sums are then used to create a new texture called *guide map* (black&white texture). The black pixels mean that their corresponding patches will use standard SSD evaluation while the white pixels will cause the patches to use the new approximation of SSD based on their arithmetic means. To create the new map, the evaluated sums of variances are compared to user-provided

ϵ value where values below the ϵ imply that the patch pixels are considered to be of similar value and are white on the map.

It is important to note the drawbacks of this approach. Threads of a single warp will not necessarily have the same value in the guide map. This leads to their instruction set divergence as some of the threads quickly use the new mean values and then have to wait for others to crawl through all of w_P^2 pixels in order to evaluate the standard SSD. However, as we will see a number of black pixels in the map get sparse very fast with increasing ϵ while the quality of results is maintained. Also, extra GPU memory is required to store the guide map and the pre-evaluated means for both target and example guidance textures. Plus, there is still an overhead of reading pixel weights from the occurrence map [11] where we cannot use this trick as their exact values are crucial for our method to properly encourage uniform patch assignments.

5.2.2 Implementation

At the start of each pyramid level, the mean textures have to be evaluated for both guidance textures $\bar{B}$.guidance and $\bar{A}$.guidance. The guide map is evaluated at the start of the level as well, but only for $\bar{B}$.guidance. The following kernel Evaluate_Means evaluates the mean texture M_X for a guidance texture X:

```
function EVALUATE_MEANS($X$, $M_X$)
    $P \leftarrow$ patch of this thread in $X$
    sum $\leftarrow$ zero vector of $n_X$ channels
    for $y \leftarrow -r_P \dots r_P$ do
        for $x \leftarrow -r_P \dots r_P$ do
            for $i \leftarrow 1 \dots n_X$ do
                $\text{sum}^i \leftarrow \text{sum} + f_P(x, y)$
            end for
        end for
    end for
    $M_X(u_P, v_P) \leftarrow \frac{\text{sum}}{w_P^2}$
end function
```

The guide map G is created by evaluating sum of variances over all patch channels using already evaluated texture of target guidance means M_B and then comparing the sums to ϵ value. The kernel of the Evaluate_Map is as follows:

```
function Evaluate_Map($\bar{B}$, $M_B$, $\epsilon$, $G$)
    $P \leftarrow$ patch of this thread in $\bar{B}$.guidance
    mean $\leftarrow M_B(u_P, v_P)$
    sum $\leftarrow$zero vector of $n_{\bar{B}}$ channels
    var $\leftarrow 0$
    for $y \leftarrow -r_P \ldots r_P$ do
        for $x \leftarrow -r_P \ldots r_P$ do
            for $i \leftarrow 1 \ldots n_{\bar{B}}$ do
                $\text{sum}_i \leftarrow \text{sum}_i + (f_i^P(x,y) - \text{mean}_i)^2$
            end for
        end for
    end for
    for $i \leftarrow 1 \ldots n$ do
        var $\leftarrow$ var $+ \frac{\text{sum}_i}{w_P^2}$
    end for
    if var $< \epsilon$ then
        $G(u_P, v_P) \leftarrow$ white
    else
        $G(u_P, v_P) \leftarrow$ black
    end if
end function
```

We use the new textures M_B, M_A and G in our improved SSD evaluation function.

```
function SSD($P$, $Q$, $M_B$, $M_A$, $G$)
    $P_g \leftarrow \bar{B}$.guidance portion of $P$
    $P_s \leftarrow \bar{B}$.style portion of $P$
    $Q_g \leftarrow \bar{A}$.guidance portion of $Q$
    $Q_s \leftarrow \bar{A}$.style portion of $Q$
    $n_g \leftarrow n_{\bar{B}.\text{guidance}}$
    $n_s \leftarrow n_{\bar{B}.\text{style}}$
    res $\leftarrow 0$
    if $G(u_P, v_P)$ is white then
        mean$^B \leftarrow M_B(u_P, v_P)$
        mean$^A \leftarrow M_A(u_Q, v_Q)$
        for $i \leftarrow 1 \dots n_g$ do
            res $\leftarrow$ res $+ \frac{\mu}{n_g}(\text{mean}_i^B - \text{mean}_i^A)^2$
        end for
        res $\leftarrow$ res $w_P^2$
    else
        for $y \leftarrow 0 \dots w_P$ do
            for $x \leftarrow 0 \dots w_P$ do
                for $i \leftarrow 1 \dots n_g$ do
                    res $\leftarrow$ res $+ \frac{\mu}{n_g}(f_i^{P_g}(x, y) - f_i^{Q_g}(x, y))^2$
                end for
            end for
        end for
    end if
    for $y \leftarrow 0 \dots w_P$ do
        for $x \leftarrow 0 \dots w_P$ do
            for $i \leftarrow 1 \dots n_s$ do
                res $\leftarrow$ res $+ \frac{1}{n_s}(f_i^{P_s}(x, y) - f_i^{Q_s}(x, y))^2$
            end for
        end for
    end for
    return res
end function
```

5.2.3 Comparison

The results of our new improvement with increasing ϵ parameter are compared to the results of Ref with $\lambda = 10000$ and StyLit. Parameter λ in our method is set to 10000 as well to simplify the comparison. The results display corresponding guide maps for selected ϵ values as well.

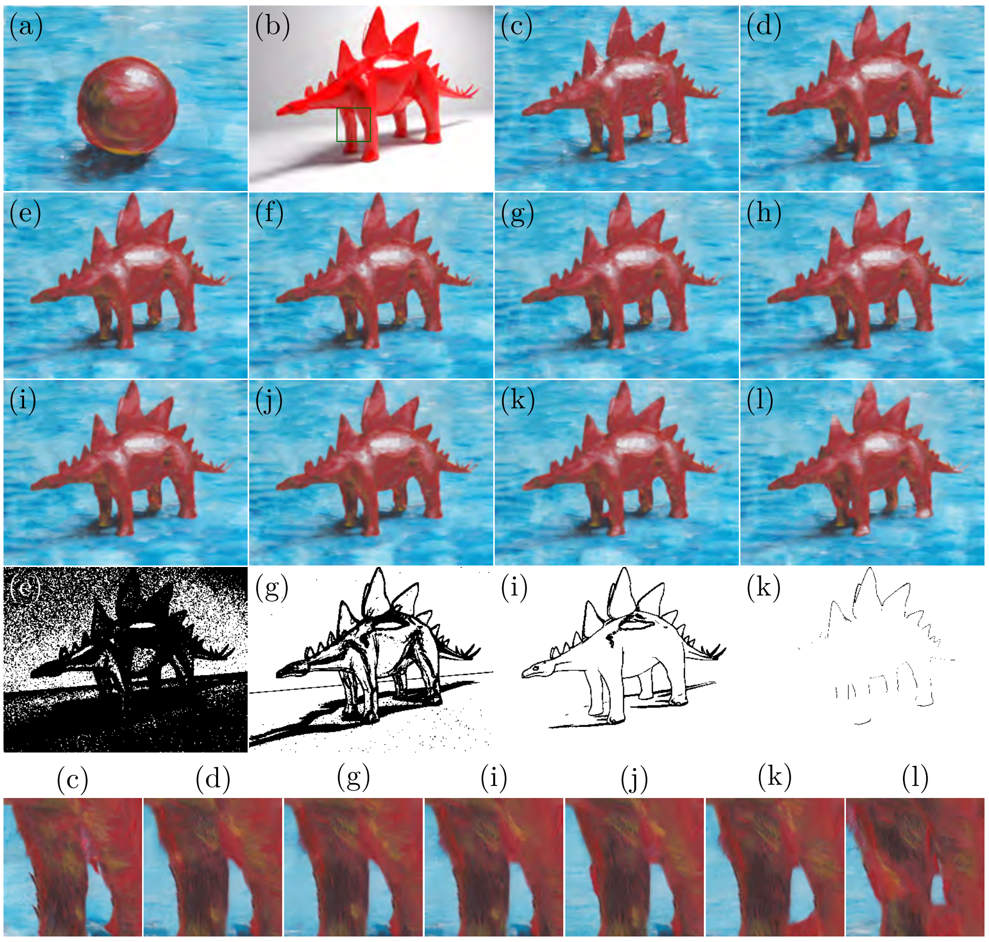

Figure 5.2: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) … (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(\text{x})}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2453	-	-
(d)	-	55.758	-	43.994
(e)	3	45.939	1.214	53.397
(f)	5	33.379	1.670	73.489
(g)	10	25.656	2.173	95.611
(h)	25	22.404	2.489	109.489
(i)	100	20.266	2.751	121.040
(j)	500	19.461	2.865	126.047
(k)	5000	18.404	3.030	133.286
(l)	10000	17.505	3.185	140.131

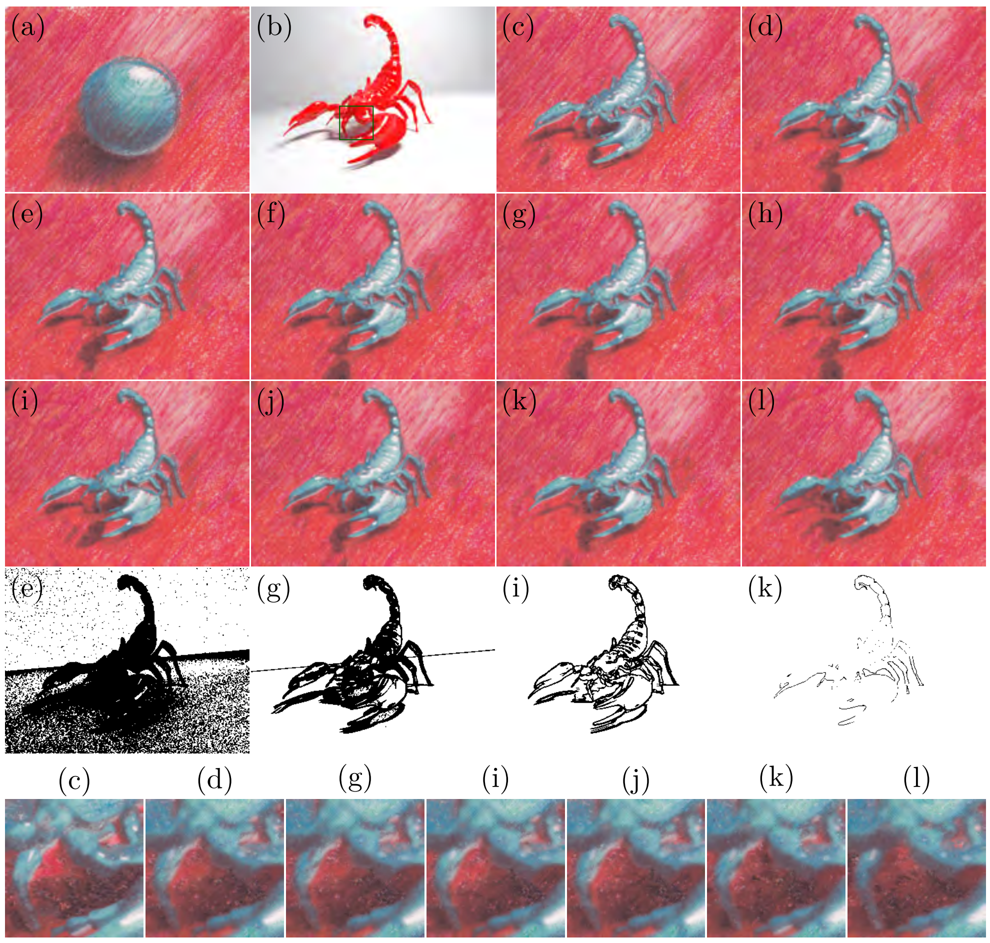

Figure 5.3: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) … (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(\text{x})}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	1628	-	-
(d)	-	50.310	-	32.359
(e)	3	33.358	1.508	48.804
(f)	5	25.050	2.008	64.990
(g)	10	22.003	2.287	73.990
(h)	25	20.930	2.404	77.783
(i)	100	19.090	2.635	85.280
(j)	500	17.568	2.864	92.668
(k)	5000	16.234	3.099	100.283
(l)	10000	15.408	3.265	105.659

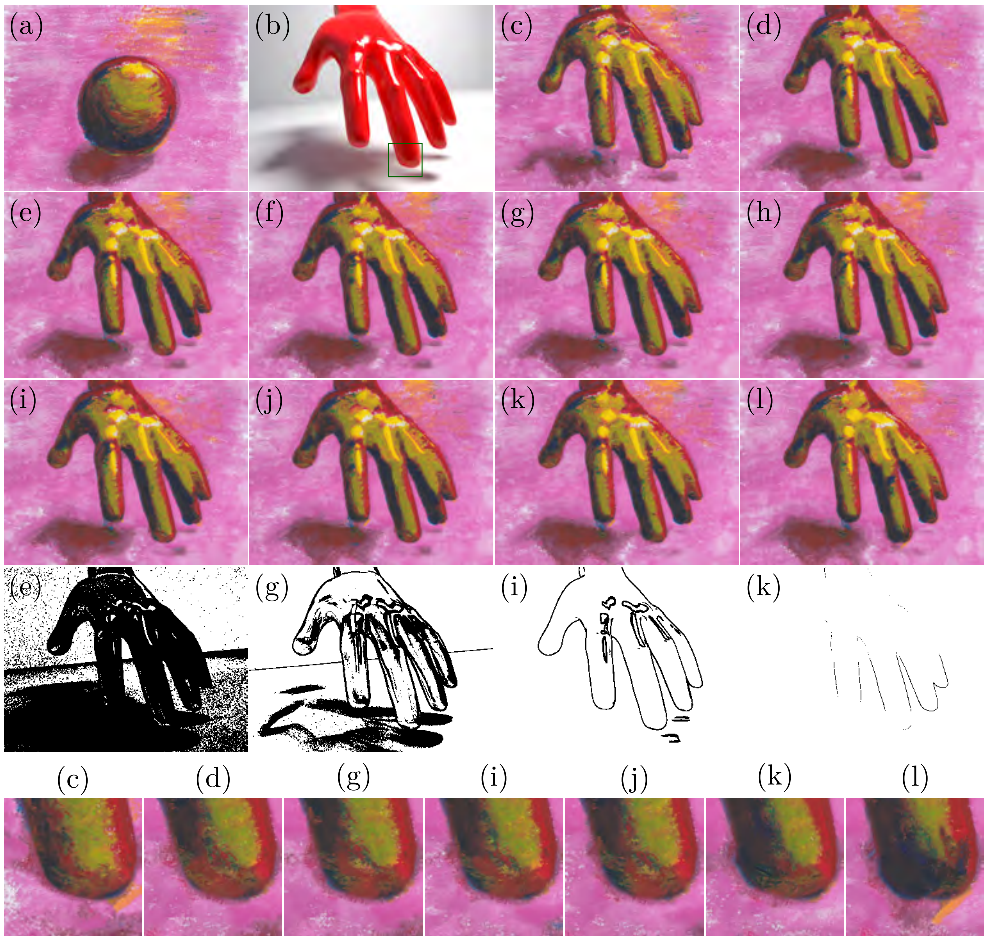

Figure 5.4: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Pavla Sýkorová. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) . . . (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(\text{x})}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2139	-	-
(d)	-	59.543	-	35.924
(e)	3	42.261	1.409	50.614
(f)	5	34.875	1.707	61.333
(g)	10	27.648	2.154	77.365
(h)	25	22.790	2.613	93.857
(i)	100	20.911	2.847	102.291
(j)	500	20.339	2.928	105.167
(k)	5000	19.250	3.093	111.117
(l)	10000	18.577	3.205	115.142

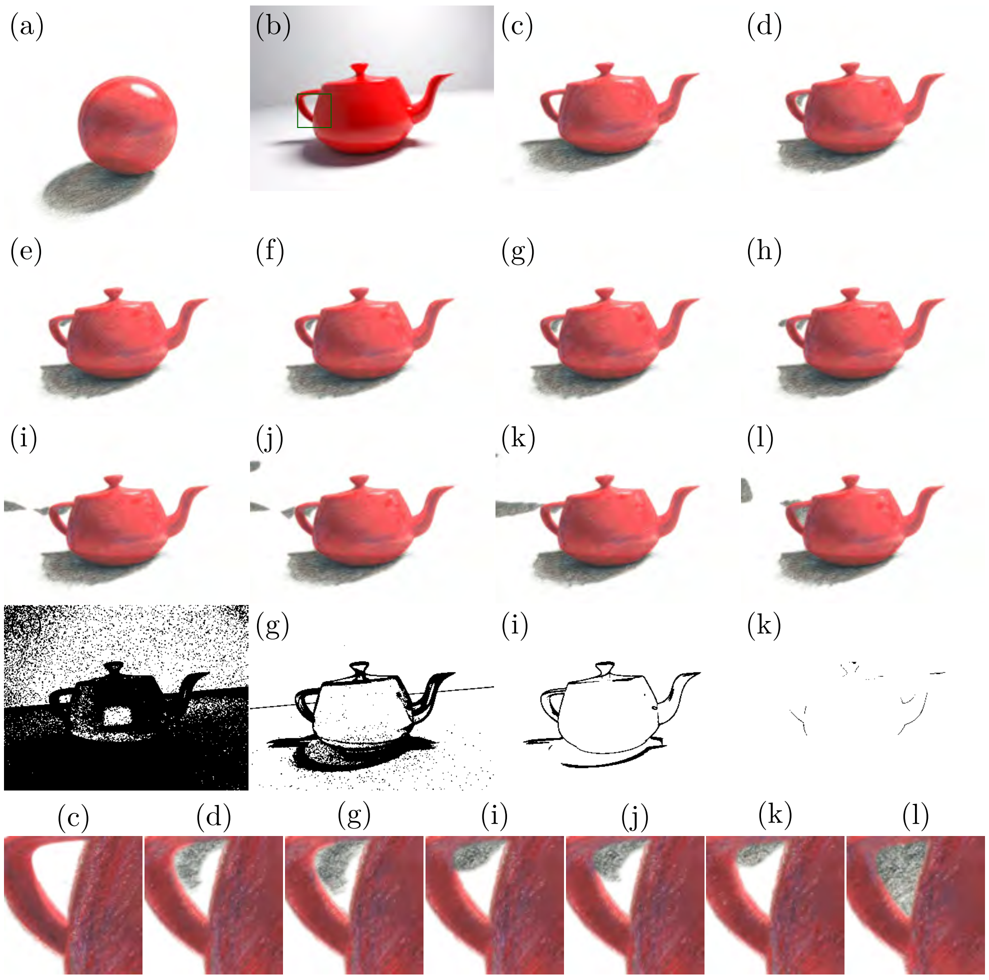

Figure 5.5: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) ... (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(x)}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(x)}}$
(c)	-	1714	-	-
(d)	-	113.362	-	15.120
(e)	3	69.126	1.640	24.795
(f)	5	49.991	2.268	34.286
(g)	10	35.126	3.227	48.796
(h)	25	32.379	3.501	52.936
(i)	100	31.277	3.624	54.801
(j)	500	30.795	3.681	55.658
(k)	5000	30.277	3.744	56.611
(l)	10000	30.041	3.774	57.055

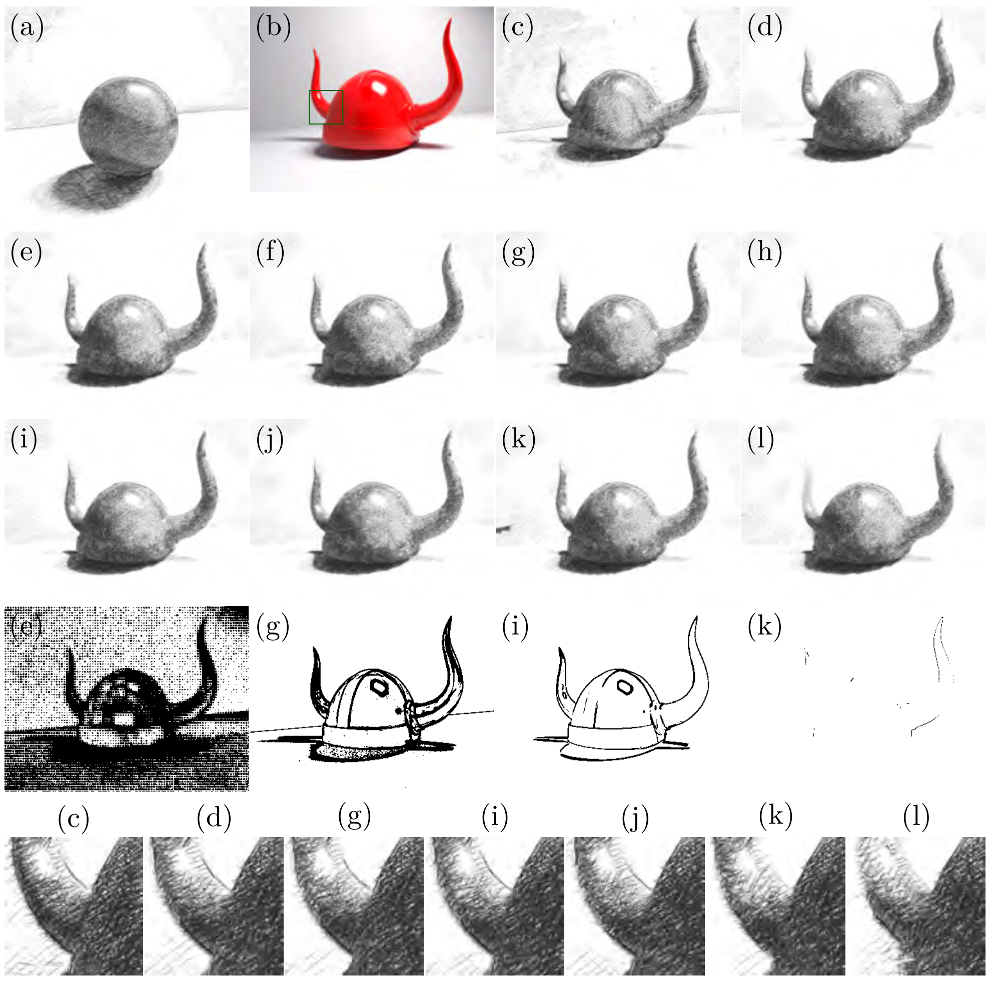

Figure 5.6: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) … (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(\text{x})}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2663	-	-
(d)	-	107.939	-	24.671
(e)	3	61.015	1.769	43.645
(f)	5	39.351	2.743	67.673
(g)	10	33.266	3.245	80.052
(h)	25	31.722	3.403	83.948
(i)	100	30.451	3.545	87.452
(j)	500	29.797	3.622	89.371
(k)	5000	29.010	3.721	91.796
(l)	10000	28.762	3.753	92.587

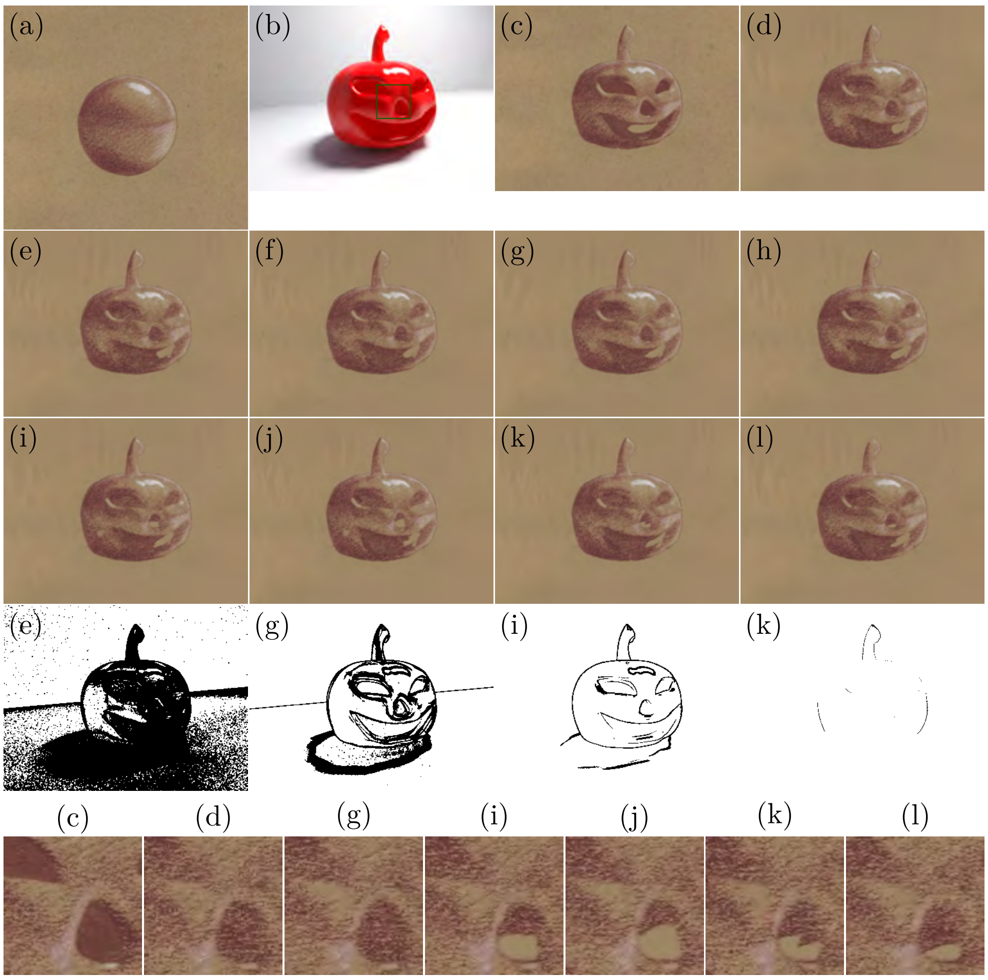

Figure 5.7: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) . . . (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(x)}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(x)}}$
(c)	-	1796	-	-
(d)	-	109.294	-	16.433
(e)	3	52.674	2.075	34.097
(f)	5	38.231	2.859	46.978
(g)	10	33.104	3.302	54.253
(h)	25	31.493	3.470	57.029
(i)	100	30.725	3.557	58.454
(j)	500	30.065	3.635	59.737
(k)	5000	29.512	3.703	60.857
(l)	10000	29.222	3.740	61.461

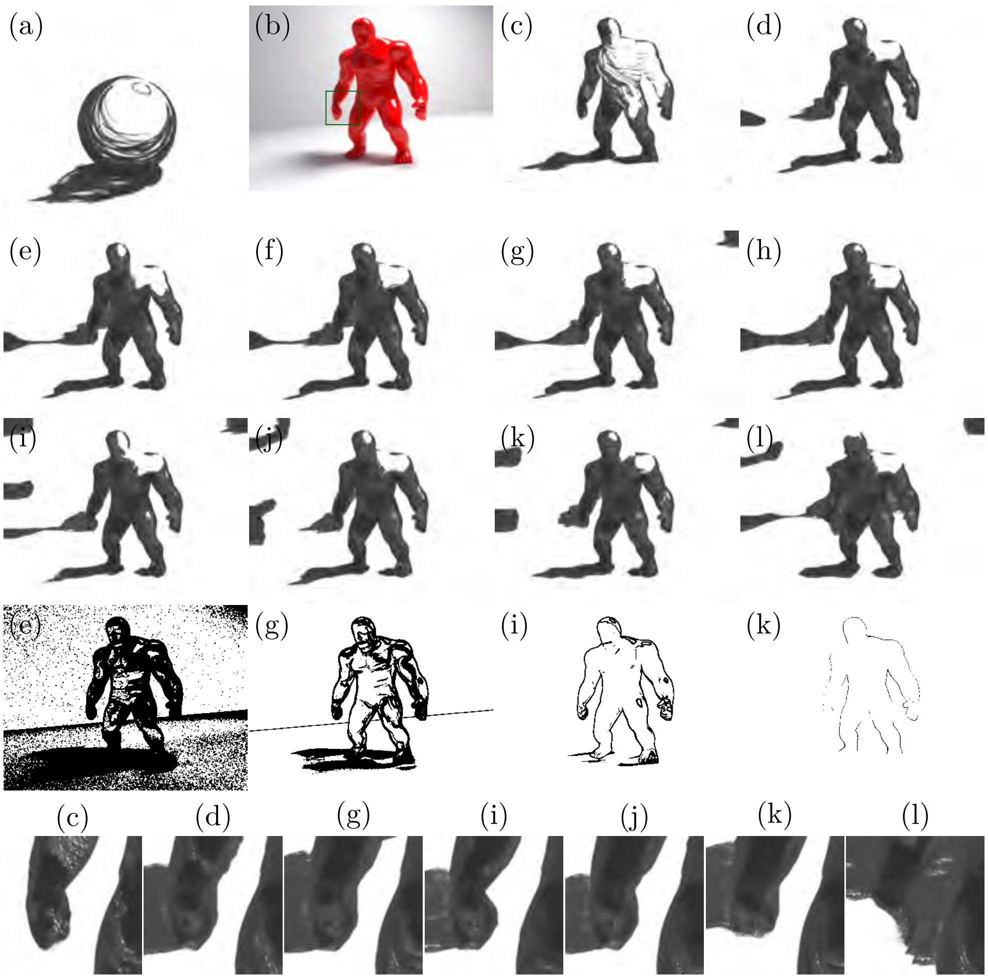

Figure 5.8: Comparison of our results with those of Ref and StyLit. (a) Stylized example texture A' © Daichi Ito. (b) Target guidance (only B_1 is displayed). The green square displays area used for the zoom-ins (bottom images). (c) StyLit [6]. (d) Ref with $\lambda = 10000$. (e) ... (l) Our with increasing ϵ.

	ϵ	time [s]	$\frac{\text{time}_{\text{Ref}}}{\text{time}_{(\text{x})}}$	$\frac{\text{time}_{\text{StyLit}}}{\text{time}_{(\text{x})}}$
(c)	-	2561	-	-
(d)	-	122.495	-	20.907
(e)	3	61.019	2.007	41.971
(f)	5	44.587	2.747	57.438
(g)	10	38.979	3.143	65.702
(h)	25	36.211	3.383	70.724
(i)	100	33.768	3.628	75.841
(j)	500	33.004	3.712	77.597
(k)	5000	32.619	3.755	78.513
(l)	10000	31.864	3.844	80.373

5.2.4 Discussion

The execution times of our improved approach have significantly decreased compared to Ref (up to 3.8×) thanks to the reduced memory load. The performance tends to improve the most for small ϵ changes in the beginning ($\epsilon < 100$) and then later stabilize with minor changes as the value grows further ($\epsilon > 100$). In most cases, the new results are identical to those of Ref where the (i) results with values around $\epsilon = 100$ tend to be in a "sweet spot" between performance and final quality. The guide maps with values of ϵ over 5000 start to lose important edge information. As a result, body of a stylized target might start to bloat (see the space between front limbs in Fig. 5.2), or the edges might fade out (see the left horn of the helmet in Fig. 5.6). In some cases, there are artifacts present in all results no matter what ϵ value we use (see the "spilled" shadow in the teapot and golem scenarios in Fig. 5.5 and 5.8). These special cases of "spilling" are mostly style-dependent and can be easily fixed by providing additional guidance textures B_6 and A_6 carrying ID information of the scene objects, i.e., target rendering containing only 3 distinct colors that strictly distinguish background, floor, and object from each other.

Unfortunately, the user-provided ϵ value suffers the same problem as in the case of λ in Ref. To achieve the best results, we would have to fine-tune the values in each scenario by comparing the outputs. Plus in this case, we would have to fine-tune both λ and ϵ at the same time which increases the complexity of the parameter setting.

Conclusion

In this work, we have first presented the problem of stylization by example and the motivation to solve such problem. Numerous related publications from the area were described, most important of all Fišer et al. (StyLit) [6].

Based on StyLit, we formulated a new problem of energy minimization in texture synthesis that uses extended energy term. The term includes pixel weights coming from the occurrence map Ω [11] that encourages uniform distribution of patch assignments while also making effective GPU implementation possible.

We then presented a method that solves the problem with texture optimization algorithm [12, 19] run in a pyramid scheme to synthesize the result and PatchMatch algorithm [3] to evaluate NNF. We also used additional guidance textures of StyLit to be able to preserve the example stylization details provided by the artist.

As a next step, the reference GPU implementation was presented along with its detailed kernels. The ability to provide effective GPU accelerated version of the synthesis process proved to be beneficial as the execution times were in orders of magnitude lower than those of StyLit. The quality of the implementation results, when compared to those of StyLit, decreased as expected. However, the quality was still within satisfactory and comparable range.

Finally, we conducted experiments that further improved performance of the reference GPU implementation, one of which brought exceptional results additionally lowering execution time multiple times and without any additional drawbacks on the quality in most cases. The final version of our GPU algorithm is able to provide real-time synthesis performance on today's high-end GPUs (i.e., GTX 1080) on high resolutions which successfully fulfilled our goal. As a result, we were able to provide the artists with much faster and more reliable feedback.

For future work, there is still space for additional further experiments that might decrease the rate of accessing global GPU memory during execution of PatchMatch [3] which is still the slowest part of the process. Algorithmic improvements that include approximations seem to be the key to achieving this as their correctly configured scenarios might be able to notably improve performance without harsh impact on the quality.

Bibliography

[1] *NVIDIA CUDA Toolkit Documentation.* http://docs.nvidia.com/cuda. Accessed: 24.11.2016.

[2] Ashikhmin, Michael. *Synthesizing natural textures.* In Proceedings of Symposium on Interactive 3D graphics, pp. 217–226. 2001.

[3] Barnes, Connelly, Shechtman, Eli, Finkelstein, Adam, and Goldman, Dan B. *PatchMatch: A randomized correspondence algorithm for structural image editing.* ACM Transactions on Graphics, 28(3):24, 2009.

[4] Barnes, Connelly, Zhang, Fang-Lue, Lou, Liming, Wu, Xian, and Hu, Shi-Min. *PatchTable: Efficient Patch Queries for Large Datasets and Applications.* ACM Transactions on Graphics, 34(4):97, 2015.

[5] Bénard, Pierre, Cole, Forrester, Kass, Michael, Mordatch, Igor, Hegarty, James, Senn, Martin Sebastian, Fleischer, Kurt, Pesare, Davide, and Breeden, Katherine. *Stylizing Animation By Example.* ACM Transactions on Graphics, 32(4):119, 2013.

[6] Fišer, Jakub, Jamriška, Ondřej, Lukáč, Michal, Shechtman, Eli, Asente, Paul, Lu, Jingwan, and Sýkora, Daniel. *StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings.* ACM Transactions on Graphics, 35(4), 2016.

[7] Fišer, Jakub, Lukáč, Michal, Jamriška, Ondřej, Čadík, Martin, Gingold, Yotam, Asente, Paul, and Sýkora, Daniel. *Color Me Noisy: Example-based Rendering of Hand-colored Animations with Temporal Noise Control.* Computer Graphics Forum, 33(4):1–10, 2014.

[8] Gatys, Leon A., Ecker, Alexander S., and Bethge, Matthias. *A Neural Algorithm of Artistic Style.* CoRR, abs/1508.06576, 2015.

[9] Hertzmann, Aaron, Jacobs, Charles E., Oliver, Nuria, Curless, Brian, and Salesin, David H. *Image Analogies.* In SIGGRAPH Conference Proceedings, pp. 327–340. 2001.

[10] Jamriška, Ondřej, Fišer, Jakub, Asente, Paul, Lu, Jingwan, Shechtman, Eli, and Sýkora, Daniel. *LazyFluids: Appearance Transfer for Fluid Animations.* ACM Transactions on Graphics, 34(4):92, 2015.

[11] Kaspar, A., Neubert, B., Lischinski, D., Pauly, M., and Kopf, J. *Self Tuning Texture Optimization.* Computer Graphics Forum, 34(2):349–360, 2015.

[12] Kwatra, Vivek, Essa, Irfan A., Bobick, Aaron F., and Kwatra, Nipun. *Texture optimization for example-based synthesis.* ACM Transactions on Graphics, 24(3):795–802, 2005.

[13] Newson, Alasdair, Almansa, Andrés, Fradet, Matthieu, Gousseau, Yann, and Pérez, Patrick. *Video Inpainting of Complex Scenes.* SIAM Journal of Imaging Science, 7(4):1993–2019, 2014.

[14] Portilla, Javier and Simoncelli, Eero P. *A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients.* International Journal of Computer Vision, 40(1):49–70, 2000. ISSN 1573-1405. doi: 10.1023/A:1026553619983.
URL `http://dx.doi.org/10.1023/A:1026553619983`

[15] Rong, Guodong and Tan, Tiow-Seng. *Jump flooding in GPU with applications to Voronoi diagram and distance transform.* In Proceedings of Symposium on Interactive 3D Graphics and Games, pp. 109–116. 2006.

[16] Simonyan, Karen and Zisserman, Andrew. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* CoRR, abs/1409.1556, 2014.

[17] Sloan, Peter-Pike J., Martin, William, Gooch, Amy, and Gooch, Bruce. *The Lit Sphere: A Model for Capturing NPR Shading from Art.* In Proceedings of Graphics Interface, pp. 143–150. 2001.

[18] Wei, Li-Yi and Levoy, Marc. *Fast Texture Synthesis Using Tree-structured Vector Quantization.* In Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, pp. 479–488. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 1-58113-208-5. doi:10.1145/344779.345009.
URL `http://dx.doi.org/10.1145/344779.345009`

[19] Wexler, Y., Shechtman, E., and Irani, M. *Space-Time Completion of Video.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 29(3):463–476, 2007.

Appendix **A**

Acronyms

RGB Red green blue color model

CPU Central processing unit

EM Expectation-maximization

GPU Graphics processing unit

LPE Light path expressions

NNF Nearest-neighbor field

OS Operating system

SIMT Single instruction multiple thread

SM Streaming multiprocessor

SSD Sum of squared differences

Appendix B

Contents of enclosed CD

```
  readme.txt ....................... the file with CD contents description
├── exe ................................... the directory with executables
├── src ....................................the directory of source codes
│   ├── wbdcm ................................... implementation sources
│   └── thesis .............the directory of LaTeX source codes of the thesis
└── text .......................................the thesis text directory
    ├── thesis.pdf .........................the thesis text in PDF format
    └── thesis.ps ...........................the thesis text in PS format
```