master’s thesis

Example-based Stylization of 3D Models Using Unreal Engine

Mykola Isaiev

August, 2022

supervisor: prof. Ing. Daniel Sýkora, Ph.D.

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

Acknowledgement

I would like to thank prof. Ing. Daniel Sýkora, Ph.D. for the project supervision. I would also like to express gratitude to CTU in Prague for providing great opportunities and much needed support for students around the world in these trying times.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

Abstrakt

Tato práce se věnuje stylizaci virtuálních scén na základě předloh pomocí algoritmu StyleBlit, který vytváří jedinečné nefotorealistické umělecké styly využitelné pro vykreslování virtuálních scén v reálném čase. Na základě analýzy algoritmu je navržena implementace pro Unreal Engine, jejíž výsledky jsou dále porovnány s výstupy jiných stylizačních algoritmu.

Klíčová slova

Stylizace obrazů základě předloh, Unreal Engine, stylizace v reálném čase, výtvarné zobrazování, StyleBlit

Abstract

This thesis is focused around the task of example-driven virtual scenes stylization, specifically on the StyleBlit algorithm, which enables the creation of unique non-photorealistic art styles that can be used for virtual scenes rendering in real time. The algorithm is studied and an implementation for Unreal Engine is proposed. The results of the implementation are evaluated against other stylization approaches.

Keywords

Example-based stylization, Unreal Engine, Real-time stylization, artistic rendering, StyleBlit

Contents

1 Introduction

Many computer games and other interactive virtual projects developed today tend to deviate from the classical photo-realistic portrayal of the virtual objects in the scene, and seek a unique distinctly non-realistic visual style. However, creating a stylized artistic rendering of interactive virtual scenes is a complex task that requires incorporating knowledge of vastly different fields to achieve a good result. Many popular approaches to the task revolve around the creation of a single particular look. The adoption of an existing stylization algorithm into a different project may often lead to the complete restructuring of the algorithm. For it, the developer may require the expertise of an artist. In this context example-driven stylization techniques present a potent alternative that helps to separate the software engineering tasks from the artistic process and provides more ways to adjust the resulting stylization on the fly.

A large number of approaches to automated stylizing of digital images and videos have been studied over the last three decades. The process of image stylization can be broadly defined as creation of a stylized image, which preserves the semantic information of the target image, while also containing recognizable features of the specified style.

This thesis explores an algorithm used to create non-realistic renders of virtual scenes based on style image examples. The general technique of image-based stylization is widely used. The main idea is to use an image or a video containing information about the desired style and apply the information to the target image in a meaningful manner. The style reference (or style exemplar) can often be an image created by an artist without any strict limitations on the medium, or retrieved in any other way. Some approaches can transfer style information from a preexisting image, others require a specific format of the exemplar, and thus the example has to be created specifically for the task. The way style information is derived from the exemplar and applied to the target varies between different algorithms.

The algorithms for the task of image-based stylistic rendering (also referred to as artistic rendering) are often categorized according to the methodology they use. An overview of some of the techniques can be found in [1]. A large family of algorithms bases the transfer process on neural networks. These algorithms are generally suitable for style transfer from a preexisting exemplar, as they apply sophisticated image analysis techniques to derive semantic information from the exemplar and use it as a guide for the stylization. Other techniques use the notion of strokes and a canvas. The canvas represents the resulting image and is covered with atomic graphical elements (i.e. strokes) that are selected and placed on the canvas based on the content of the target image and user input. An alternative approach to the task is to use low-level computer-vision techniques such as edge detection to derive the semantic information of the target that can later be used for transfer. Some of the existing algorithms combine several image processing approaches to achieve the desired result. Different methods are suited for different purposes and usage scenarios.

The algorithm explored in this project is StyleBlit [2]. This algorithm is specifically tailored for real-time artistic rendering of virtual scenes based on additional information about the scene geometry. It uses style exemplars created for the usage with the algorithm and can be used in a wide variety of applications, computer games being one of the primary examples.

1.1 Project Aims

Nowadays, a vast majority of computer games are developed using a preexisting game engine (e.g. Unity or Unreal Engine) that contains frameworks and libraries useful for game development. A game engine usually contains the runtime logic functionality that takes care of the game execution, and an editor, i.e. a user interface for creation of in-game content. Many popular game engines allow users to create their own shareable pieces of functionality that can be used by different users in different projects. In Unreal Engine such entities are called plugins. They present a large set of tools for developers to extend both the runtime and editor capabilities of the engine and can be shared between users to quickly add new features into any project.

This thesis is centered around the creation of a StyleBlit plugin for Unreal Engine 4. The plugin must contain an implementation of StyleBlit stylization algorithm capable of stylizing complex scenes. The StyleBlit plugin must also provide a way for the user to adjust parameters and apply their own style exemplars. The implementation is compared with the reference implementations as well as other stylization algorithms. The flexibility of the implementation is demonstrated with an example of a compact interactive scene.

2 Analysis

This chapter is dedicated to establishing the background of exemplar-driven stylization algorithms and analyzing the existing solutions of the task from the perspective of stylized rendering of virtual scenes. It further contains a detailed description of the StyleBlit algorithm as presented in the original article. The description is supplemented with the analysis of two existing implementations of StyleBlit. They are studied to see how the applied algorithm integrates into existing computer image synthesis pipelines. Next a brief overview of the Unreal Engine is provided and the structure of the implementation is proposed. Finally the chapter contains a description of other stylization techniques included into the implementation.

2.1 Background

A precise task formulation of example-driven stylization can prove helpful in designing stylization algorithms. However, any formulation of the task that attempts to cover all possible usage scenarios runs the risk of being either too broad or too abstract to be of any use. It is also important to narrow down the exact applications of the developed algorithm. This thesis is primarily focused on real-time example-based stylization of virtual scenes based on the available scene information.

For the sake of this analysis an artistic image can be viewed as a set of artistic decisions that aim to express certain aspects of the depicted scene. An artistic decision is a combination of visual expressive tools applied to an entity in the scene to convey semantic information about it. The corresponding scene information is called a guide channel. A stylization algorithm uses the guide channels of a scene to apply a predefined set of artistic decisions according to preset rules.

Generally speaking, there is no limit to what can be considered an artistic decision, e.g. composition of a scene on the image, exaggeration of geometric proportion, or the usage of a specific color schema, but the focus of this thesis implies the following limitations: the geometry of the scene must not be changed significantly as well as the layout of the elements on the screen. The limited scope of artistic tools can be used to convey information about scene geometry, lighting, and color.

A guide channel is a data structure containing semantic information about the depicted scene. The type of information stored in a guide channel can vary from purely semantic (e.g. distinguishing the entities in the scene or segmentation of different facial features as in [3]) to purely visual (e.g. the color of corresponding scene objects). While in certain cases it can be beneficial to use a dedicated data structure to ensure faster queries into the guide channel, the most popular approach is to store the guide information in an image, i.e. a matrix of pixels, where each element of the matrices is in a bijection with a exemplar pixel or a target pixel.

Guide channels for stylization can be obtained in a number of ways, a large set of algorithms make use of computer vision techniques to retrieve the semantic data directly from the image. These methods are most prominently used for stylization of photos, as the additional scene information is rarely available for them. The guide channel retrieval stage does not have to be executed explicitly as a separate stage. Many algorithms calculate the semantic information based on the input image and perform the style transfer in a single step. The semantic information retrieval methods applied for the task can vary in their complexity (from relatively simple, e.g. [4], to complex, e.g. [5]). Other algorithms rely on supplementary inputs.

An example-driven stylization algorithm is a stylization algorithm, in which the set of predefined artistic decisions is stored in form of an additional image or a set of images - an exemplar. The stylization can then be executed by means of finding the correspondences between the exemplar guide channels and the target ones and using them to guide the style transfer. A popular approach to such style transfer utilizes neural networks to retrieve the corresponding style information from the exemplar image and reconstruct the resulting image based on the exemplar style layer and the target content layer (as is described in an overview [6]). This technique allows the style transfer to be executed between arbitrary images and neither imposes restrictions to the exemplar format, nor requires supplementary information about both target and exemplar scenes. A major limitation of the technique is the lack of direct control over the transfer.

An alternative approach to exemplar-driven stylization is based on the knowledge of the guide channels of the target image and the exemplar. The guide channels are not retrieved from the input images using neural networks or other image processing techniques, but are rather supplied as the inputs to the algorithm alongside the exemplar itself. This family of algorithms has a significantly narrower range of use cases when compared with the neural style transfer described earlier. It requires the scene guide information, which is not available in a general case. It also requires a certain format of the exemplars, which have to be created specifically for the task, as opposed to the neural transfer that can be applied on preexisting images. These algorithms are most commonly applied to stylized rendering of virtual scenes as it enables the access to additional scene data. They also provide significantly more artistic control over the result.

The general structure of these algorithms tend to be similar. They can be roughly separated into two general stages. The first stage is the search for the best correspondences between the source and the target guide channels. The second stage is the style transfer itself, in which parts of the exemplar are transformed according to the computed correspondences and applied to the target image to get the final result. The algorithms may also have additional post-process steps that adjust the final result.

For such algorithms the task of style transfer can also be expressed as an optimization problem (initially described in [7], further formalized in [2]).

$$E(G_t, I_t, G_s, I_s, p, q, \mu) = ||I_t(p) - I_s(q)||^2 + \mu||G_t(p) - G_s(q)||^2, \tag{2.1.1}$$

$$\boldsymbol{E} = \sum_{p \in I_t} min_{q \in I_s} E(G_t, I_t, G_s, I_s, p, q, \mu) \tag{2.1.2}$$

Here $E(G_t, I_t, G_s, I_s, p, q, \mu)$ is the error of transfer for pixels p and q, G_t is the guide channel for the target image, I_t is the result image, G_s is the source guide channel, and I_s is the style exemplar. μ is the weight of the guide channel in the transfer. The first term forces the preservation of the exemplar's features, the second term ensures the correspondence of the guide channels. The overall sum of all pixel transfer errors $\boldsymbol{E}$ is minimized. The exact definitions of these terms vary significantly from algorithm

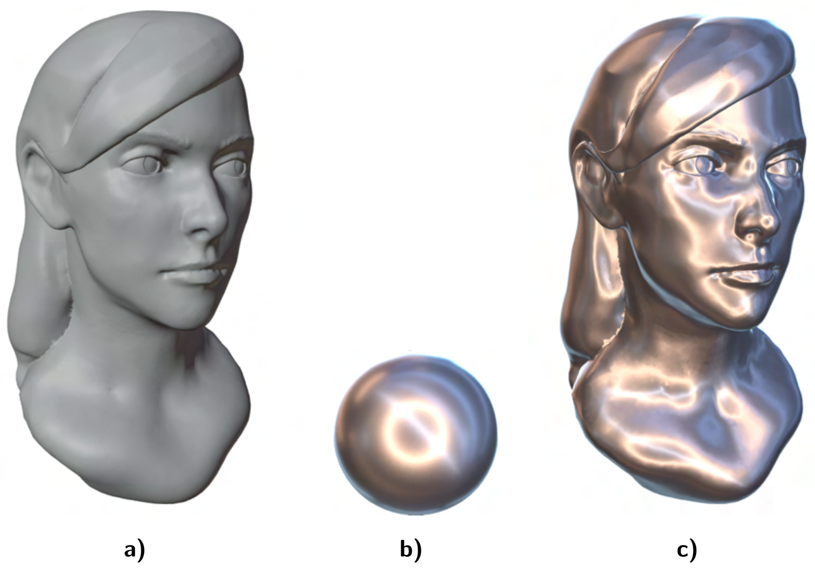


Figure 1 Sculpted model Portrait2 in Blender 2.81: *a* - using no MatCap (rendering via an empiric shading model), *c* - rendered using exemplar *b* (included with the program's distributable).

to algorithm and can express additional requirements to the final result such as local coherence of the transferred style information or the uniform usage of all parts of the exemplar.

Before applying this framework to the analysis of StyleBlit, it is demonstrated on two adjacent style transfer algorithms: the Lit Sphere and StyLit.

2.1.1 The Lit Sphere

The Lit Sphere approach to non-realistic rendering of virtual scenes was introduced in [8]. It is often referred to as MatCap (as for example in [9]). The algorithm has found significant popularity thanks to its simplicity and efficiency. It is often used in 3D modeling software as an alternative to the classical shading models as it can provide more visual information about the scene geometry to the user, while sacrificing the realism of the image. It can also be helpful for fast assessments of a scene's look where the precise shading is not necessary.

The main structure of the algorithm fits neatly into the exemplar-based style transfer framework laid out earlier. The algorithm uses view-space normals of the geometry in the virtual scene as the guide channel of the transfer. The exemplar is required to be a rendering of a sphere centered in the middle of the image. The style transfer optimization problem is stated in the simplest form:

$$E(G_t, I_t, G_s, I_s, p, q, \mu) = ||G_t(p) - G_s(q)||^2, \tag{2.1.3}$$

The transfer of the style is determined solely by the correspondence between the style guide and target guide channels. The preservation of the style features is not explicitly enforced, which ensures the simplicity of the optimization problem's solution. For each pixel of the target image an exemplar pixel with the same normal vector has to be found. This is further simplified by the exemplar's required format. A simple sphere contains every possible normal vector of a visible point of a surface and the positions

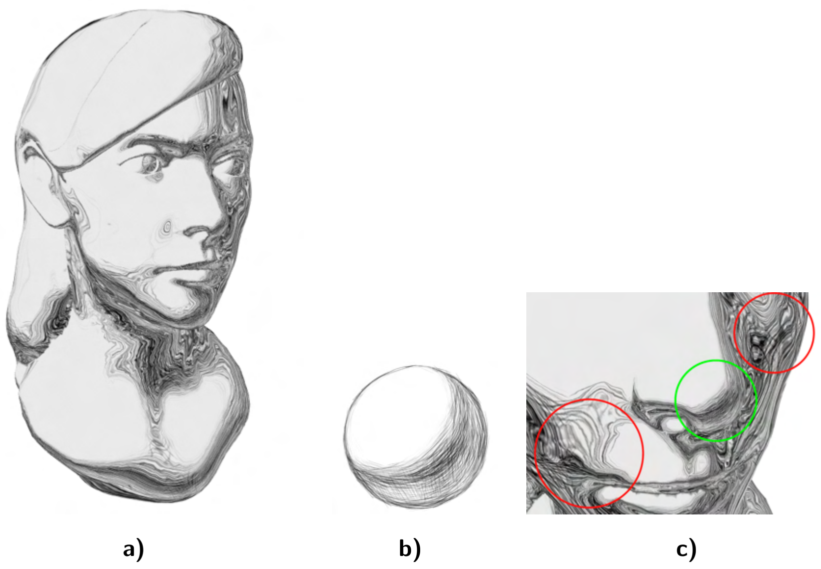


Figure 2 *a* - Model Female1 in Blender 2.81 using exemplar *b*, which contains high-frequency details, in a close-up *c* the area which approximates a sphere is rendered preserving some exemplar coherence (highlighted green), in other areas noticeable distortion is visible (highlighted red).

of the corresponding pixels can be retrieved analytically Thus the exemplar guide can be inferred and does not have to be supplied as an input.

A significant disadvantage of the algorithm is the lack of explicit exemplar coherency enforcement. The algorithm performs well on spherical objects, but any noticeable deviations in the geometry introduce significant distortions to the high-frequency style features. The algorithm is not suited for flat surfaces as those present large areas with no variation of the normal vectors and a single pixel of the exemplar is then used to cover the entire region of the target image. The algorithm does not suppose alternative exemplar formats or guide channels.

2.1.2 StyLit

The StyLit algorithm was introduced in [10]. It presents an alternative approach to exemplar-based style transfer capable of producing high quality results on complex scenes. StyLit requires significantly more computations for the style transfer than MatCap, but is able to produce visually interesting results on a variety of scenes. The idea of the algorithm is based on the notion that an artist may use distinctly different ways to portray different lighting scenarios of the scene regardless of the factual image luminance.

The algorithm's input is a simple reference scene aligned with a prepared style exemplar (i.e. artistic interpretation of the reference scene). Rather than relying solely on view-space normal vectors, StyleLit uses information about scene lighting as the main guide channel. The required lighting information about the scene is produced using a common light propagation approach, and recording different light-path data into the corresponding channels. The different light path channels then serve as a single complex guide channel for the style transfer. The selection of this guide channel makeup aims to guide the transfer based on the nature of the perceived lighting scenario in every part of the target image, thus producing guide channels with more discriminative power (i.e. with less ambiguity).

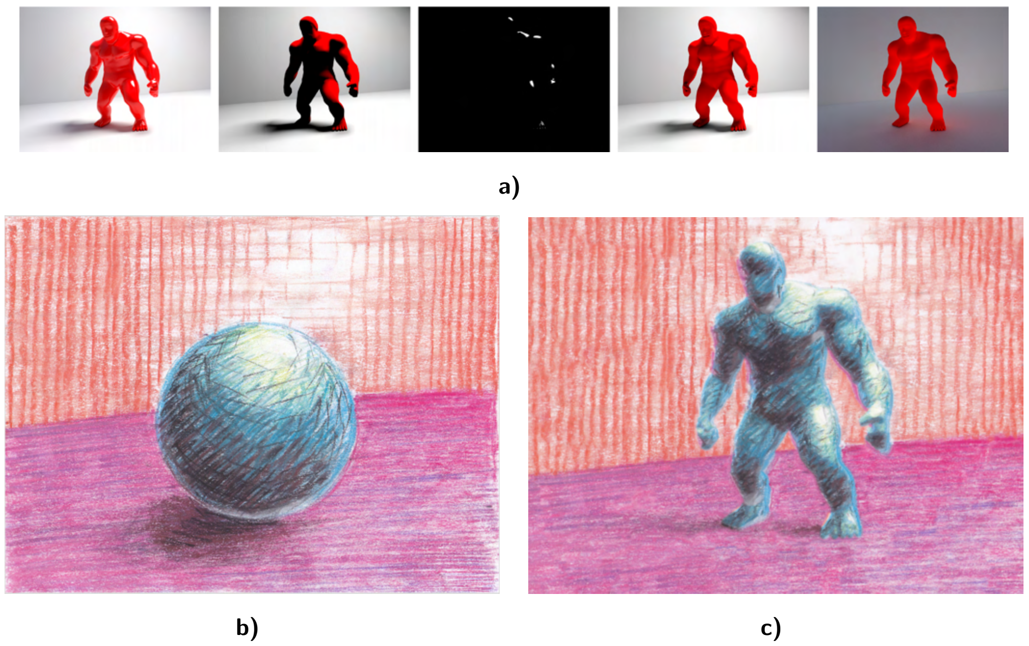


Figure 3 StyleLit algorithm out put for Golem model: *a* - individual light path channels that make up the target guide channel (taken from the supplementary materials of the original article), *b* - the artistic rendering of the exemplar scene, *c* - output of the algorithm (generated by the demo implementation provided with the article, which does not include edge detection guide layer, thus the finer details of Golem's geometry like lingers and facial features are lost).

The first step of the algorithm is to generate the guide channels for the exemplar scene. This information may be stored and reused for different targets. The second step is to compute the guide channel for the target scene. This is done in the same way as for the exemplar. Next, the algorithm executes the style transfer. The basic approach is for every pixel in the target correspondence to find the nearest neighbor in the reference correspondence. The basic approach will be able to correctly transfer color of the exemplar to the target based on the lighting scenario in each pixel, however a modified iterative approach must be applied in order for the algorithm to preserve recognisable features of the style (e.g. individual brush strokes), such that the exemplar pixels are transferred in patches and the patches themselves are used uniformly.

The proposed solution reverses the direction of the classical nearest neighbor search, which allows for uniformity of patch usage enforcement (a similar technique is used e.g. in [11]). While the intuitive approach to the problem would be to find the best matching region of the exemplar for every region of the target image, StyLit performs the search differently. In broad strokes, the algorithm iteratively finds best matches for pixel regions of the exemplar among those of the target image. The region is assigned, if the error of the region assignment is below a certain threshold, if not, the region is evaluated in a subsequent iteration. In a subsequent iteration a smaller region size is selected and the process repeats. The iterations continue until the entire target image is covered. Unlike in the nearest neighbor field search described in [11], the uniformity of the exemplar region usage cannot be enforced directly, as it would lead to inaccurate results, when the proportion of the image areas covered by different light path groups in the exemplar differs significantly from that of the target. In practice the implementation has to strike a balance between the region usage uniformity and the overall assignment error.

Due to its structure the algorithm is unable to correctly perform style transfer in cases when the target scene contains lighting scenarios not present in the exemplar. While this makes intuitive sense, it can also be a significant disadvantage for stylization of dynamic scenes with varying lighting conditions. It may be challenging to create an exemplar that covers a sufficient amount of lighting scenarios for the perceptually seamless stylization and there is no option of providing multiple different exemplars. The algorithm's nearest neighbor field search approach also leads to some of the finer geometry details being lost in the final image. This is due to the averaging of the evaluated patches guide information, which acts as a low-pass filter on the input image. The original article suggests a solution for this problem by introducing an additional edge layer to the transfer guide channel.

The computational complexity of the algorithm is significantly higher than that of the Lit Sphere technique. The main contributors to the StyLit computation time are the light propagation and multi-dimensional nearest neighbor search that are executed for the transfer. The results of light propagation can only be stored for a static exemplar. In case of an interactive virtual target scene the algorithm would require to perform the computations anew every frame. While thanks the recent advances in GPU architectures it is becoming closer to a realistic goal to execute such calculations in real time, the large computational complexity can still be a major limiting factor (as described in [12]). On top of the light propagation, the iterative nearest neighbor field search has to be executed each frame. The nearest neighbor search is performed in the guide channel space. The suggested guide channel structure consists of five RGB images corresponding to different light path filters. This makes the total dimensionality of the search space 15, which, applied iteratively every frame, introduces a significant computational overhead. Together these two factors limit StyLit's usage for real time stylization scenarios.

2.2 StyleBlit Recapitulation

This thesis is primarily focused on a stylization algorithm called StyleBlit (introduced and described in [2]). It is a follow up technique that draws some of its inspiration from StyLit and aims to generate visually interesting style transfer results with less computational costs. The algorithm follows the same general framework and structure of StyLit and MatCap and attempts to combine the advantages of both of them while alleviating their disadvantages.

Just like in Matcap, the exemplar used in StyleBlit is an artistic rendering of a simple sphere with no information about the surrounding scene. In theory the algorithm can be extended to use an arbitrary exemplar scene with little computational overhead thanks to the simplicity of the guide channel structure. However, the sphere exemplar scene is close to the optimal choice for the task as it is difficult to cover the domain of the possible guide values with any other arbitrary scene.

Similarly to The Lit Sphere, StyleBlit relies on view-space normal vectors of the rendered surfaces as its primary guide channel. A general normal vector consists of 3 real numbers (x, y, z), but with the introduction of 2 implicit constraints the dimensionality of the guide channel is effectively reduced to just 2 dimensions. A normal vector is expected to be normalized, i.e. to lay on the unit sphere. It is further assumed that any rendered surface is oriented towards the camera, thus all normals lay in the hemisphere that is visible from the camera view point. Every point on the surface of the visible hemisphere is in a bijection with a point on the flat circle parallel to the view plane. This means that the z-coordinate of a normal vector can be trivially reconstructed from

x and y coordinates.

The original article also applies the technique to portrait stylization utilizing a different guide channel structure. The proposed solution explores the usage of displacement fields as the guide channel. A displacement field is a representation of a transformation that closely matches the pixels of one image onto another. The suggested approach retrieves the transformation fields applying an optimization method onto a matched set of points on both images. The points represent facial features and the transformation is performed using Moving Least Squares (described in [13]). This selection of the guide channel does not increase the dimensionality of the problem and widens the potential range of StyleBlit applications. However, the implementation proposed in this thesis focuses on the general virtual scene stylization and does not contain this approach.

In general the algorithm consists of three stages: target guide channel generation, the initial style transfer, and the blending post-process stage. Similarly to the algorithms reviewed earlier, the guide channel for the exemplar scene may be precomputed and stored for later usage in the form of a texture. In case of normal guidance, the target guide generation is executed by means of the traditional rasterization pipeline based on the geometry data that usually comes as a part of the polygonal model. The remaining algorithm's steps are examined more closely in the following sections.

2.2.1 Initial Style Transfer

The majority of the work is executed during the initial transfer stage. In it the correspondences between the target and exemplar guide channels are found. Unlike MatCap, StyleBlit does not use the result of pixel-wise nearest neighbor search for the transfer. Instead a variant of the coarse nearest neighbor field look-up is used to ensure a certain amount of style texture coherence in the final image. In order to execute the search efficiently, StyleBlit sacrifices some of the global constraints that are used in StyLit and does not ensure uniformity of the patch usage. The transfer is also executed in chunks, which are coherent exemplar image areas of arbitrary shape as opposed to the uniformly sized square patches used in StyLit.

The core of the algorithm revolves around the nearest neighbor search in the guide channels space. While the search algorithm used in StyleBlit is similar in its iterative nature to that of StyLit, it is executed in the traditional direction (i.e. for every chunk in the target guide the best matching chunk in the exemplar is found). The algorithm is designed to run on a highly parallel architecture of a GPU which dictates certain limitations. Launched on individual pixels it must form coherent structures spanning regions of multiple pixels. This is achieved by iteratively applying a rule to each of the pixels that assigns a pixel to a certain landmark point. Together with other pixels assigned to the same landmark it forms a coherent chunk.

The suggested rule for chunk assignment is to use a threshold to limit the maximum distance of the assigned exemplar pixel's guide to the target pixel guide. For a given pixel P_i, the algorithm looks at a limited number of pixels in the neighborhood $n(P_i)$ with a certain radius r_k. Among the selected pixels from $n(P_i)$ the closest pixel P_{ni} is selected and the threshold rule is applied. Namely, the algorithm computes the vector v from P_i to P_{ni}, the exemplar nearest neighbor E_{ni} for P_{ni} is retrieved, and a candidate matching pixel E_i is constructed as $E_{ni} - v$. The exemplar guide channel's value $G_e(E_i)$ for E_i is then compared to the target guide channel value $G_t(P_i)$ for the evaluated pixel P_i. If the distance between $G_e(E_i)$ and $G_t(P_i)$ is sufficiently small, the coordinates of E_i are used for the style transfer, if not, the same procedure is run again with a smaller neighborhood radius r_{k+1}. The algorithm's steps of the initial transfer is listed in Algorithm 1.

Algorithm 1: StyleBlit algorithm (no chunk border blending)

```
Input  : Target image $T$, target guide map $G_t$, style exemplar image $S$, style
         guide map $G_s$, number of levels $L$, threshold constant $t$
Output: The stylized image $R$
Function FindNearestSeed(pixel $P_i$, level step $r_k$):
    $d^\star = \infty$
    for $x_{step} \in (-1, 0, 1)$ do
        for $y_{step} \in (-1, 0, 1)$ do
            $P_{ni} = \lfloor P_i / r_k \rfloor + (x_{step}, y_{step})$
            $j = \text{RandomJitter}(P_{ni})$
            $P_{ni} = P_{ni} + j$
            $d = |P_i - P_{ni}|$
            if $d < d^\star$ then
                $d^\star = d$
                $P^\star_{ni} = P_{ni}$
            end
        end
    end
    return $P^\star_{ni}$

Function CalculateError(target pixel $p$, candidate pixel $p_c$):
    return $|G_t[p] - G_s[p_c]|$

for each pixel $p$ of the target image $T$ do
    for $L_i \in (L, ..., 1)$ do
        $P_{ni}$ = FindNearestSeed($a$, $b$)
        $E_{ni} = argmin_U(|G_t[P_{ni}] - G_s[U]|)$
        $v = P_i - P_{ni}$
        $E_i = E_{ni} - v$
        $e$ = CalculateError($E_i, P_i$)
        if $e < t$ then
            $R[P_i] = S[E_i]$
            break
        end
    end
end
```

The so-called landmark pixels in the neighborhood $n(P_i)$ must be selected unanimously by all evaluated pixels in the neighborhood. For it the algorithm uses a uniform step, i. e. in a given iteration k it selects only the pixels P_{kj} with texture coordinates $T(P_{kj})$, for which $T(P_{kj}) mod r_k = 0$. This guarantees the unanimous selection of pixels, but it also results in a noticeable regularity of the transferred chunks. To alleviate this issue a random offset for every landmark pixel is introduced breaking up the uniform grid pattern. The random offset must also be applied in a predictable way, so that it stays the same when evaluated by different pixels.

This coarse search method allows the overall error to be controlled by an input threshold parameter. It serves as a termination criterion of the iterative search and

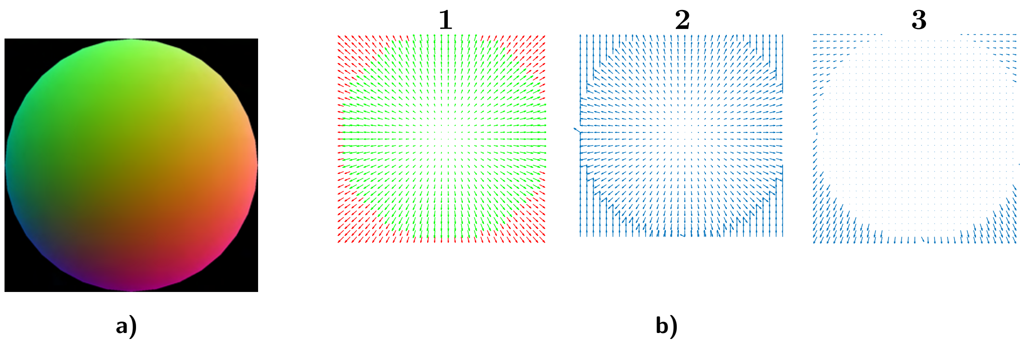


Figure 4 A visualization of the NNS analytical solution for the simple sphere exemplar scene: a - view-space normal vectors of a sphere rendered as RGB color, $b.1$ - pixels of a square texture mapped onto the linear space $< -1, 1 >^2$ using the transformation 2.2.1, each arrow depicts the orientation and norm of the corresponding coordinate vector, vectors with length more than 1, and thus do not represent feasible normal vectors, are marked red, $b.2$ - colors of picture a interpreted as vectors, $b.3$ - difference vectors between $b.1$ and $b.2$ demonstrate perfect matches. The illustrations b are undersampled for visibility.

thus controls the maximum deepening. Visually it corresponds to the average size of the chunks, as the sooner the search terminates, the further apart are the selected landmark pixels, and larger the areas that are covered by the corresponding chunks. Simultaneously it represents how closely the stylized object's geometry is followed. Additionally, the algorithm provides the user with control over the jitter by exposing the random offset function as an input. It can be used to simulate a hand-painted look's imperfections and temporal jittering.

The search uses a nearest neighbor look-up as a black box and does not explicitly require any specific exemplar format. However, a brute-force solution to the problem is extremely inefficient and does not allow for the algorithm to run in realtime. A more sophisticated approach must be used. A possible option is to use the analytical solution as in The Lit Sphere algorithm. It requires the specific exemplar scene format, namely, a sphere centered in the middle of the image and occupying all the available image space. This makes it easy to retrieve the position of exemplar pixels that have any given normal vector. The coordinates x and y of a normal vector are trivially mapped onto the coordinates of a corresponding pixel of the exemplar guide channel by the following transformation:

$$p_e(x, y) = (0.5(x + 1), 0.5(y + 1)), x, y \in < -1, 1 > \tag{2.2.1}$$

Another popular approach to realtime nearest neighbor search and other potentially time-intensive operations is the usage of a look-up table (LUT). A look-up table is a data structure that stores precomputed results of the required operation for the entire domain of the expected operation inputs. Instead of computing the result every time, a stored result is retrieved based on the input parameters. In case of the normal vector nearest neighbor search the lookup table can be stored as a simple two-dimensional texture. A general lookup table texture can be constructed in the following manner: the domain of the input parameters is mapped onto the coordinate space of a texture (for the case of view-space normal vectors the inverse of transformation 2.2.1 can be used). The mapped domain is sampled in discrete points that correspond to the texture pixels, i.e. for each texture pixel a corresponding target function result is computed

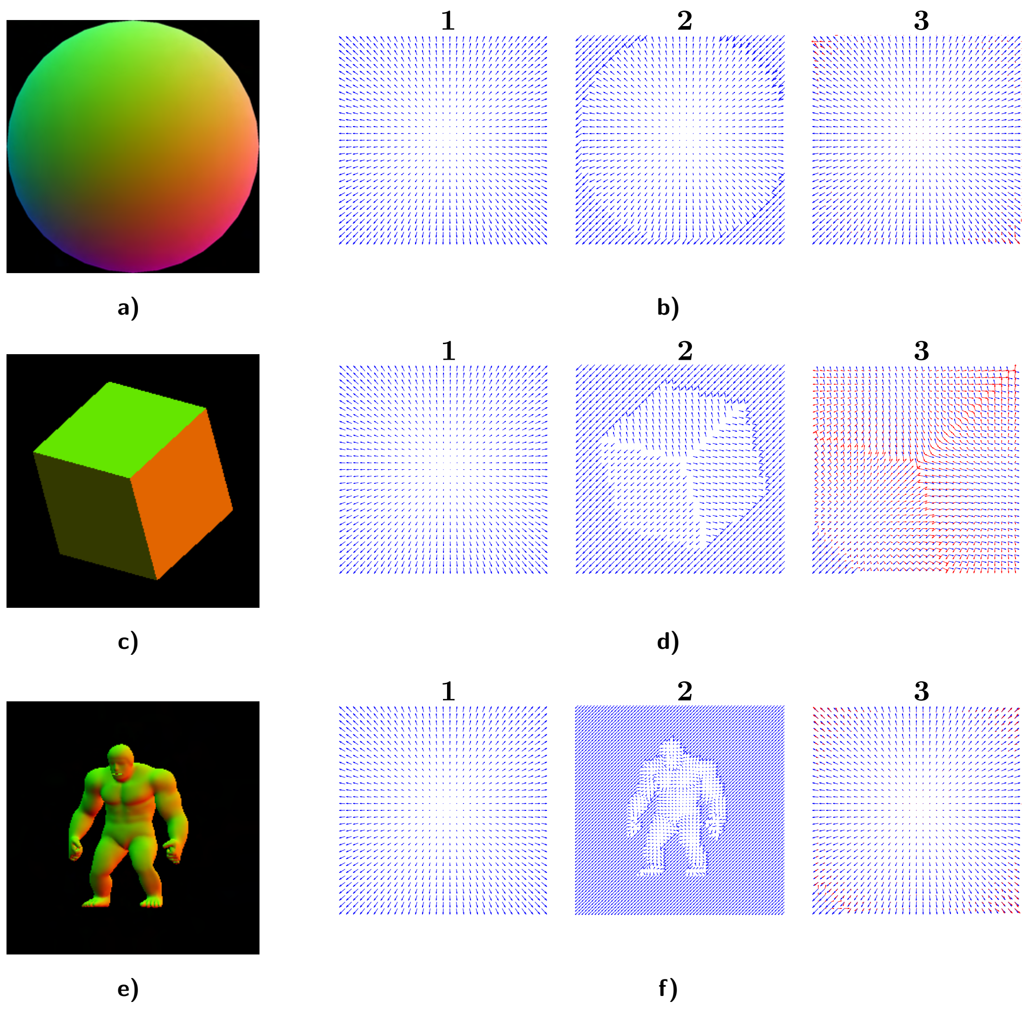


Figure 5 Visualization of LUTs for different exemplar scenes: a, c, e - exemplar guide channels, $b.1, d.1, f.1$ - linear space the LUT is computed for, $b.2, d.2, f.2$ - visualization of the guide channels, $b.3, d.3, f.3$ - visualization of the generated LUTs (blue arrows represent the nearest neighbor for every sample point, red arrows represents the error of the assignment).

using the respective parameter values. The results of the computation for each pixel are then stored in the texture object. The LUT texture can be used during look-up for a set of parameters by transforming the parameters into texture coordinates and reading the closest texture pixel storing the precomputed result.

In the case of StyleBlit, the coordinates of the pixels in the LUT texture correspond to x and y coordinates of a normal vector. The precomputed results can be stored in the pixels directly or an additional compression can be applied to account for the low precision of the texture's individual components. The selection of the LUT solution to the nearest neighbor search enables the algorithm to operate on an arbitrary exemplar scene, but, as discussed earlier, it may be challenging to create an arbitrary scene that produces visually interesting results after the transfer.

Different exemplar scenes result in different nearest neighbor LUTs. An important characteristic for the LUT usability discussion is the initial assignment error. The LUT solution of the nearest neighbor query does not guarantee that the results retrieved from the LUT match perfectly. Unlike with the simple sphere scene and the appropriate analytical solution, an arbitrary scene may not contain some of the possible normal

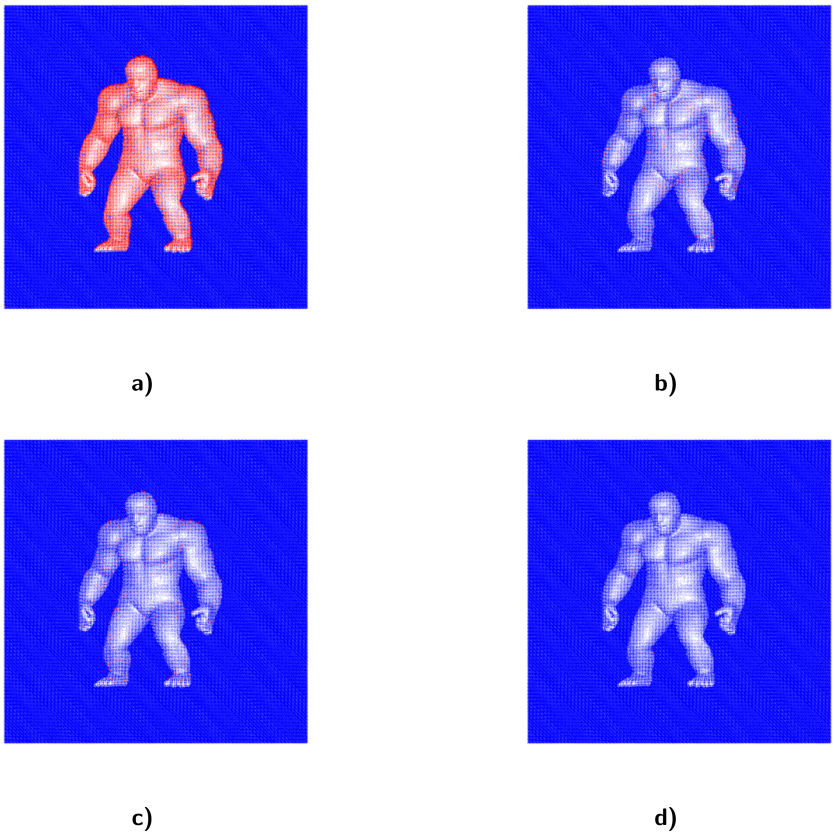


Figure 6 The visualization of the Golem scene guide channel and the ambiguous nearest neighbor search results during LUT construction: *a* - red vectors denote the normals of pixels that are used 1 or more times in the generated LUT, *b* - red vectors are used exactly 1 time, *c* - red vectors are used 8 times, *d* - red vectors are used 20 times. The blue vectors are not used in the LUT The usage of vectors is scattered in a seemingly random manner, the scatter depends on the implementation of nearest neighbor search used.

vector configurations and thus the nearest neighbor for the corresponding target pixels is assigned with an error. If the assignment error is too large, the algorithm may not function as expected due to the failure of the coarse search termination criterion. Additionally, an arbitrary exemplar scene may contain some of the normal vectors multiple times. This may result in the target pixels with identical or close guide channel values being assigned drastically different nearest neighbors due to the exemplar scene ambiguity. In turn this interferes with the way exemplar chunks are formed during the transfer. These metrics indicate that the best exemplar scene candidate must uniformly cover the guide channel value domain.

Figure 5 contains visualizations of some of the possible exemplar scenarios. The LUT built for a sphere exemplar scene outputs the LUT solution that matches the analytical solution for all possible vectors. The cube scene generates a LUT with significant initial assignment errors. This means that the algorithm is unable to find coherent chunks to transfer the style and will always select the nearest neighbor for every evaluated pixel. The implementation of the nearest neighbor search selects the same pixel for all matches which results in its abuse over the large areas of the image. Increasing the error threshold does not solve the problem as the error of the assignment is comparable with other pixels and the majority of the information about the scene geometry is lost.

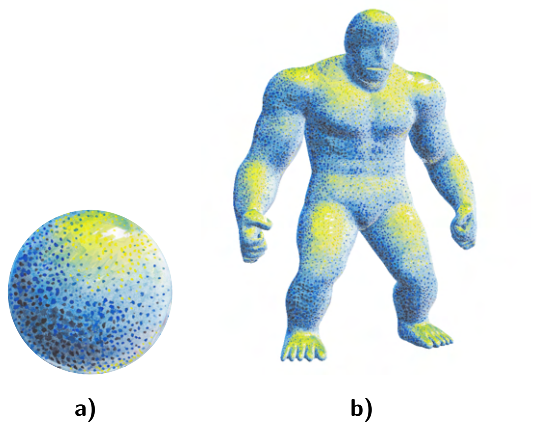


Figure 7 StyleBlit demonstration: *a* - style exemplar (taken from the original article), *b* - output of the initial transfer stage on the Golem model (generated using the implementation supplied with the original article).

The Golem exemplar scene generates a much more uniform coverage of the guide channel domain, however it suffers from significant ambiguity of the assignment(as demonstrated in figure ()). The ambiguous assignment of nearest neighbors results in pixels that belong to a continuous surface retrieving vastly different information about the exemplar guide channel and failing to form coherent chunks. The transfer based on the results of nearest neighbor search alone does not perform well either, because due to the assignment ambiguity target pixels select seemingly random exemplar pixels.

The selected exemplar pixel coordinates can be used for the transfer directly. This produces correct results, moreover, the original StyleBlit article makes an argument that the incoherencies at the chunk borders are not visually noticeable for a wide range of exemplar textures. The exemplars that the approach is best suited for contain high-frequency details with no apparent regular structure, as the basic algorithm does not handle the alignment of exemplars chunks during transfer. An additional post-process stage can be applied for chunk border blending, further improving the visual quality of the style transfer result.

2.2.2 Chunk Border Blending

The second stage of the algorithm does not perform any additional style transfer but only aims to improve the quality of the result obtained in the previous stage. It attempts to make the result appear more visually coherent while preserving as much style information as possible. It is done using a specific variant of adaptive blending technique that utilizes the knowledge of the chunks borders.

In order to apply the adaptive blending pass, the StyleBLit algorithm formulated earlier has to be slightly restructured. In its current form the algorithm executes one pass over the target image looking for the best matching pixel in exemplar for each target pixel. The proposed border blending step makes use of the selected exemplar pixels' coordinates as opposed to the final color, so the algorithm must be adjusted accordingly, so that it outputs exemplar pixel coordinates instead of color (i.e. for every pixel P_i of the initial transfer output $R(P_i)$ store the assigned exemplar coordinates).

Instead of executing blending by averaging the color of the image based on some kernel, the blending stage guides its averaging by the texture coordinates of the selected exemplar pixels in a pixel's neighborhood. The original paper refers to the approach

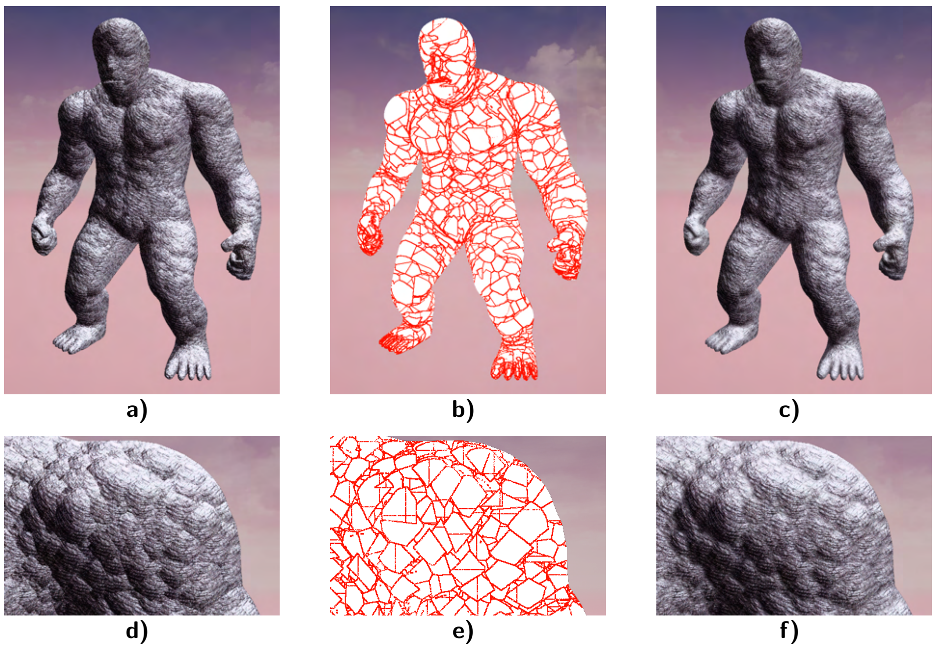


Figure 8 Patch border blending stage applied to the Golem model: a, d - results of the initial style transfer with no additional blending, b, e - red lines mark patch borders, white color represents the interior of the patches that must not be blended, c, f - result of the blending stage with $r = 3$.

as a voting mechanism: for a given pixel P_i its neighborhood $n(P_i)$ with radius r is examined. For each pixel $P_j \in n(P_i)$ a vector $v = P_i - P_j$. The exemplar color $C_j = E(R(P_j) + v)$ constitutes "the vote" of pixel P_j. The "votes" of all pixels in the neighborhood are averaged to get the final color for pixel P_i. When all the pixels in the neighborhood fall inside a coherent chunk, the result of the "voting" coincides with the output of the original initial style transfer stage. When a part of the neighborhood falls into a different chunk, the pixels vote for different output colors which are then averaged together producing a blending effect that occurs only at the borders of the chunks. The method's individual steps are listed in Algorithm 2.

This approach to the chunk blending has several advantages over more general adaptive blur approaches. A more universal image-processing approach (e.g. adaptive bilateral filtering as described in [14]) may be more computationally heavy while not performing as well as the described approach. It is difficult for a blending algorithm with no additional knowledge about the chunk structure to distinguish between the edges of a chunk that must be smoothed, and a feature of the exemplar that should be preserved.

2.2.3 Algorithm's Strengths and Weaknesses

StyleBlit algorithm generates visually interesting style transfer results for a large variety of virtual scenes and exemplars. The original article states that the approach is best suited for transferring the exemplars with dominant stochastic details (i.e. random high-frequency details), which benefit from the preservation of local style coherence while the low-frequency information that does not get transferred as well by the algorithm do not play as important of a role to the style. It is also proposed to use a combination of StyleBlit with other transfer algorithms for different detail frequency bands, namely

Algorithm 2: StyleBlit algorithm (with patch border blending)

```
Input  : Target image $T$, target guide map $G_T$, style exemplar image $S$, style
         guide map $G_S$, number of levels $L$, threshold constant $t$, blending
         radius $r$
Output: The stylized image $R$
1  Let $R_t$ be image with the size of $R$
2  for each pixel $p$ of the target image $T$ do
3      for $L_i \in (L, ..., 1)$ do
4          $P_{ni}$ = FindNearestSeed($a$, $b$)
5          $E_{ni} = argmin_U(|G_t[P_{ni}] - G_s[U]|)$
6          $v = P_i - P_{ni}$
7          $E_i = E_{ni} - v$
8          $e$ = CalculateError($E_i, P_i$)
9          if $e < t$ then
10             $R_t[P_i] = S[E_i]$
11             break
12         end
13     end
14 end
   /* Combine and blur the content of the image                         */
15 for each pixel $P_i$ of $T$ do
16     $w = 0$
17     for $x \in [-r, r] \cap \mathbb{Z}$ do
18         for $y \in [-r, r] \cap \mathbb{Z}$ do
19             $P_j = R_t[P_i - (x, y)]$
20             $w = w + 1$
21             $R[p] = R[p] + S[p_t + (x, y)]$
22         end
23     end
24     $R[p] = R[p]/w$
25 end
```

the Lit Sphere approach can be used to transfer low-frequency information. The result of the transfer can then be combined with the output of StyleBlit applied on the high-details layer of the exemplar.

The biggest advantage of the algorithm over StyLit is the computational speed. Thanks to the simplicity of the guide channel and the incorporation of the fast coarse nearest neighbor look-up technique the algorithm is able to perform in real time on a variety of platforms. The original article's supplementary materials include the demonstrational implementation on using OpenGL and WebGL, which is able to run on modern mobile devices with no noticeable lag. While it is difficult to uniformly assess the performance of the algorithm in a more realistic usage scenario such as a real video game, as it would involve a large quantity of implementation details that could influence the results, the light-weight demonstrational implementation showcases a significant potential of the algorithm to be used in an existing project.

The original article notes that StyleBlit performs worse than StyLit for exemplars with noticeable regular patterns. Additional alignment step is required to improve the chunk coherence that can help increase the perceived regularity of the transferred

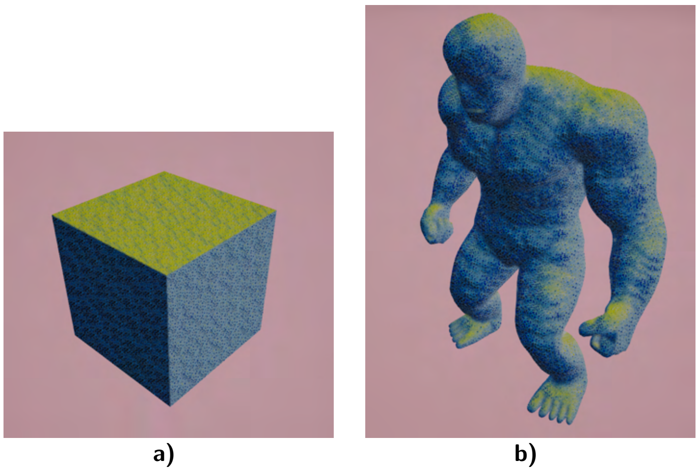


Figure 9 Limitations of StyleBlit described in the original article: *a* - noticeable pattern repetition on flat surfaces, *b* - finer geometry details lost to blending (especially noticeable the Golem's face and hands).

exemplar (e.g. as proposed in [15]). The algorithm also is not suited for scenarios, when large parts of the target image are occupied by pixels with nearly constant guide channel values. These regions occur when rendering objects with large flat parts such as buildings.

Another issue of the current algorithm occurs due to the structure of the chunk border blending procedure. The "voting" mechanism of the edge blending is able to preserve the style details, however, the geometry discontinuities such as contours and finer details cause the votes of the neighbor pixels to differ and the regions are blended along with the chunk borders. This results in the loss of details in the final image.

2.3 Reference Implementations Analysis

The original StyleBlit article is complemented with two implementations of the algorithm suited for different goals. The first implementation is a concise demonstrational application with minimalistic user interface and interactive elements. The second comes in the form of a plugin for Unity and contains a reusable implementation of StyleBlit along with supplementary tools for custom style development. In this section the implementations are studied in the context of the future implementation of the technique into Unreal Engine. There are several broad aspects of the implementations that have relevance to the subject of the project:

1. The algorithm separation into encapsulated logical entities - how the computations relevant to StyleBlit are distributed across the code and when they are executed.
2. The interactive components - the ways the user can interact with the implementation and use it to create custom visuals.
3. The algorithm integration into the surrounding systems - the way the implementation integrates into existing frameworks and interacts with other application components.

The aspects were selected based on the issues that arose during the algorithm implementation in the context of Unreal Engine. Inspecting the existing implementation for these issues is meant to help in resolving them.

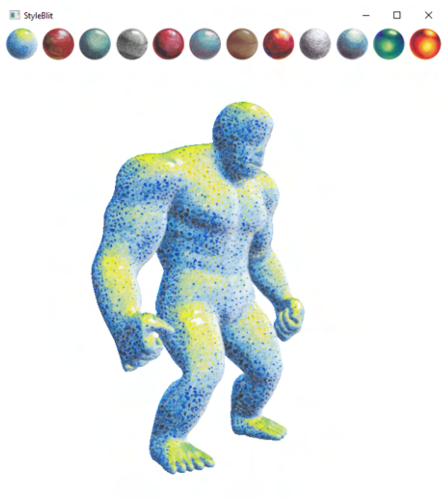


Figure 10 StyleBlit demo application screenshot.

2.3.1 Pure OpenGL implementation

The first implementation to be analyzed is the C++/OpenGL implementation published along with the original article. The purpose of the application is to showcase the algorithm on a variety of exemplars. This implementation does not tackle the majority of the questions posed in the beginning of the section and is mostly explored for inspiration about the code structure of StyleBlit itself.

It is a small application with several keyboard controls and mouse input support. The main window is a 3D viewport with a number of buttons that serve as a means to select the current style exemplar. The exemplars are showcased on a simple scene with a single model of Golem. The user can interact with the application by rotating and moving the view around the model and by changing the algorithm parameters via the respective keyboard keys. The application does not allow to load a custom model or use a different exemplar except from those which are built in. It is first and foremost a demonstration of the algorithm.

The source code of the demo is relatively compact. Beside the standard libraries for working with OpenGL and reading certain types of files, the entire code is contained in three C++ files and several shader files. These C++ files handle application interaction logic and the rendering itself. The interaction logic is fairly standard and is not relevant for the discussion. The rendering code separates the render into 3 distinct passes in accordance with the algorithm itself. In StyleBlit terminology the first pass creates the target guide map (i.e. the screen space normal vectors for every pixel of the screen). The second pass performs chunk transfer, assigning each pixel the corresponding coordinates of the nearest neighbor on the source. The third pass smooths the edges of the chunks and assigns the final color to each pixel.

The division of the rendering process is implemented similarly to a general deferred rendering technique (e.g. as described in [16]) that is widely used for a large number of tasks in computer graphics. As in a deferred rendering set up, the implementation makes use of the results of the previous steps in every next step. A classical deferred renderer executes two rasterization passes for every frame. First, the information about the visible points of the scene is gathered into the so-called G-buffer. The second pass uses the gathered information to calculate the final color of the pixel. The G-buffer typically includes several information channels (e.g. normals, diffuse color, specularity, etc),

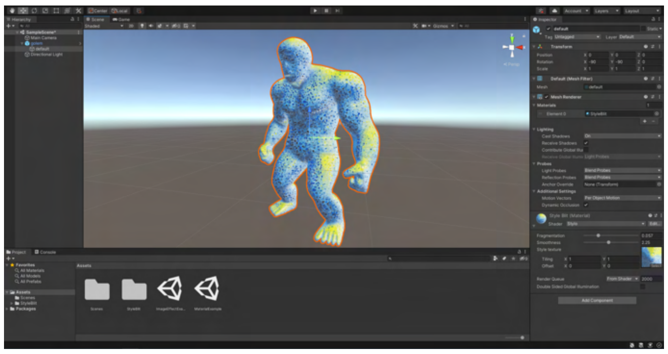


Figure 11 StyleBlit plugin for Unity screenshot.

which with combination of the general scene information like the position of the light sources allow to evaluate the final color of each pixel independently from every other. There are extensions and rendering techniques based on the deferred approach that add further computation passes based on the result of lighting evaluation. The initial transfer stage of the StyleBlit algorithm fits neatly into the basic deferred rendering pipeline, however this implementation adds a blending stage which is executed in the form of an additional pass.

The implementation relies on the analytical solution for the nearest neighbor retrieval. The exemplars supplied as a part of the demo application thus have to be renderings of a sphere position strictly in the middle of the image. The algorithm's formulation also uses a random jitter to prevent the chunks from forming noticeable regular patterns. The demo handles this in the form of a texture look up into a texture containing pseudo-random numbers. The random texture is generated for every frame on the CPU side and is sent to the GPU as a usual texture object.

2.3.2 Unity implementation

The second implementation analyzed in the project comes in the form of a Unity plugin. It is a collection of C# scripts and shader-based materials that execute the style transfer computations. Unlike the previous example the asset is designed to work with a more flexible set up and allow the user to upload their own exemplars. The implementation also supports arbitrary reference scenes to derive source guide maps from.

Unity has a flexible programmable rendering pipeline system which allows the user to switch between forward and deferred rendering paradigms (see the official engine documentation for more information [17]). The implementation of StyleBlit for Unity is based on the custom multi-pass shader materials. The ShaderLab framework allows the shader programs written in a version of HLSL to have multiple sub-shaders for individual passes. Each proceeding sub-shader can have access to the output of the previous pass. This allows the user to freely create among other things materials and visual effects based on the deferred rendering approach, and thus it is suitable for the StyleBlit algorithm as well.

The functionality of the plugin can be separated into two categories: the editor side of the computation and the run-time side. In order to better describe the categories, some of the implementation details have to be clarified first. As it has been already mentioned the plugin allows the user to use custom exemplars and reference scenes. This means that, compared to the previous concise implementation, the Unity adaptation has to sacrifice the simplicity of the nearest neighbor queries, as the reference scene may have arbitrary geometric shape. The nearest neighbor queries are handled using a look-up table, which is generated based on a specific reference scene beforehand. The guide map of a reference scene can also be generated once and is usually used as a basis for the artistic interpretation in order to provide sufficient level of alignment between the two.

This informs the internal structure of the plugin: a part of the implementation is dedicated to reference guide map generation and look-up table precomputation, while the rest handles the algorithm computation in the game. There are several utility scripts included in the plugin that handle the reference scene guide map generation and a script dedicated to computing the look-up table. The table does not have to be recomputed, if the reference scene was not changed, so it makes sense to store it as a texture asset as well. One of the scripts is responsible for noise texture generation which is executed on CPU and passed to the material as a texture. The run-time computations are executed via the shader containing the algorithm's implementation. It is connected to a material with some interface for users to change the StyleBlit input parameters. The material can be applied to arbitrary objects in the scene and can be used alongside other materials and assets.

The Unity implementation of StyleBlit proposes two approaches that allow the application of different style exemplar textures. The first utilizes the Unity shader architecture: the ShaderLab shader passes are executed per object and do not affect the surrounding objects.This allows the algorithm to be used with multiple exemplars without any explicit extensions. However, custom per object passes are computationally costly.

The second approach executes the post-process stages over the entire image. The exemplars are stored in a texture array data structure that allows texture lookups based on three dimensional coordinates. Each object the stylization has to be applied to has a number of the desired style exemplar applied to it. During the rendering an additional pass is executed per object. In this pass the corresponding exemplar numbers are written into the frame buffer for every pixel occupied by a stylized object. In the post process passes the style index buffer is read and the indexes are used during the texture array lookup to retrieve different style information for different objects in the scene. The approach imposes certain limitations on the style transfer, namely the parameters of the transfer are applied uniformly and cannot be adjusted per object or per style exemplar. The texture array data structure does not support differing scale of individual textures included in it. This means that all the exemplars used in a scene have to have the same texture dimensions.

Unlike the demonstrational OpenGL implementation, the plugin uses the look-up table solution for fast nearest neighbor queries. While the simple sphere is close to the optimal exemplar scene in terms of the guide channel domain coverage, the analytical solution restricts it to being centered. Using the LUT approach allows the Unity implementation to utilize the exemplars initially created for StyLit demonstration and are not properly aligned.

Even though the look-up table construction must be executed only once per reference guide map, the naive approach to nearest neighbor search (i.e. checking every sample point against every pixel of the target guide map) can still take an unreasonable amount of time for relatively small guide maps. The StyleBlit Unity plugin uses a kd-tree (a data structure described in [18]) to improve the efficiency of the look-up table construction. The implementation relies on an independent library to handle the data structure construction and query evaluation. The LUT generation functionality is separated into a dedicated in-editor menu, where the user has the option to generate and save LUT based on an arbitrary input image. The user is trusted to generate the appropriate guide channel image on their own.

In the following chapter the described framework is applied to StyleBlit and Unreal Engine, and the structure of the final implementation is proposed.

3 Implemetation

This chapter of the thesis contains detailed discussion of the proposed StyleBlit implementation for Unreal Engine 4. The possibilities for StylieBlit integration into the existing Unreal framework are analyzed.

3.1 Unreal Engine Rendering Pipeline

The architecture of the implementation is dictated in a large part by the structure of Unreal Engine's rendering pipeline. Unreal Engine follows the general deferred rendering paradigm, however, its rendering system includes a large number of extensions and optimization techniques which enable Unreal to perform well on a wide range of scenes. It also presents many options for interesting visual effects and simulates a variety of lighting phenomena. As the engine undergoes constant rapid development, the list of rendering features changes and the order of operations performed in the pipeline of one version of Unreal may differ significantly from that of another. This thesis is focused on a specific version of the engine, namely, the version 4.27, and the following analysis may not hold for future releases of the software (e.g. Unreal Engine 5 adds conceptually new stages to the renderer such as Nanite virtualized geometry technology described in the official documentation [19]).

The rendering pipeline used in Unreal has to accommodate a large spectrum of goals from particle systems management to lens flares and other post-processing effects. The engine handles those tasks utilizing a rendering thread. It is a dedicated thread of the engine that has to execute all communications between the CPU and the GPU. The rendering thread runs in parallel to the main engine thread and is not accessed directly by the user scripts. Instead, the engine issues render thread commands that are subsequently processed by the rendering thread. Beyond queuing render commands directly the engine allows generate render dependency graphs which consist of individual commands and provides advanced features for automated video memory resource management (as stated in the official documentation [20]).

It is difficult to come across a comprehensive overview of the exact stages of the rendering pipeline. The users have used external graphics debugging software (e.g. in the blog posts [21]) to dissect a frame rendered by Unreal Engine and outlined the general steps. In such analysis of the pipeline it is important to run the engine on an appropriate scene so that every type of the supported graphics operations is triggered. The analysis' results show the following executed by the renderer:

1. Particle simulation - the initial pass performs early computations of particles positions and velocities to be used later in the render.

2. Z-Prepass - early scene depth evaluation for the rendered geometry. It serves two major roles in the pipeline. The pixels that are occluded by other pixels are discarded earlier without performing costly fragment shader computations. The result of the pass is also used in a later stage of the pipeline for lighting computations.

3. Occlusion queries - rough conservative visibility tests that help the CPU decide which objects in the scene must be queued for render. They are evaluated based on the results of the early depth pass. The evaluation results are communicated from GPU to CPU. Due to the synchronization latency the evaluation results from a previous frame may be used.
4. Hi-Z pass - the construction of a hierarchical data structure for fast scene depth look ups that can be used for computation of complex lighting phenomena (e.g. as described in [22]).
5. Shadow map rendering - generation of shadow atlases for different types of lights using the hardware accelerated depth test (the general technique is well known, a basic version described in [23]).
6. Light assignment - compute shader dispatch for frustum intersection queries with geometric representation of light source volumes. The result of the stage is used to reduce the amount of lighting computation performed for a region of the screen by only accounting for the relevant light sources (a popular example of such technique is described in [24]).
7. Volumetric fog - computations for atmospheric lighting effect based on the evaluation of the lighting conditions in a three dimensional grid in view space (the general approach described in [25]).
8. G-Prepass - writing the geometry information such as surface normals and diffuse color into the dedicated render buffers to be used in later lighting evaluation stages.
9. Particle simulation - second pass for the particle effects adjusts the results of the early simulation based on the information about the scene geometry that was generated after the initial particle system pass.
10. Velocity rendering - render of moveable objects into the dedicated frame buffer used to simulate motion blur.
11. Ambient occlusion - occlusion of the geometry cavities is approximated with the help of the hierarchical z-buffer generated earlier. The pass also used the results of the ambient occlusion calculations from the previous frames to smooth the potential bias of the approximation results.
12. Lighting - a combination pass that uses the information gathered in G-buffer along with the shadow maps to produce the majority of the lighting effects.
13. Image space lighting - a stage that takes care of the screen space lighting effects, namely screen space reflections. The glossy reflections are simulated based on the data available on the frame buffers which enables fast look-up, but limits the possibilities of the algorithm (described in [26]).
14. Fog and atmospheric effects - computations of atmospheric effects that require additional care and could not be handled in the volumetric fog stage. Atmospheric scattering (as described in [27]) and light shafts simulations (described in [28]) are calculated here.
15. Transparency rendering - simulation of the transparency effects based on the image space gathering technique (similar to screen space reflections mentioned earlier). The stage uses the data gathered in the previous steps to compute an accurate result.

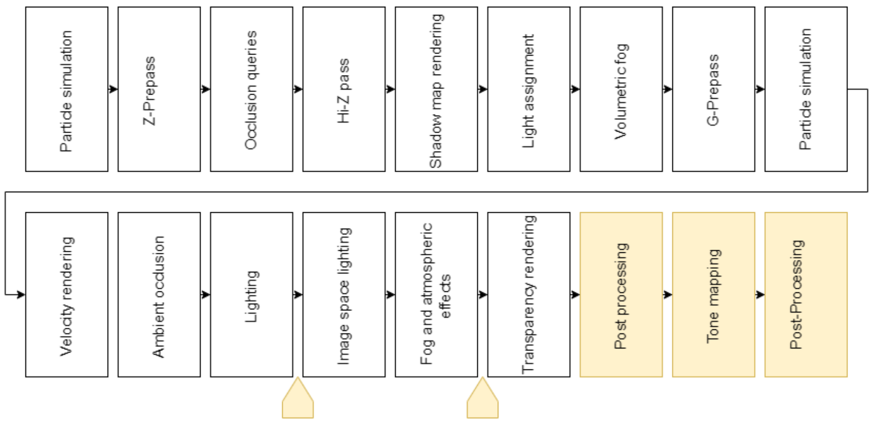


Figure 12 Unreal Engine rendering pipeline visualized (yellow denotes potentially programmable parts of the pipeline).

16. Post processing - the stage executes calculations required for automatic exposure adjustment, applies bloom effect and motion blur using the velocity frame buffer. The steps executed in this stage depend on the camera settings.

17. Tone Mapping - combination of the post process stage results and the lighting calculation results, exposure adjustments, and color grading. The result of the preceding stages is compressed into an 8-bit precision color image to be displayed.

It is worth noting that the provided outline is not exhaustive and may vary depending on the content of the rendered scene. The rendering pipeline of Unreal Engine is designed to efficiently simulate many lighting phenomena and is specifically tailored to produce results visually close to photo-realistic. A significant restructuring of the existing scene object types is usually required to use an alternative rendering set-up. It can be done through user-defined "Vertex Factories". It refers to Unreal's abstraction for vertex attribute communication to the graphics card. In essence, a vertex factory contains a description of the vertex data format in which the geometry is prepared by the CPU and supplied to the GPU. It is also connected to other object types that handle synchronization of data between the main processor and the graphics card. The user can define additional shading models and perform custom geometry operations defining the respective geometry management objects, however, selectively disabling the extensive rendering pipeline for some objects would require modifications to the engine's source code and thus would not be compatible with any other engine builds.

Thankfully, the current pipeline allows to include additional programmable stages to tweak the render result and allows the user to build custom visuals based on the output of several stages of the rendering pipeline. The programmable parts of the pipeline are referred to as post process stages. They are executed in the later part of the pipeline when the majority of the data about the scene has been computed. Unreal allows the users to program custom effects by means of the extensive material editor toolkit, hence the post process stages are called post process materials. They can also be viewed as emissive materials applied to the screen quad (i.e. a simple plane placed directly in front of the camera so that the entire viewport is covered). A set of enabled post process materials is tied either to the active camera entity, or one of the post process components (i.e. objects with special for post process material assignment).

While the number of slots available for custom post process materials in the rendering pipeline is limited - namely, before the screen space reflections, before translucency,

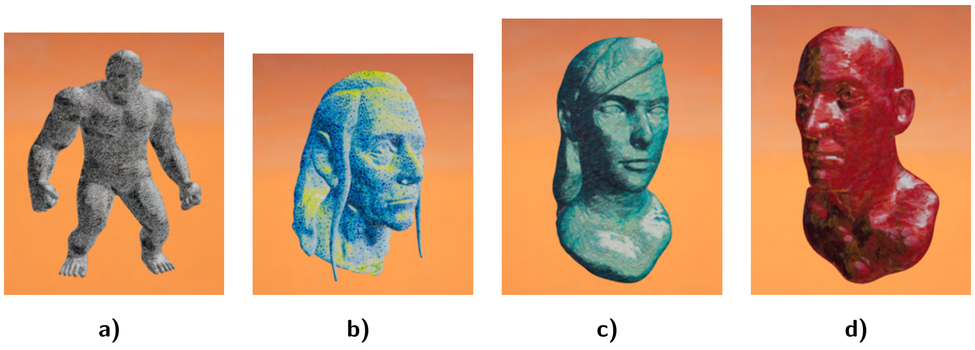


Figure 13 StyleBlit in Unreal Engine 4.27.

before tone mapping, replacing the tone mapper, and after tone mapping - each slot can hold multiple post process materials. The result of each post process material stage is available to the next consequent stage along with the entire G-buffer (render buffers containing information gathered in the previous stages of the render, e.g. normal vectors and scene depth).

Alternatively Unreal rendering pipeline can be extended by means of render targets. A render target is the interface Unreal provides for working with custom frame buffers that are not directly involved in the rendering pipeline. Render targets are often used in combination with scene capture entities which are able to launch the full render pipeline for a specified render target. Render targets can be used for a large number of tasks. They can be useful for post process materials, as a render target with a post process material applied during its render effectively makes the result of that post process stage accessible from every post process stage of the main rendering pipeline. However, launching a separate render pipeline comes at a significant computational cost.

3.1.1 StyleBlit Integration

StyleBlit requires two consequent passes over the image with access to the information about the visible scene geometry and the ability to communicate the result of the first pass to the second. This can be implemented using post process materials assigned to the same post process slot and executed consequently. The alternative implementation approach using render targets can be applied as well, but in the general case the overhead introduced by the render target makes it the less optimal approach.

The original article and the demonstrational OpenGL implementation do not tackle the possibility of using multiple style exemplars for different objects in the scene. The Unity implementation contains two alternatives suited for different needs: per object shader passes, and full screen shader passes. The StyleBlit implementation proposed in this thesis takes an approach similar to the full screen shader from Unity implementation.

The style exemplars are assigned to individual mesh objects in the scene by their respective indices in a master texture array. The indices are supplied into the rendering pipeline, where a post process material executes style transfer from a texture array data structure using the objects' style indices. Instead of writing the indices in an additional pass, an existing mechanism is used: custom stencil buffer. In Unreal Engine mesh objects have the option of being rendered into an additional render buffer, which has a restricted format. The buffer is called Custom Scene Depth and is able to contain

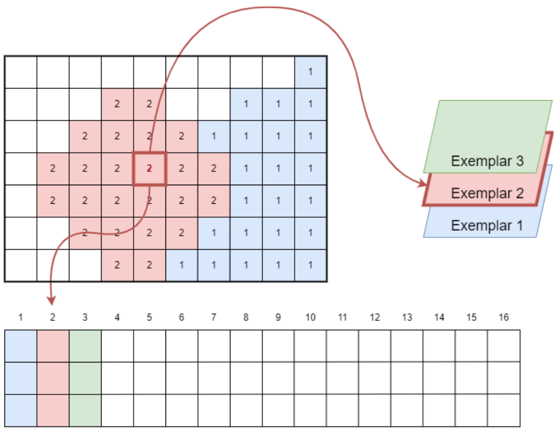


Figure 14 A schematic depiction of StyleBlit set up for to work with multiple exemplars: the image stencil buffer values are used to index into a specific exemplar from the exemplar array. The same values are used to read a column of pixels in the parameters texture.

a single unsigned 8-bit integer number per pixel. The buffer can be accessed from a post process material like any other scene texture. It is most commonly used as a mask for stencil operations, where a post process material has to be applied to a part of the screen. The implementation uses the custom scene depth buffer to store the exemplar indices for individual objects. This limits the total number of exemplars that can be used simultaneously in a scene to 256, which is assessed to be enough for the vast majority of the use cases.

The implementation proposed in this thesis contains both the post process materials only version, and the render target version. The latter could be used to either upsample or downsample the scene during the initial transfer stage. It can also be used to explicitly change the buffer format without having to modify the project settings. The implementation also extends the multi-exemplar approach by allowing different sets of StyleBlit parameters to be used for different objects. The parameter sets are stored in an additional texture in the form of pixels in such a way that they could be retrieved easily based on the style exemplar index. Additionally the limitation on the exemplar texture dimensions can be alleviated by applying multiple consequent chunk border blending post process materials with different exemplar texture objects supplied as input. The blending materials must only override the pixel content of the frame buffer with the corresponding custom scene buffer values, which requires a slight restructuring of the existing solution.

An important detail of the StyleBlit integration into the Unreal renderer is the selection of the post process materials slot. The two most potent options are Before Tonemapping and After Tonemapping. Other options imply additional processing applied to the results of the stylization which may lead to an unwanted outcome. The Replacing the Tonemapper option effectively disables the tone mapping functionality for the entire screen as the corresponding built-in shader is not executed when the slot is assigned to a user-defined post process material. Before and After Tonemapping options influence whether the tone mapping is applied onto the stylization results. For the general case of the basic StyleBlit it makes more sense to apply stylization after the tone mapper, because it will guarantee that the color scheme of the stylized object matches that of the exemplar. However, the frame buffer (i.e. the render target corresponding to the rendered image) available for reading from and writing into changes its format during the tone mapping step. Unreal calculates scene lighting in a higher definition range storing the intermediate results in 16-bit precision floats per render buffer channel. The tone mapper output is a compressed texture with just 8-bit precision float channels.

Additionally, the frame buffer's alpha channel is not handled in the same way the other channels are between the render pipeline stages. The opacity channel's value written during a post-processing stage into the buffer guides the blending of the stage's result with the previous stages. The existing StyleBlit implementations for OpenGL and Unity dealt with the loss of detail between the initial transfer and blending stages by packing the two-component UV coordinates generated in the initial stage into 4 8-bit precision channels and unpacking the information in the latter stage. Due to the lack of control over the resulting frame buffer alpha values the Unreal implementation would have to pack the UVs into a three-component float vector in order to use the After Tonemapping post-process slot. Alternatively, the Before Tonemapping allows the usage of the uncompressed 16-bit frame buffer to communicate between consequent post processing materials, albeit with slight color changes to the exemplar texture, and relieves the necessity to pack the retrieved coordinates. It also allows writing additional information into the unused component of the output pixels during the algorithm's initial stage.

The Unreal implementation of StyleBLit includes both LUT and analytical solutions of the nearest neighbor search used in the algorithm. The LUT can be generated with the dedicated tool supplied with the implementation. The corresponding guide channel image can also be generated and saved there. The following section contains a description of the final implementation structure.

3.2 Plugin Structure

Any Unreal Engine project consists of one or more modules. These are code encapsulation units that provide a number of features to help organize the internal code structure of the project's source code. Modules serve as default paths for code file inclusion. They can be conditionally included in or excluded from a project's build version based on the defined build type. This is useful when creating features that are meant to be used in editor and thus should not be a part of the game build. The features can also rely on strictly in-editor logic that will make it impossible to compile the project without the editor. For those cases the features can be placed inside an editor module that is not a part of the game build. Alternatively, a module may contain some platform-specific code that can only be included in builds for a given platform and not for others.

A module may consist of the associated source code files as well as other modules. A module also has a set of module dependencies that are required for the module to work. The cross-dependencies of all modules included in a project must form an acyclic graph in order for the project to compile successfully. A plugin is a module with additional functionality that aims to enhance its portability. Unlike general modules, plugins can store associated assets and can be managed from the editor's UI.

The developed StyleBlit implementation for Unreal Engine is a plugin that contains several blueprint types (i.e. spawnable entities that can be placed in a scene) as well as assets (such as mesh objects and textures) and levels that demonstrate the usage of the plugin. The assets and levels are not required for the plugin to compile and can be removed when not used. The plugin contains two inner modules: StyleBlit and StyleBlitEditor. The former contains a single actor (i.e. Unreal Engine entity that can be placed in a scene independently and is able to perform custom logic) class definition

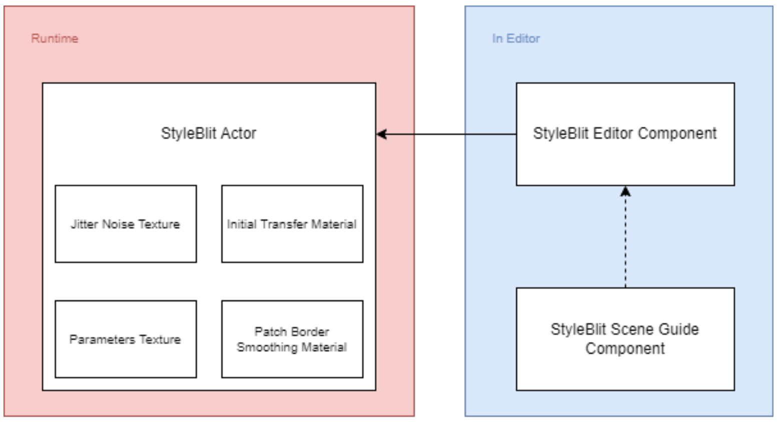

Runtime
StyleBlit Actor
Jitter Noise Texture
Initial Transfer Material
Parameters Texture
Patch Border Smoothing Material
In Editor
StyleBlit Editor Component
StyleBlit Scene Guide Component

Figure 15 A schematic overview of the StyleBlit plugin structure. The StyleBlit Scene Guide Component generates the guide channel and LUT textures that the user can select to be used for stylization in the Editor Component. The Editor Component communicates the actual StyleBlit settings and the assembled texture arrays to the StyleBlit actor, which contains the main StyleBlit logic.

and the definition of StyleBlit parameters object.

The StyleBlit actor contains a post process component which is used to set the StyleBlit post process materials in a camera placed in the same scene. The actor is also responsible for jitter noise texture generation and assignment to the corresponding material parameter. Additionally the StyleBlit actor handles the parameters texture buffer creation and writing. The parameters texture can be updated on tick for dynamic runtime StyleBlit adjustments.

The StyleBlit parameters object is used to manipulate the stylization parameters in real time. It is a simple container for a number for numeric variables with additional functionality to convert the stored data into a set of colors that can be written into the parameter texture. The parameters object is separated into a self contained entity to ensure the correct serialization when performing game builds.

Together StyleBlit actor and StyleBlit parameters make up the minimal required logic to perform StyleBlit in a game runtime. However, the adjustments of separate exemplars' parameters requires the user to perform a lot of manual asset manipulations which could be automated. The StyleBlitEditor module contains several entities that help automate the majority of the trivial operations needed to properly set up StyleBlit for runtime. It also contains the functionality for LUT generation based on a provided scene. These operations are not expected to be performed in the game and thus the module is only compiled with the editor builds of the plugin.

The majority of the implementation is contained in the corresponding shaders. In Unreal Engine the shaders are usually managed as materials (i.e. assets that provide the interface to update parameter values as well as other features). The primer way to create shaders in Unreal is by means of the materials editor. It provides functionality for flexible shader editing via visual programming. The material nodes are compiled into shader code written in HLSL. The node compilation phase also includes an optimization step, in which the operations may be restructured to achieve better performance. However, in certain cases a direct control over the generated shader code is needed, for instance, when the shader relies on functionality that is not exposed as a node in the editor. For these cases Unreal proposes a custom expression material node. In it the users can specify direct HLSL expressions, which are pasted directly into the generated shader. Unlike with the rest of the material nodes, the additional optimization stage is

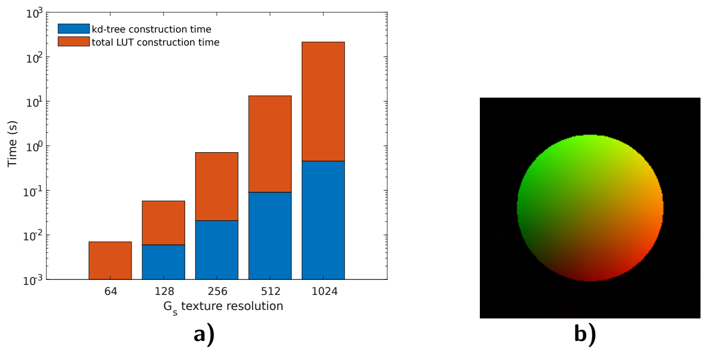


Figure 16 LUT construction times (*a*) for the guide channel texture (*b*) with different resolutions. The higher resolution LUTs take significantly longer to compute, but the render target has limited precision to 8-bit per channel so the guide values stored in it can take only 256 discrete values and the larger textures do not result in better style transfer quality.

not performed for custom expressions.

The developed plugin contains a number of post process materials responsible for either the initial style transfer stage or the chuck border blending stage of StyleBlit along with supplementary post process materials. There are several variants of each of the post process materials which correspond to different extensions implementations. The implementations are separated into different materials because switching the implementations by means of a macro would require constant recompilations. The StyleBlit post process materials can be used as any post process materials, i.e. applied via a post process component or directly in the camera settings, however, in order for temporal jitter to work they must be assigned to a StyleBlit actor instance in the scene.

The editor part of the StyleBlit plugin handles two aspects of the StyleBlit usage. The first is the exemplar scene guide channel texture generation and an appropriate LUT precalculation. This is handled by a StyleBlit scene guide component. This is a utility object which contains functions for LUT generation based on a supplied texture or a render target. In case of using a texture, the LUT is written into an image file and saved into a dedicated folder on disk. In case of a render target, the render target is automatically converted into a texture and saved together with the LUT. To use the render target for LUT construction, a scene capture actor has to be set up in the scene. A scene capture actor is an entity that contains a camera and is able to render the scene viewed from its camera into a render target. The scene capture may use a post process material to render the required scene guide channel into the render target. The same render target can then be referenced by the scene guide component to use for LUT generation. When including the generated textures into an Unreal project it is important to explicitly switch off the sRGB texture asset parameter that may be on by default. This setting serves to preserve the apparent colors of textures during the tone mapping, however, the guide channel and LUT textures are not meant to be displayed directly and their pixels have to be read in the same way they were written.

The LUT is constructed using an external implementation of an n-dimensional kd-tree (as described in [18]). The implementation can be found online here: [29]. It is a minimalistic header-only implementation of the data structure that evaluates spatial queries by recursively walking down the tree hierarchy. The tree is constructed using the basic approach: the splitting plane orientation is selected in the round robin manner (i.e. the orientation is determined by the depth of the corresponding node in the tree), the position of a splitting plane coincides with the median of the node's associated data entries. This is a naive approach, which may perform worse than other more sophisticated construction criteria (discussed in detail in [30]), however, the LUTs are constructed once and the performance of the kd-tree implementation is not crucial. The current implementation was selected for its conciseness and it performs the LUT construction for guide channel textures of an appropriate size in reasonable time.

The second aspect of the StyleBlit usage the editor part of the plugin has to handle is the asset manipulations. Without any automatization the user would have to assemble the texture arrays that contain the exemplars and their corresponding guide channel textures (and additionally a LUT array when it is used instead of the analytical solution) by hand. The StyleBlit editor component enables a certain level of automation for this process. It is a container for an array of exemplar parameters which are stored together with the associated exemplar textures. The component takes care of combining the textures into the respective arrays and communicating the parameters to the main StyleBlit actor. The editor component also has an interface to quickly assign exemplars to actors in a scene. The component relies on the asset manipulation functionality that is only available in editor builds of an Unreal project.

3.3 Proposed StyleBlit Extensions

Along with the general algorithm, the proposed StyleBlit implementation includes several extensions of the algorithm that aim to alleviate some of the issues discussed in the original article and to increase the number of feasible usage scenarios. The extensions are implemented in the form of additional StyleBlit post process material variants to allow the users to quickly switch between different versions of the effect. Internally, the extensions usually require a certain amount of restructuring of the material. Some of the additional features are wrapped in a dedicated macro inside the custom expressions code for easy toddling. Others differ from the basic algorithm more substantially and require changes in the entirety of the algorithm. These extensions are demonstrated and discussed in this section.

3.3.1 Edge-sensitive Chunk Border Blending

One of the discussed issues of the current algorithm is the unwanted loss of geometry details during the chunk border blending stage. The current blending stage relies on a voting approach and operates under the assumption that differing votes of the pixels in a given neighborhood must indicate a border of a chunk. This assumption generally holds true, however, a chunk border can be formed in a number of different ways. A chunk's border is influenced by the underlying change in geometry and by the seed (or landmark pixel) selection.

The geometry changes that influence the chunk borders can be of two types: gradual and abrupt. The gradual changes occur when on a continuous surface, they introduce chunk borders that are meant to be blended by the original algorithm. The abrupt changes in geometry correspond to the object contours. These regions communicate important information about the shape of the object and it is preferable to apply less blending here. However, the current algorithm applies as much blending in case of an abrupt chunk border as in the gradual case. This is due to the votes of the pixels separated by an abrupt border being significantly different from each other.

A solution to a similar problem is mentioned in [10]. In StyLit the finer geometry details are lost due to the iterative application of region averaging in the process of finding nearest neighbor fields for individual image regions. The proposed solution

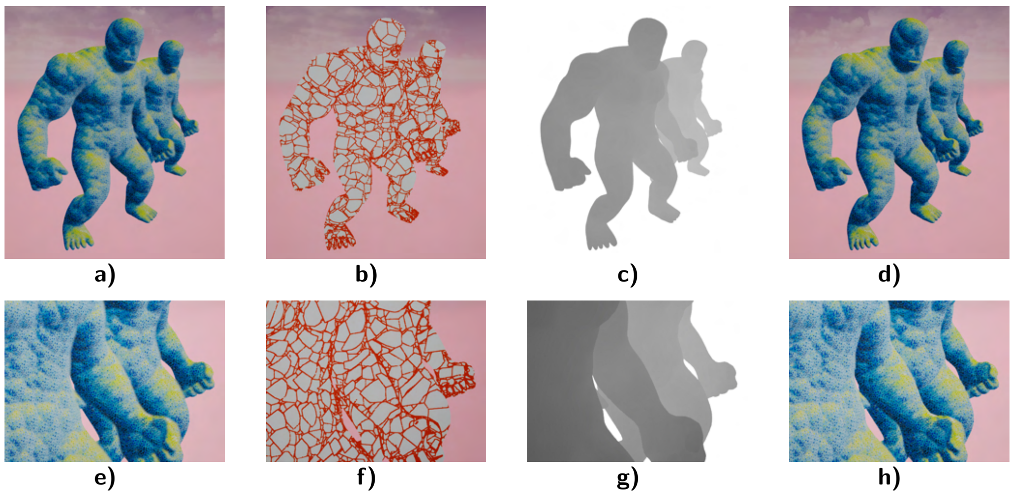


Figure 17 Edge-sensitive chunk border blending approach demonstrated on a scene with overlapping of objects: a, e - no edge abrupt detection, the contours of objects are blended, b, f - visualization of chunk borders, c, g - the scene depth buffer, d, h - blending with respect to the depth buffer.

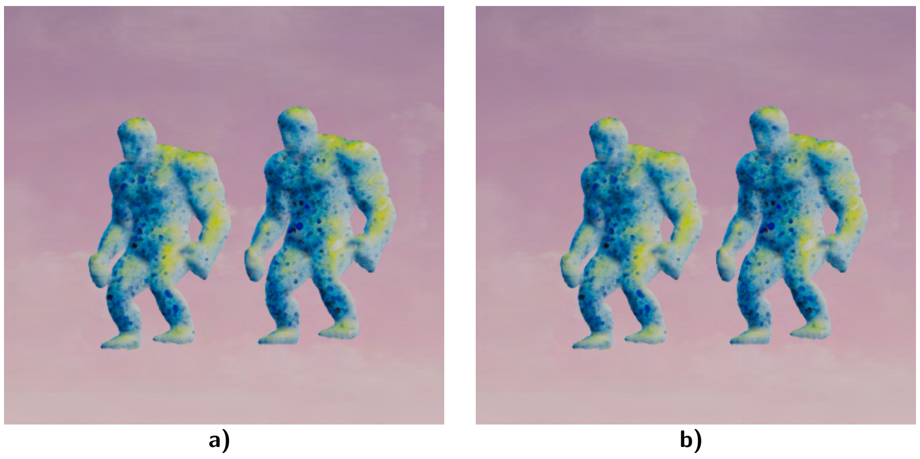


Figure 18 Chunk border blending applied on objects, when the blending radius takes up a significant portion of the objects' image projections: a - basic approach, b - blending with respect to the depth buffer.

for the detail blurring is to use an additional edge map channel. The channel can be produced by applying a general edge detection algorithm onto the exemplar and the target images. This solution not only solves the over-blending of finer geometry details, but also allows the accurate preservation of artistic rendering of the objects' contours. However, this solution increases the dimensionality of the guide channel and thus increases the complexity of the nearest neighbor search queries.

The proposed StyleBlit implementation contains a slightly different approach to geometry contours preservation. Instead of extending the guide channel, the additional information is used to drive the blending applied on a pixel. A pixel's vote is not accounted for, if the pixel is separated from the current pixel by an abrupt edge. The abrupt edges are determined using the scene depth render buffer which contains information about the distance from the camera to the geometry that occupies each pixel. An abrupt chunk edge is detected, if the difference in depth between the processed pixel

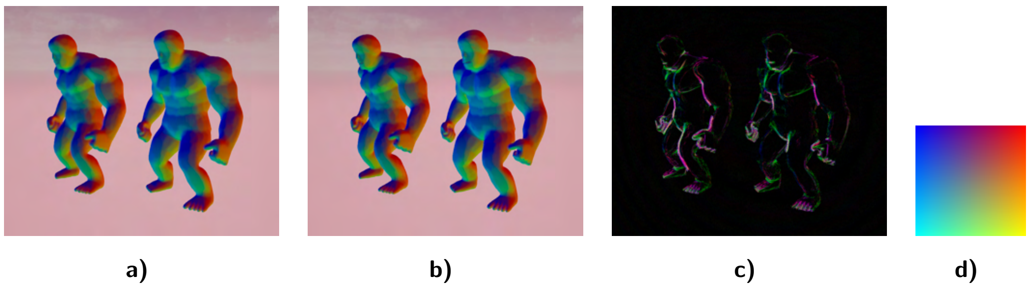


Figure 19 Bending stage with a simple gradient (d) applied as the exemplar: a - basic approach, b - edge-sensitive blending, c - visualization of the differences between a and b (the blending in b is limited mostly near the contour lines). The image c was adjusted to make the actual difference visible.

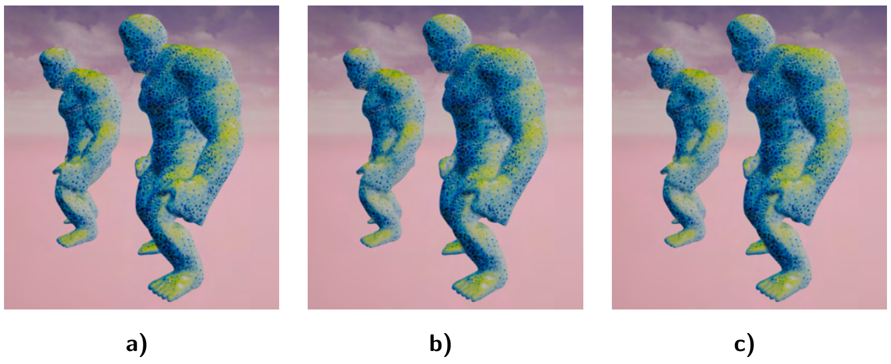


Figure 20 Blending results for different values of depth tolerance: (a) $t = 0.1$, (b) $t = 10$ (noticeable ghosting), (c) $t = 100$.

and a neighbor pixel is higher than some depth tolerance value t. The value t can be controlled by the user to achieve a desired result.

The modified blending stage is able to preserve more geometry details in challenging scenarios. Unreal Engine stores the scene depth information in an inverse manner (as described in [31]), which optimizes the distribution of the available depth values. This allows the modified blending approach to perform comparably well on the objects with different scene depth. The new blending approach has several draw-backs. It is unable to distinguish between different objects and relies solely on the tolerance parameter to guide the blending. It may make it difficult to set up the effect to work uniformly on a real scene. The blending may result in ghosting artifacts for some tolerance values (depicted on figure 20).

3.3.2 Colorized StyleBlit

Different approaches to image stylization treat the color present on the target image differently: some consider it a part of the style that has to be replaced by the exemplar, others preserve the color in the resulting image. The original formulation of StyleBlit does not take the color of the target image into account and the entirety of the color of the result image is formed by the style exemplar. This gives the artist complete control over the final look, but also limits the range of the algorithm's applications.

The color channel of the target image may contain semantically important information that can be desirable to be present in the stylization result.The basic StyleBlit algorithm can be used to communicate such information only if it is encoded in an

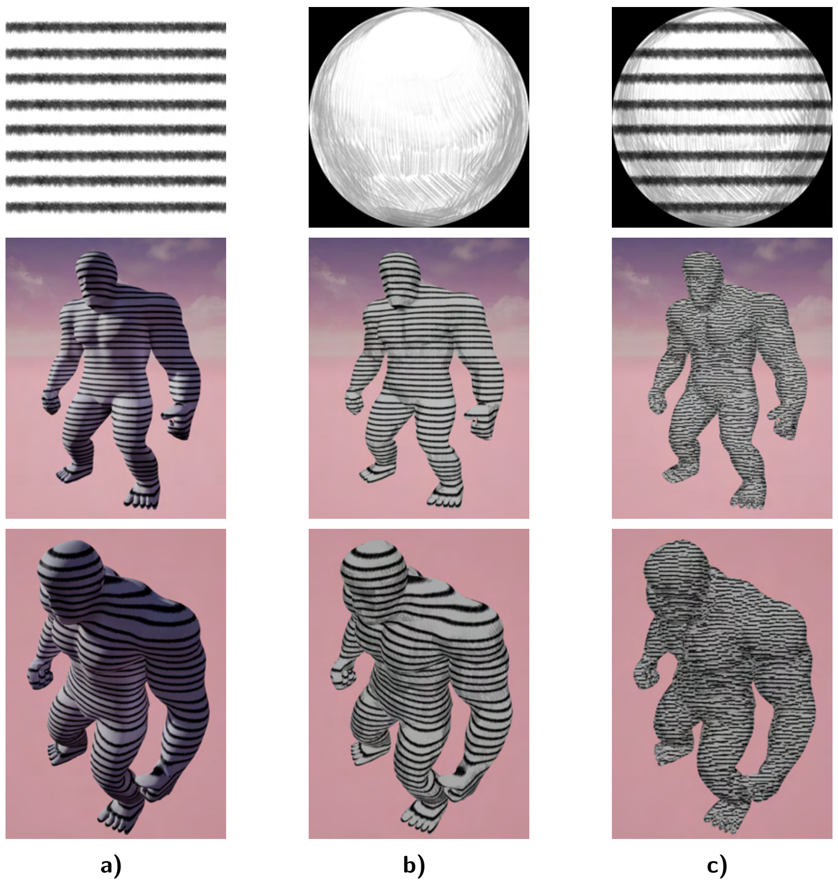


Figure 21 StyleBlit applied to communicate scene color information: *a* - no stylization applied, the stripes texture (top) is used a diffuse texture of the object, *b* - StyleBlit with color blending, the gray-scale exemplar (top) is used to communicate shading information, combined with the diffuse color to preserve semantically important stripes, *c* - an attempt to communicate the stripe texture by means of a modified exemplar (top), the misalignment of the chunks introduces visual noise, the direction of the stripes does not follow the object surface.

alternative channel. The information about color details can be contained in the exemplar directly, however, these details are not perceived as the characteristics of the object but rather as the style details. This is due to two aspects of the stylization. The first obstacle for encoding object color detail in an exemplar is the misalignment of the style chunks. The second is the fact that exemplar details are stored and transferred in screen space coordinates. This means that the details do not respect the orientation or scale of the underlying object surface.

As a work around, the differently colored parts of an object can be separated into individual objects. The corresponding exemplars can then be assigned to those objects thus preserving color semantic information without any changes to the general StyleBlit algorithm. This solution is possible, but it introduces multiple limitations. In practice, the color information of a mesh object is often stored in the form of a texture and the individual color details do not have to be aligned with the mesh edges. In this case it is not possible to apply this work around to preserve color. It is also not possible to have smooth color transitions with the basic StyleBlit approach.

The developed StyleBlit implementation for Unreal Engine allows to blend the output stylization with diffuse color of the scene objects. The blending is executed by multiplying the output stylization pixels by the pixels of the diffuse color buffer. The

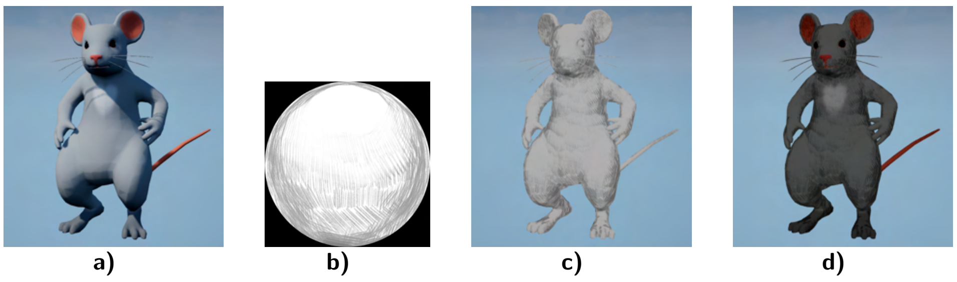


Figure 22 StyleBlit applied to a character with a diffuse texture: *a* - no stylization, *b* - style exemplar, *c* - basic StyleBlit output, *d* - StyleBlit output combined with the diffuse color.

weight of the effect is controlled by a parameter. This approach widens the possible applications of StyleBlit without introduction of a significant computational overhead. The current implementation works best for gray-scale style exemplars which do not interfere with the scene color during multiplication. The disadvantage of the proposed approach is the unpredictability of the style transfer result. The artists do not have the direct control over the result as opposed to the original algorithm, instead they have to check with the stylization result during the exemplar creation process.

3.3.3 Illuminated StyleBlit

The information about a scene's lighting is important for human perception of the three dimensional objects in it (e.g. as discussed in [32]). While not necessary, the shading information can enhance the immersive qualities of a virtual scene. It provides additional information about the spatial relations between the objects and the geometrical features of each individual object. Beyond the pragmatic aspect, a virtual scene lighting can be used by an artist as an expressive tool. The lighting can communicate the mood of a scene or serve a certain gameplay purpose in a game. The original StyleBlit algorithm does not utilize the scene lighting information which can be a significant limitation when it comes to real world applications. It also makes it difficult to combine the results of StyleBlit with the rest of the scene into a visually coherent image as there may be little information about the objects' interrelations.

An obvious approach to extend StyleBlit to incorporate scene lighting information is by adding additional layers to the guide channel. For instance, instead of using just the view-space normal vectors as the style transfer guide channel, the pixels' luminance can be combined with the normals to provide more informed guidance to the transfer process. This approach introduces two major issues. The first is the higher guide channel domain dimensionality, which results in more computationally heavy nearest neighbor queries. It would not be possible to apply the analytical NNS solution even in the case of using the simple sphere exemplar scene, because the lighting would not behave in the same way normal vectors do. In turn The LUT solution would introduce significant memory costs and make it more expensive to look up the precomputed results. Alternatively one can employ an auxiliary spatial data structure to handle the nearest neighbor queries in real time with more optimal memory usage (an example of fast kd tree implementation for highly parallel processor architectures can be found is [33]). However, another limiting factor of the guide channel extension lies in the nature of the exemplar data.

It has been discussed in the previous chapters that one of the benefits of the simple sphere scene selection for the exemplar guide channel is the exhaustive coverage of the guide channel domain. However, in the case of the extended guide channel that includes the luminance information the coverage provided by the exemplar guide channel leaves significant areas of the domain uncovered. For those areas the high assignment error would result in the abuse of certain parts of the exemplar during the transfer or other artifacts. It is difficult to create an exemplar scene that would contain a majority of the guide channel domain values (i.e. a majority of the possible surface orientations under a wide range of lighting conditions). This can be remedied by introduction of multiple additional lighting layers, in which case the error in one of the guide value components can be leveled out by other channels. This approach is used in StyLit, where the high-dimensional nearest neighbor lookup is a significant contributor to the overall computational complexity.

Another alternative for the lighting information incorporation into StyleBlit is the introduction of the secondary guide channel values. The main idea of the approach is to split the nearest neighbor query evaluation into two stages. The first stage performs the lookup among the primary guide channel values, e.g. normal vectors, in the same fashion it does in the original algorithm. Once the result of the main lookup is available, the second stage performs a simple search among a relatively small array of the associated secondary values. The primary guide values retrieved in the first stage combined with the secondary stage results constitute the approximate nearest neighbor. It is worth noting, that such search does not retrieve actual nearest neighbors and it does not guarantee any limitations on the approximation error of the result, however, the precision is not crucial in the context of StyleBlit.

The proposed StyleBlit implementation uses the secondary guide channel values approach to extend the original algorithm by incorporating shading information. The primary guide values are normal vector x and y coordinates, and the secondary guide value is the pixel luminance. It operates in a similar vein as StyLit and assumes that a fine artist may approach rendering of different lighting conditions differently. This assumption is used in the following way: the artist provides a number of exemplars, each of which corresponds to the rendering of a lighting scenario. A luminance value $L_i \in L_e$ is assigned to each of the exemplars. When performing the nearest neighbor search the primar search stage is executed in the normal vector coordinates. Then the closest exemplar luminance L_i is selected from the array of exemplar luminances. The assignment error for the pixel is then a weighted sum of the normal vector NNS error and the difference of the selected exemplar luminance and the target pixel's luminance value (see equation 3.3.1).

$$CalculateError(p, q) = ||N(G_t[p]) - N(G_s[q])|| + \alpha||L(G_t[p]) - L(G_s[q])||. \tag{3.3.1}$$

Here $N(g)$ retrieves the normal vector coordinates from the guide channel value g and $L(g)$ retrieves luminance.

The effect of the modified approach is similar to the popular toon shading technique (described in a following section), the shading forms distinct bands that correspond to different brightness levels. A similar method could be implemented in the chunk border blending stage, where instead of using just the corresponding stencil value to read from the exemplar texture array, the appropriate exemplar index is retrieved based on a pixel's luminance. However, this method does not provide the user with any control over the shading. Furthermore the shadows may look distinctly digital when the smooth shadow borders are juxtaposed with the juggeded chunks and rough brushwork of the exemplar. Incorporating the shading information into the transfer stage allows for a certain amount of user control over the precision of the shadows.

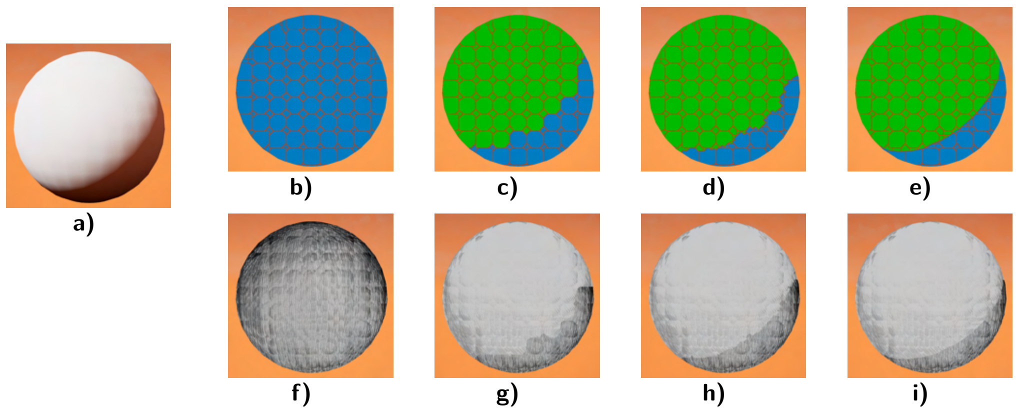


Figure 23 Visualization of the illuminated StyleBlit style transfer with 2 exemplars on a simple scene: a - the scene with no stylization applied, b, c, d, e: style chunks visualization (red lines denote chunks borders, green ares correspond to the lighter exemplar, blue to the darker exemplar, the chunk noise texture is disabled for more obvious visualization), b - using just the darker exemplar, c - using 2 exemplars, the shading weight $\alpha = 0$, d - $\alpha = 0.1$, e - $\alpha = 0.5$, f, g, h, i - the corresponding style transfer results.

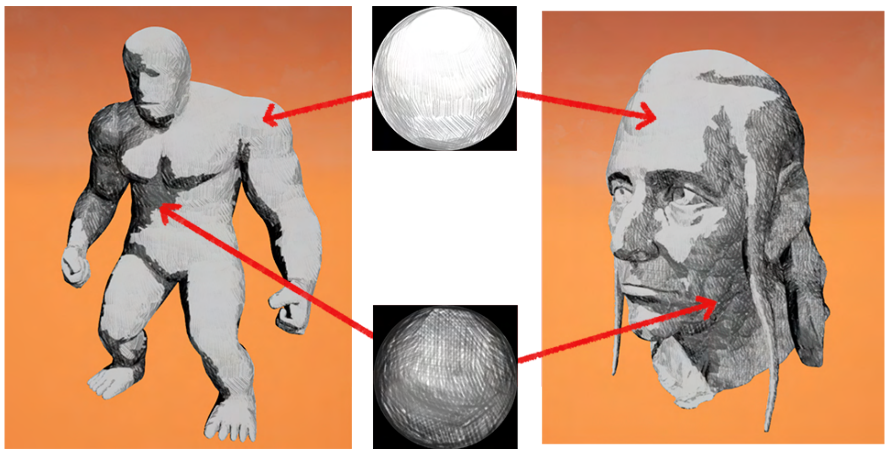


Figure 24 Demonstration of the illuminated StyleBlit on Golem and Portrait1 models using 2 exemplars that aim to simulate a pancil drawing style.

The proposed approach can be easily integrated into the basic StyleBlit algorithm by changing the body of the error calculation function. The weight α of the luminance in the error function can be used to control how precisely the shading information is preserved during the transfer. The implementation allows for up to 4 different exemplars tied to different luminance values to be used on one object. A higher number is deemed to be impractical as it would require the creation of multiple different exemplars and the width of the individual shading bands would be smaller than the average size of a chunk. The low number of secondary values means that the additional secondary search stage during the nearest neighbor lookup comes at little computational cost.

Unlike the original StyleBlit, the extended version relies on the output of multiple preceding stages of the rendering pipeline: normals taken from the G-buffer and shading information. The latter can be retrieved as the output of different stages leading to slight differences in the result. The selection of the stage that outputs the lighting information influences which post process material slot is to be used to apply StyleBlit effects. The first option is to apply the material after the tone mapping stage. This

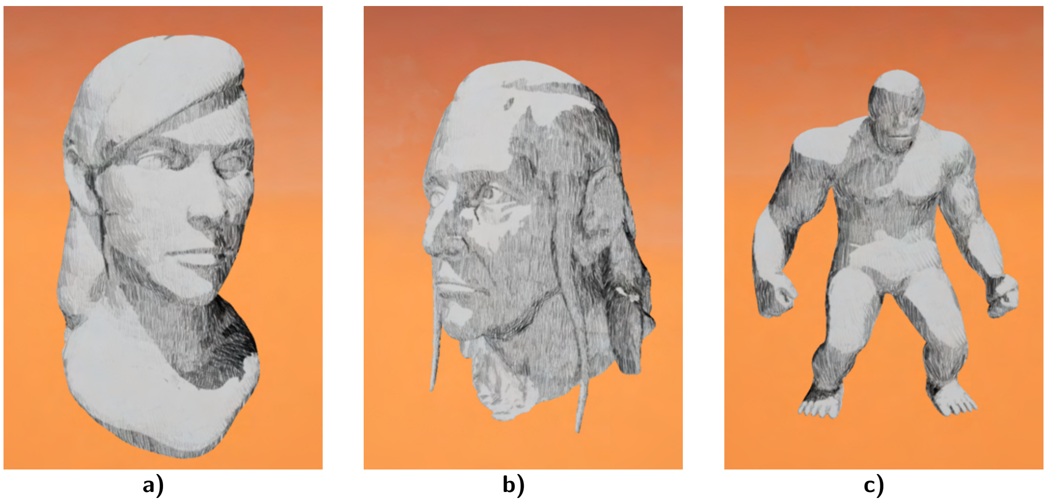


Figure 25 Additional demonstration of the illuminated StyleBlit on models: *a* - Portrait2, *b* - Portrait1, *c* - Golem.

has the benefit of preserving the precise color of the exemplars. However, in this case the lighting information supplied to the illuminated StyleBlit is a subject to tone mapping and may behave unexpectedly, as the tone mapper adjusts the brightness of individual pixels based on global statistical characteristics of the image which change with the content (an overview of tone mapping techniques can be found in [34]). The implementation chooses the alternative of placing the StyleBlit post process materials before the tone mapper for more controllable results.

3.4 Alternative Stylization Approaches

The extended StyleBlit algorithm is suited best for stylizing a certain group of geometric objects. It inherits the limitations of the original approach in this regard and performs poorly on objects with mostly flat surfaces. The application of the algorithm on objects that occupy large parts of the screen also results in visible pattern repetition induced by the abuse of certain parts of the exemplar. In practice this means that the algorithm is rarely suited for stylization of environments and is more useful when applied to characters or general organic shapes.

It may be challenging to create a stylized environment that would fit the style of the characters with StyleBlit applied to them. This section describes the proposed additional stylization techniques contained in the developed implementation. Although only tangentially related to StyleBlit, these are included in the plugin to enhance the usability of the algorithm by providing an example of fully stylized results that aim to form a coherent visual style.

3.4.1 Toon Shading

A popular approach to virtual scenes stylization is the application of toon shading (sometimes referred to as Cel-Shading, however, Cel-shading is generally a broader term). The general technique (described e.g. in [35]) performs unary operations of the pixels of the stylized objects to retrieve a non-realistic final image. The approach may be used for stylization of photographs (e.g. as discussed in [4]), however, the additional information available during virtual scene stylization enhances the possibilities of the

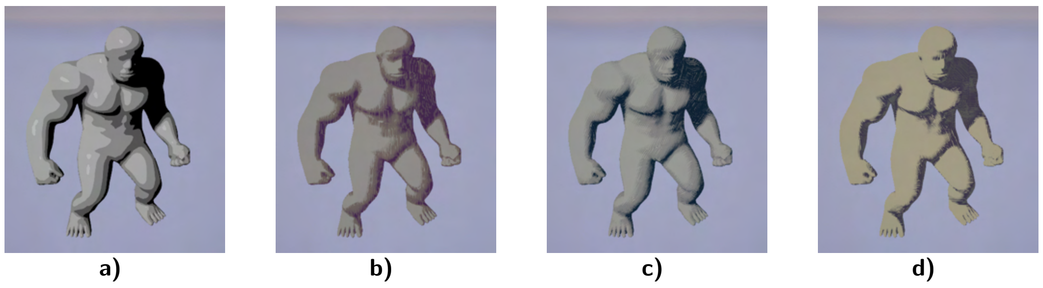


Figure 26 Toon shading applied to the Golem model: *a* - basic toon shading by applying a piece-wise constant transformation function, *b* - blending of the bands' edges using a hatching, *c* - the hatching texture mapped onto the model and then used for bands' edges blending, *d* - the hatching texture is divided in cells and each cell is oriented to align with the steepest normals change of the underlying surface.

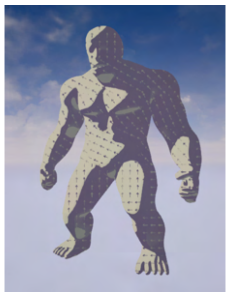


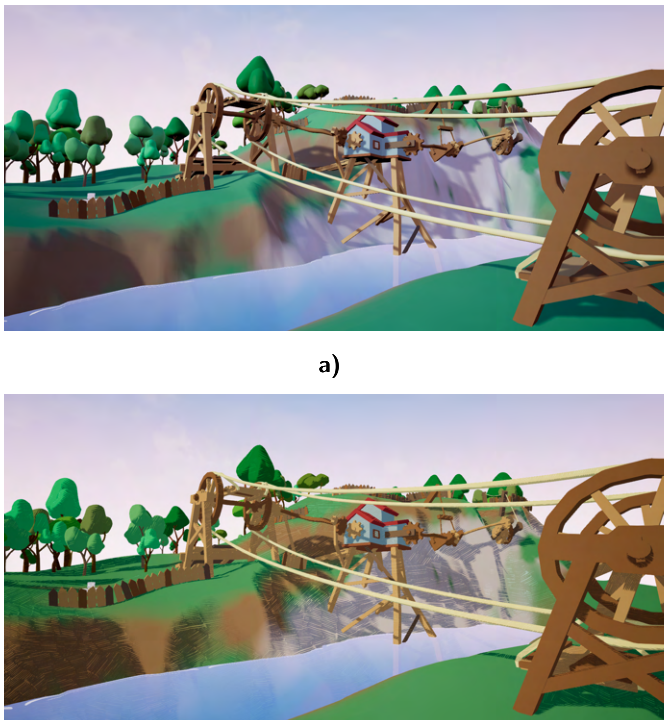
Figure 27 Oriented textures for toon shading: a visualization of the corresponding texture orientation (arrows denote the direction of the steepest normals change for a given texture cell).

technique. An important advantage of the technique is its computational efficiency, as the pixels of the target image are processed independently based on global input parameters without any additional costly operations.

A common approach to toon shading is to transform the target pixels' color based on the associated shading information. The article [35] suggests an extension of the shading model applied per pixel. The suggested algorithm uses the basis of the common Phong shading model which relies on the angle between the normals in rendered points of a surface and the corresponding vectors towards a lightsource. The information about the angle is used to interpolate between two selected colors to produce a result image which contains more information about the scene geometry.

Later extensions of the described approach suggested more sophisticated color mapping techniques. The article [36] presents complex non-linear transformations during the shading computations to achieve a specific visual style. A more universal approach is proposed in [37]. In it the color transformation is based on the orientation of the surface against the light source and an additional information channel. Notably, the approach uses a texture to define the result color transformation, as opposed to the former approaches, which used an analytical definition.

These approaches allow for significant modifications of the target images when compared with a conventional shading model output. However, they can be applied in the stated form only for a limited set of scenes. Namely, they operate under the assumption that the scene is lit by a single light source. They also do not take more complex lighting phenomena (e.g. cast shadows) into account. A common alternative approach is to use the output of a complete shading calculation pipeline as the value guiding the



a)

b)

Figure 28 Toon shading applied to the rendering of an environment: *a* - no post process materials, *b* - normal oriented textured toon shading.

color transformation This allows to apply the toon stylization on the cast shadows and other complex lighting phenomena, but it is also restricted by the lighting output. The areas with flat lighting cannot have gradients after the stylizations. This would not be the case with the previous approaches.

As mentioned earlier, there are several ways to encode the pixel color transformation for toon shading. A popular approach uses a lookup table (i.e. a texture, where each pixel corresponds to a possible guide channel value). It is flexible and easy to implement. Alternatively the transformation can be defined parametrically. It is less flexible but may provide additional information about the transformation structure that is difficult to deduce from a LUT. The parametric approach also enables the user to edit the shading on the fly without having to edit a texture.

The developed plugin contains a number of toon shading post process materials that utilize the output of the extensive Unreal Engine rendering pipeline for lighting information and use additional methods for more advanced results. Specifically, the most basic implemented approach uses a piecewise constant transformation function to generate a simple flat shading look. The lighting information is effectively quantized resulting in bands corresponding to different luminance levels. An extension of the basic flat toon shading material employs a texture to introduce natural-looking noise to the borders of the bands. The next extension separates the screen into irregular cells and orients the noise texture in accordance with the normals gradient calculated in the middle of each cell. This approach is similar to the one proposed in [38]. An alternative implementation maps the noise texture in the world coordinates and via procedurally generated surface UV maps and uses the output during the color transformation in the same way the screen space texture has been used in the previous two extensions.

The implementation contains several variants of the oriented texture toon shading material with different guide channels driving the texture cells' orientation, namely the normals guide, the shading guide, and the scene depth guide. For the latter two the gradient is calculated simply by finding the differences in the corresponding values of the neighboring pixels along the x and y axes. These differences are the components of the steepest guide gradient in the given pixel, which is then used to orient the texture. The shading guide channel's results are limited by the shading model used. The shading model used in Unreal tends to produce large areas with no change in luminance and thus the algorithm has to fall back onto a default orientation. The depth guide does not have an issue with the gradients, however, the result looks unnatural, as the orientation of the texture along the scene depth fails to communicate the geometry in a variety of cases.

To determine the steepest normal vector change a modified approach must be used. The suggested toon shading material uses the length of the difference vector between the neighboring pixels' normals as the metric for the gradient vector. This produces desirable results, however, the approach does not differentiate between positive and negative directions due to how the vector length is calculated. This limits the possible texture adjustment angle to just 90 degrees, however, this limitation is rarely significant. Due to using coarse gradient calculations (the orientation is calculated uniformly for coherent regions of the target image to preserve texture details), the gradient vector must adequately approximate the underlying guide channel information for the entire region of the image. The proposed solution prevents the result gradient from being skewed off due to a local feature by calculating an average over a selection of pixels that fall inside the region. In general this prevents the result orientation from drastic changes due to small features influencing the result.

3.4.2 Outline

Another popular approach to image stylization is outline rendering. It is an effective way to communicate the hand drawn nature of an image and preserve a lot of semantic information about the content of the image. The technique can be used in combination with other stylization methods to improve the overall result. Outlines can be applied to stylize both virtual scenes and photographs. The general technique is based on the notion of the contours, which can be defined in a number of ways. A common definition for virtual scenes stylization is as follows. A point of an object's surface lies on a contour against some view position, if the normal vector for the surface in this point is perpendicular to the vector towards the view position (e.g. as described in [39]). The definition is often extended by other contour conditions to help illustrate significant semantic region borders in the stylized image.

While many approaches to outline stylization treat the contour definition empirically aiming for a particular visual style, [40] provides a statistical analysis of how people interpret 3D shape into a line drawing and how well they can retrieve 3D shape from a line art rendering. The analysis indicates that there are several types of outlines that can coexist on a drawing and depict different aspects of the objects' shape.The lines in a line drawing may be dependent or independent on view position or lighting scenarios. The unification of the large variety of lines into a single approach is challenging and many popular approaches focus on a specific subset of the possible line features. Additionally, the line itself can convey additional information through the color, precision, thickness and other parameters.

There are several methods to render the outline of a virtual scene. A commonly used approach to outlines rendering is to use a back face render of an expanded model of the

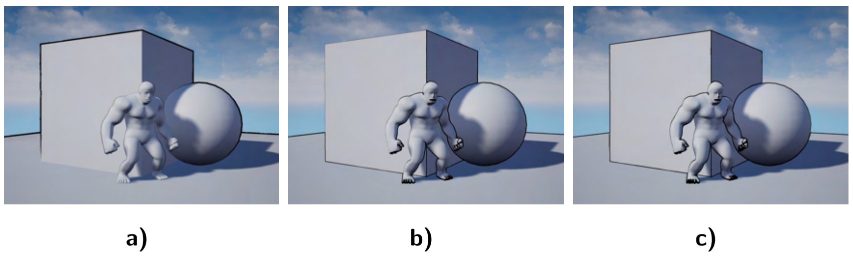


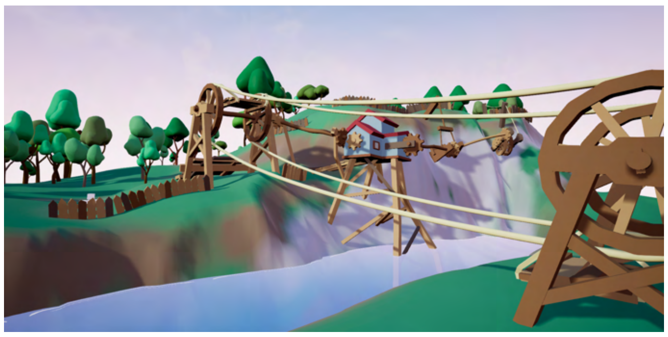
Figure 29 Outline applied to a simple scene: *a* - contours dictated by pixels' scene depth, *b* - contours by changes in normals, *c* - combined result.

stylized object and overlay it with the original model's rendering (as described in [41]). This method provides an important advantage: it can be used without any access to a render buffer, which reduces the computational requirements and allows the effect to be used with a minimalistic rendering pipeline. It also provides the artist control over the exact result. The expanded version of a model can be created by hand specifically for the task. However, this approach does not respect the different semantic meanings the lines can convey.

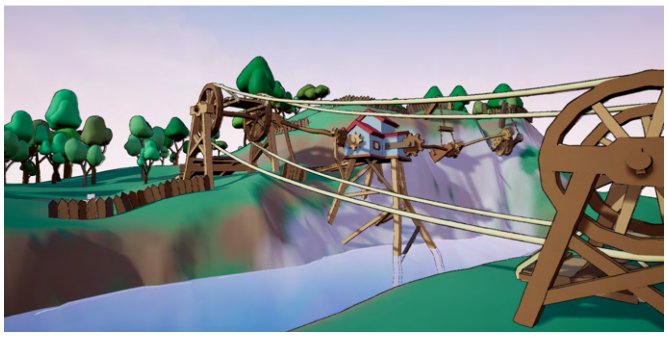
Alternatively, the outline effect can be applied in a post processing render stage, utilizing the access to the scene geometry information retrieved in the previous rendering stages. The approaches described in [39] and [42] rely on both contours as described earlier and the notion of ridges and valleys. Those are regions of an object's surface characterized by their curvature. These regions can often be denoted in a drawing by a line and so are included in the contour definition. Additionally the general contours are proposed to be calculated from multiple view positions closely located to the original view position to provide more potential outline information. The generated contour information is then gathered and interpreted as color to produce lines.

A challenge of a real time outline rendering implementation is the control over the line parameters. The contour criteria described earlier help determine which points have the line-inducing potential. However, the rendered outlines may have additional parameters and usually affect pixels which were not marked as contours in the gathering stage. The available expressive line parameters are among others the color, thickness, texture, and precision. The rendering of the outlines wider than a pixel requires additional care to ensure efficiency and flexibility of the implementation. The approach presented in [42] uses a more flexible definition of a contour point to produce line parameter variety. Instead of marking the pixels as either a contour pixel or not, the method assigns an opacity value which corresponds to the outline opacity. The texture and color can be additionally applied to the resulting line rendering.

An alternative approach to wide lines is described in the blogpost [43]. Instead of marking more pixels of the stylized image as contours, the approach suggests explicit expansion of the one-pixel lines generated in the gathering stage. There exist multiple ways to expand the rendered outlines. They are commonly executed as an additional pass over the result image that performs pixel-wise operations based on each pixel's neighborhood. A popular technique is to apply a version of a Gaussian blur to the initial outline image and mark each pixel with a contour value greater than a given threshold as a part of the outline. This can be executed relatively efficiently, however, the technique presents several disadvantages and is inferior in the performance to the suggested jump flooding method (as described in [44]). With the use of the jump flooding approach the outlines of arbitrary width and high precision are made possible in real time.



a)



b)

Figure 30 Outline applied to the rendering of an environment: *a* - no post process materials, *b* - using the proposed outline.

The proposed implementation contains several outline post process materials for the outline rendering. The materials take an approach to outline rendering similar to one described in [42]. The output of the rendering pipeline is analyzed to detect contour lines. The contours are generated based on the scene objects' surface curvature (i.e. changes in the normals in the direct neighborhood of a pixel) and scene depth discontinuities. The corresponding values are evaluated for each pixel of the screen and their weighted sum is assigned to the respective pixels. The sum is interpreted into the outline color based on a parametric transformation of the value. Some of the implemented outline materials apply a noise to the outline if the form of a texture whose luminance is subtracted from the pixel contour values to introduce more irregularities. Additionally a jitter is applied to the scene render buffers so that some of the final image pixels contain contour values of their neighbors. This jitter effect can also be controlled by the user.

In the following chapter the results of the implementation are demonstrated and analyzed.

4 Evaluation of Results

The implementation developed in this thesis contains the StyleBlit plugin for Unreal Engine 4.27 along with an example project consisting of a playable level that aims to simulate the algorithm's usage in a practical scenario and a demo level similar to the OpenGL demo application supplementing the original article. The demo level allows the user to switch between several models and apply a number of exemplars to evaluate the result of the post-process effect. The demo level also contains UI for the users to tweak the parameters at runtime.

The implemented playable level relies on a number of stylization approaches to get a uniform visual style on the entire scene. For this a combination of the implemented algorithms is used. The temporal jitter was disabled in StyleBlit and the toon shading as it created a disorienting effect. The final level uses StyleBlit only on a portion of the scene objects (the main character and the trees) and those use relatively mild exemplars to simulate a colored pencil drawing feel with an additional liner retouch. It has proved challenging to create a visually coherent style using more abstract exemplars. For this reason the showcase level is included to demonstrate a wider extent of the algorithm's capabilities.

This chapter is dedicated to the evaluation of the implementation results. The implementation is analyzed from several points of view. The visual quality of the stylization results is compared to the existing approaches. The algorithm's performance in the context of Unreal Engine is discussed. The structure of the plugin and its applicability for real-world development scenarios is considered.

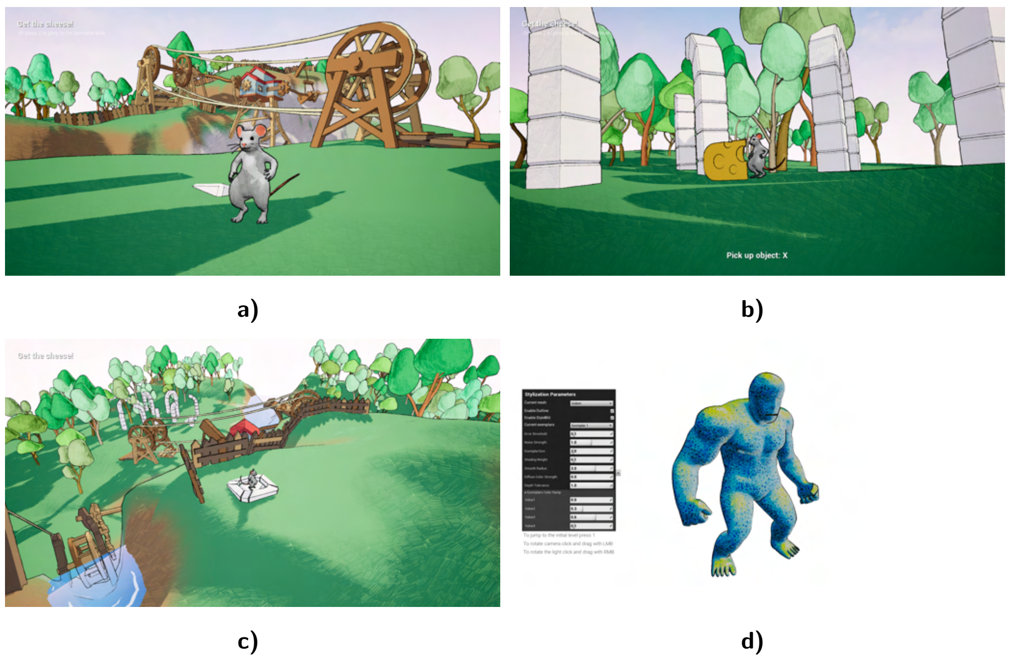


Figure 31 Unreal StyleBlit demonstration project: a, b, c - playable level, d - turn table showcase level.

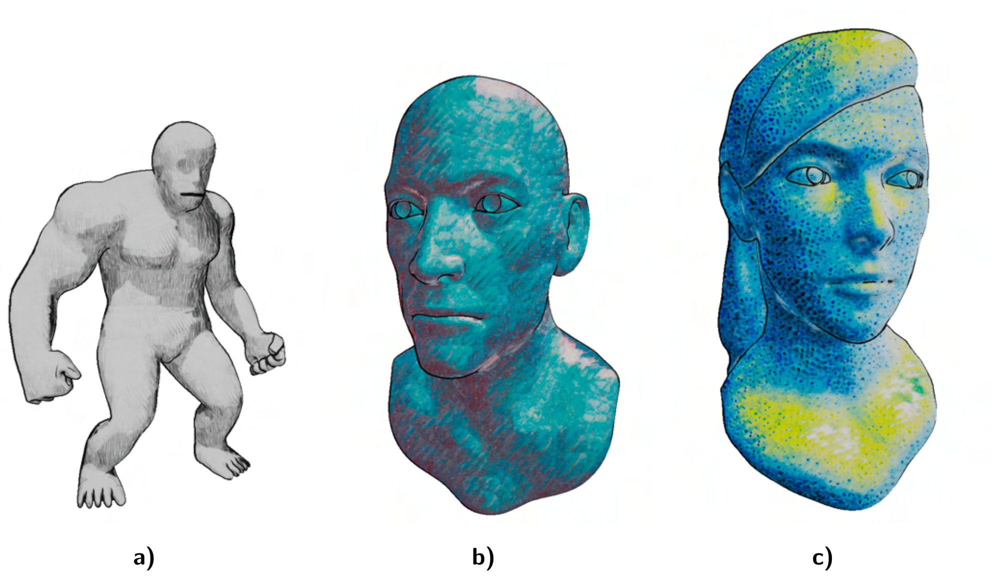


Figure 32 Demonstration of StyleBlit combined with an outline render on (*a*) Golem, (*b*) Portrait3, and (*c*) Portrait2 models using exemplars from the original article and new exemplars for the illuminated cases (screenshots from the showcase level).

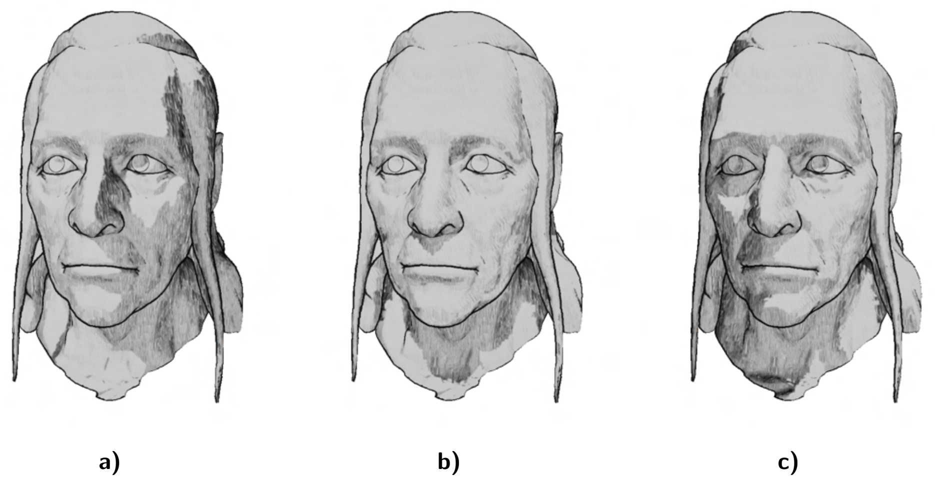


Figure 33 Illuminated StyleBlit demonstration (screenshots from the showcase level): Portrait1 model under different lighting conditions.

4.1 Comparison with Existing Stylization Approaches

In this section the Unreal StyleBlit implementation outputs are compared with the alternatives, namely, with reference solutions described in a previous chapter and other approaches to virtual scene stylization. The images generated by the StyleBlit plugin for Unreal Engine are first compared with the original article's demo application outputs (figure 34). The pixel-perfect matching results are difficult to achieve due to the different implementations using different units for each of the parameters and different camera setups. However, the results for the tested models demonstrate a sufficient level of similarity of the reference and proposed implementations.

The slight visual differences of the outputs can also be caused by the misalignments in the guide channel texture and exemplar texture sizes. The original OpenGL implementation handles it in an ad-hoc manner, each exemplar has an assigned individual number representing the desired exemplar size relative to the application window size. The proposed implementation takes a similar approach and allows the user to tweak the size of the exemplar texture, which can be beneficial in the cases of using low-resolution exemplars, but it also affects the results' visual correctness. Namely, the apparent curvature of the object may be not as steep as the as rendered in a scaled exemplar texture. This may not be a problem for flatly lit exemplars, but can influence the visual quality of the result in the case of a more descriptive exemplar. It could be solved by tweaking the remaining parameters of the stylization, such as the maximum error threshold.

The style transfer results are also compared to those of the StyleBlit plugin for Unity (figure 35). The biggest noticeable difference is the blurring of the rendered object's borders in the Unity implementation. The effect is not present in the Unreal plugin as it is deemed undesirable. Another aspect of the implementation which is worth exploring is the temporal jitter. The Unity plugin exposes a setting for the user to influence the refresh rate of the jitter noise texture used during the coarse neighbor lookup procedure. This is done by effectively capping the frame rate of the entire game. While this is logical and simulates the traditional hand-drawn animation, which often has a noticeably lower frame rate, it comes with certain disadvantages.

Lowering the frame rate may cause a tiring effect on the player. It also degrades the quality of some effects and algorithms that rely on updating the results from a previous frame. For instance, the implemented playable level uses an inverse kinematic solver to position the feet of the main character correctly on the uneven terrain using a method similar to [45]. To do so the transformation from the current feet positions to the closest points on the ground must be retrieved before the animation is updated. In practice this means using the previous frame's feet positions in the calculations for the next frame. When the frame rate is capped to a relatively low number the difference in the feet positions between the frames is too large and the inverse kinematics solver outputs visibly wrong results. The Unity implementation also allows setting a higher frame rate and only updating the noise texture once in every n frames to allow for the game updates with higher frequencies. The Unreal StyleBlit implementation has additional options for temporal jitter customization. The jitter can be enabled and a time period between the noise texture updates can be set. Optionally the user can choose to cap the frame rate to match the updates of the jitter noise.

One of the biggest advantages of StyleBlit over the commonly used cel-shading approaches (i.e. combinations of toon shaders with outline effects) is its versatility and control over the final look. In case of a cel-shading stylization, the results are often bound to a specific visual style and significant restructuring of the approach is required to achieve a different style. As mentioned in the previous chapters, StyleBlit is best suited for stylization of organic shapes and generally performs worse on flat surfaces, which tend to make up large parts of a game's environment. On the contrary, the cel-shading stylization provides room for adjustments and creation of a more universal stylization stage, which can be applied to the entire scene and recognisably translate the underlying semantic information for a wider range of geometric shapes when compared to StyleBlit. The comparison of the StyleBlit implementation outputs for a simple scene with those of the implemented cel shader are demonstrated in figure 37. In practice a variety of visual effects are often needed to be applied to different objects in one scene. The proposed implementation consists mostly of post process materials which can be easily applied in a chain. The advantages of different stylization approaches can thus be combined for creation of a desired visual style.

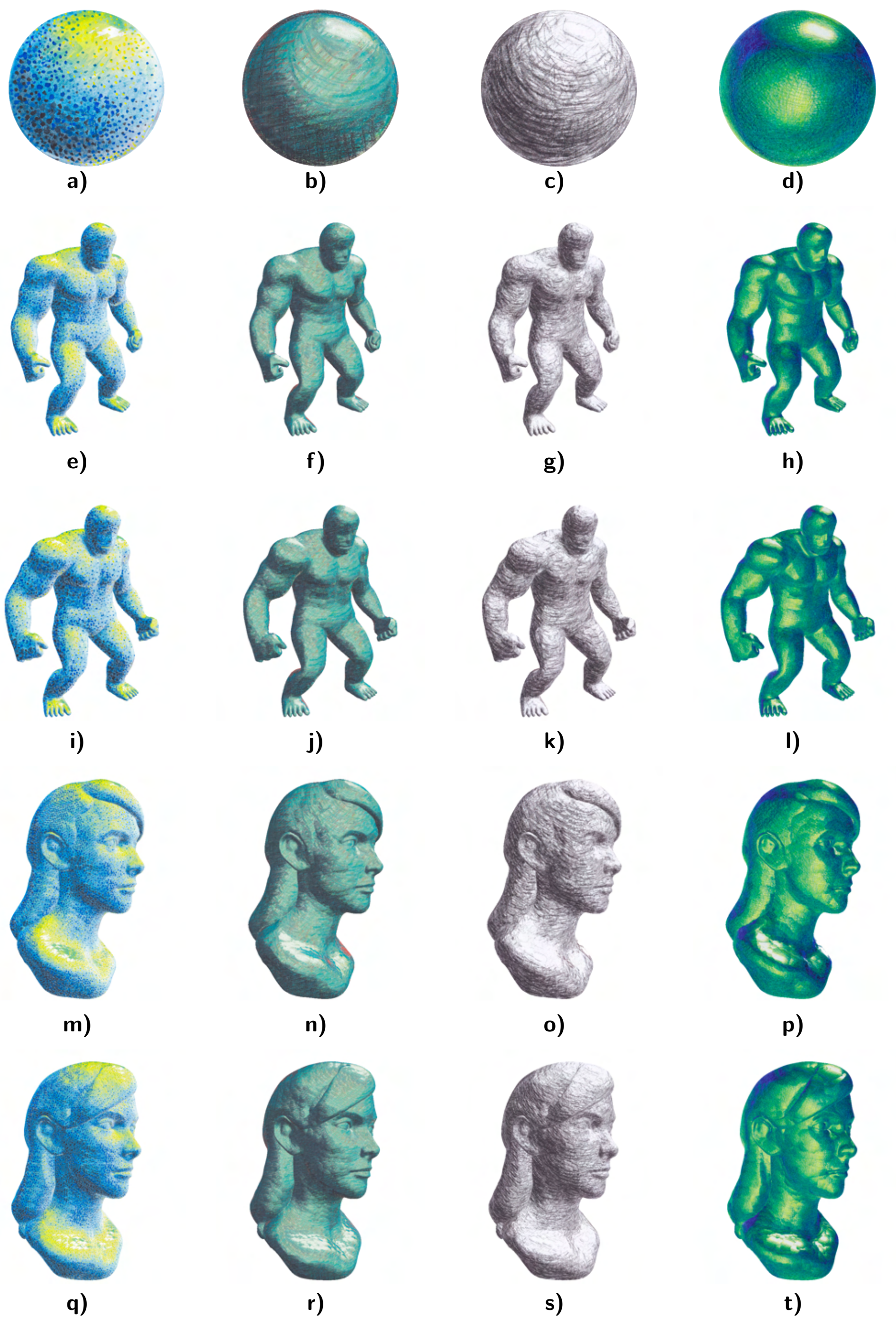


Figure 34 Comparison of the Unreal plugin output with the reference OpenGL implementation: $a - d$ - exemplars from the original article, $e - h$ - Golem model in the OpenGL application, $i - l$ - Golem in Unreal, $m - p$ - Portrait2 in the OpenGL application, $q - t$ - Portrait2 in Unreal. The models are mirrored in OpenGL implementation due to the import implemetation.

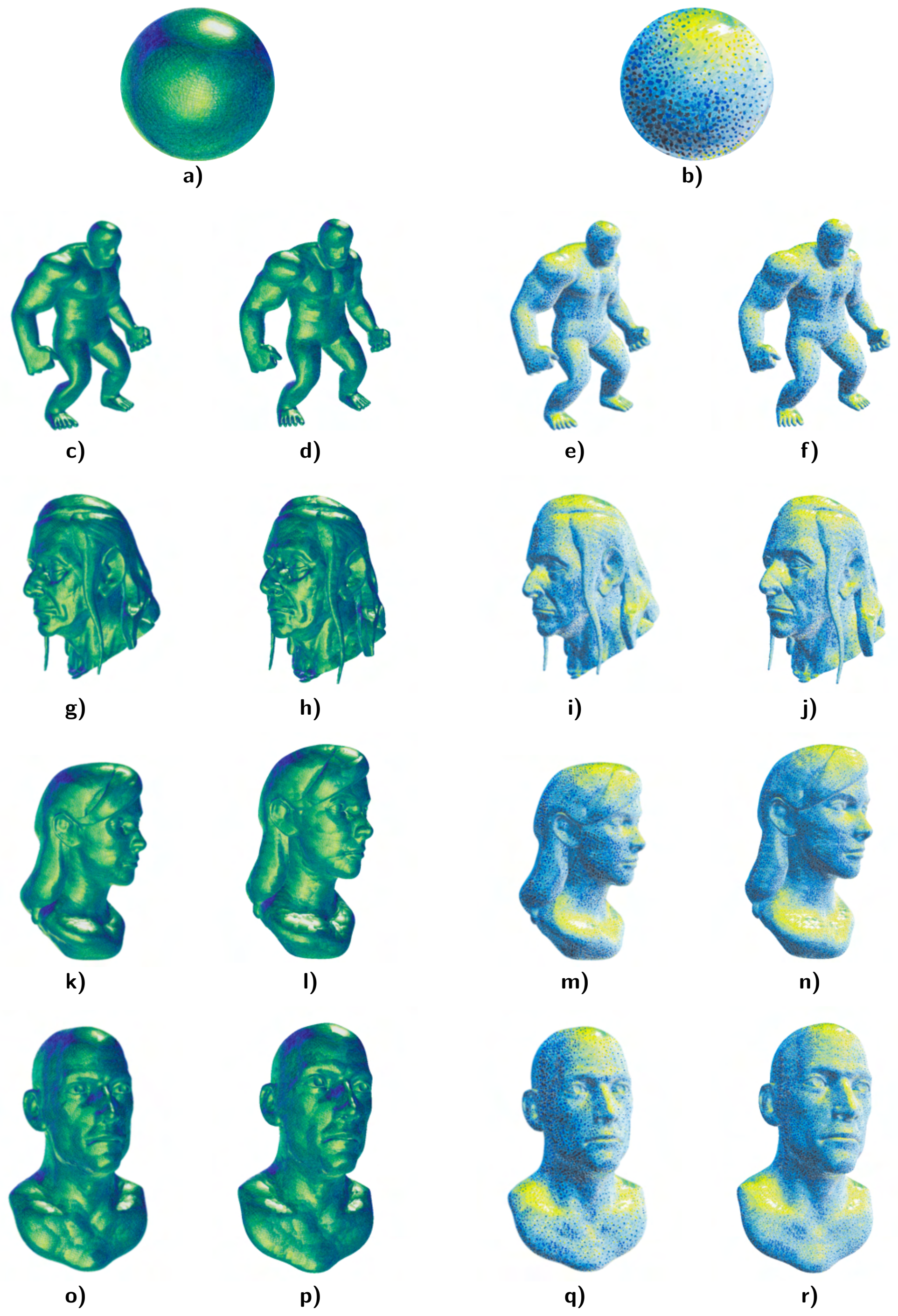


Figure 35 Comparison of Unreal and Unity StyleBlit implementations: a, b - exemplars; c, g, k, o - Unity outputs for exemplar a; d, h, l, p - Unreal outputs for exemplar a; e, i, m, q - Unity outputs for exemplar b; f, j, n, r - Unreal outputs for exemplar b.

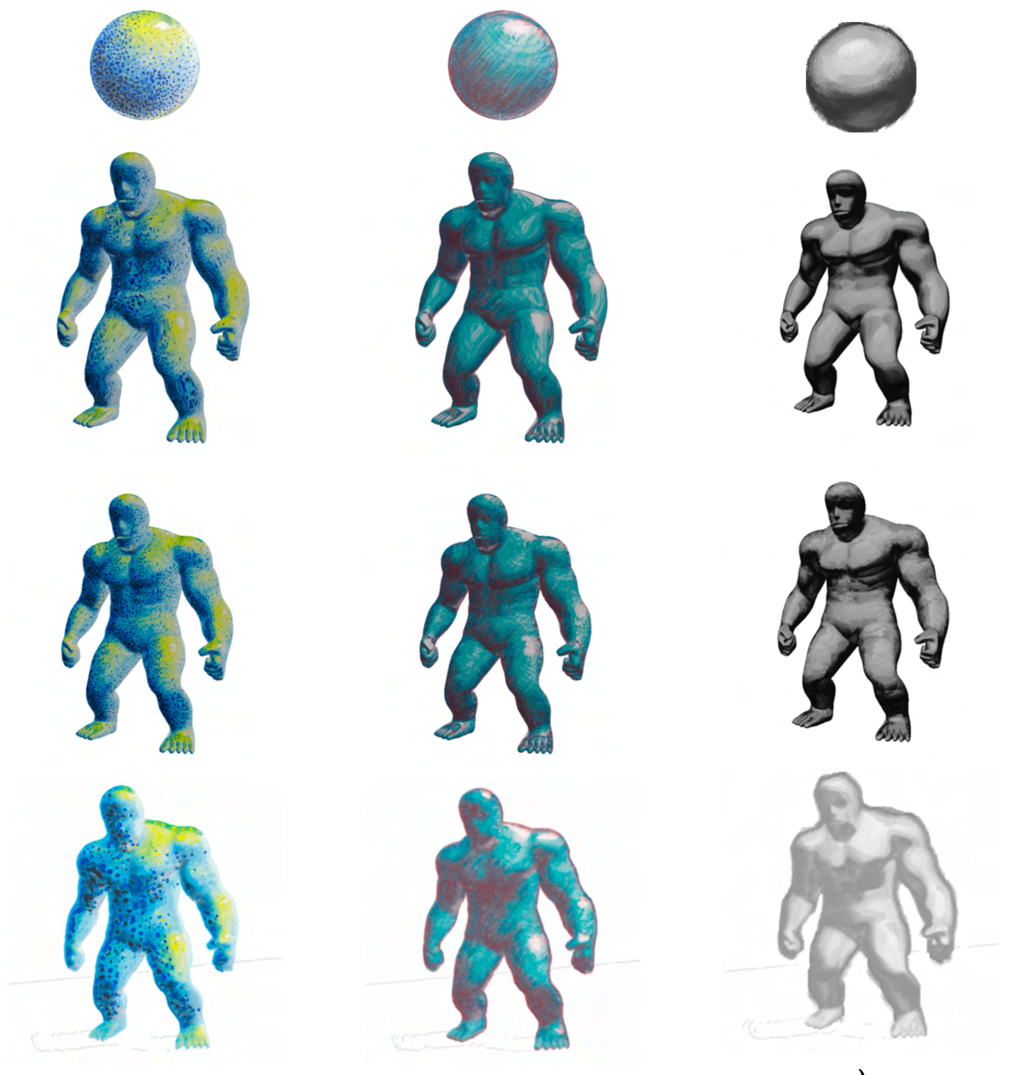


Figure 36 Comparison of stylization outputs for exemplars from the original StyleBlit article: first row - exemplars, 2nd row - the Lit Sphere, 3rd row - StyleBlit, last row - StyLit (colors are slightly off due to color space change). The Lit Sphere performs well on exemplar *c* as it does not feature dominant high-frequency details. StyLit looses geometry details, but is able to capture context features such as blurred borders. StyleBlit correctly captures the scene geometry and high-frequency style features.

4.2 Plugin Usability and Performance

There are two major hurdles the developer looking to use advanced stylization algorithms in their projects often face: the algorithm performance toll and its integration in the development workflow. In this section these aspects of the proposed StyleBlit implementation for Unreal Engine are analyzed.

The usability of the plugin is discussed as well as the prospects for its improvement from the perspective of a developer incorporating the plugin into a project. This discussion stems from the personal experience of the demonstration project creation and thus may not be sufficiently exhaustive. It may be beneficial for further plugin improvement to perform a more formalized assessment of the dedicated UI to better understand the weaknesses for the current implementation.

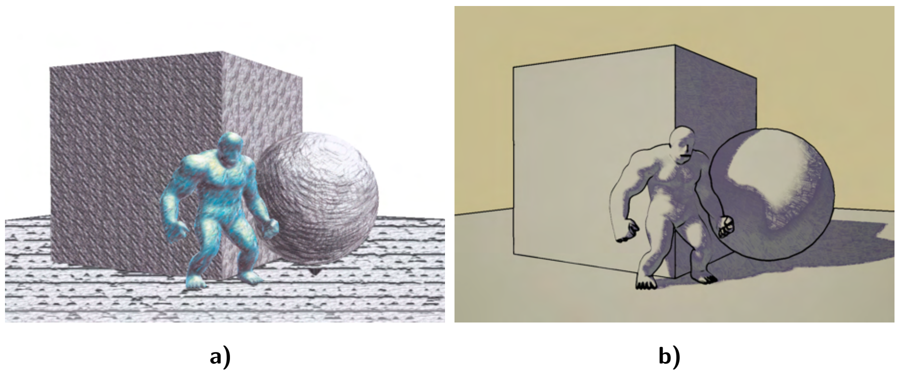


Figure 37 Comparison of (*a*) StyleBlit and (*b*) textured cel-shading approaches to stylization of a simple scene with a variety of geometric objects. The cel-shaded approach outputs more consistent results on all objects in the scene, namely, it is able to handle cast shadows and flat surfaces. However, it is difficult to simulate any other visual style with it.

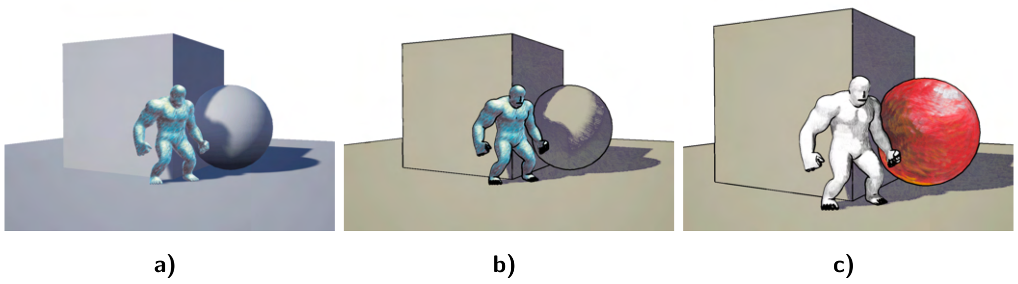


Figure 38 Demonstration of StyleBlit in combination with (*a*) realistic rendering, and (*b*, *c*) cel-shading.

As described in the previous chapter, the plugin contains two modules: StyleBlit and StyleBlitEditor. The former handles the in-game logic related to StyleBlit (i.e. temporal jitter and parameter texture update), the latter handles the logic of editing the StyleBlit parameters in the editor. The in-game module is intended to be as concise as possible. It exposes the required interface to drive the stylization parameters by external game logic and performs updates of the parameters texture every tick (i.e. Unreal loose equivalent of a frame) if the corresponding option is enabled. The initial setup of the resources (i.e. exemplars and guide channel texture objects) and actor states required for StyleBlit to run is delegated to the editor part of the plugin.

The StyleBlit editor interface is contained in a component which can be added to a StyleBlit actor and performs the required resource management actions. The individual exemplar bulks (i.e. sets of exemplar textures and associated stylization parameters) are stored inside an array and can be managed by the user using the common dedicated Unreal interface. The interface may be difficult to navigate and the depth of the hierarchical view makes it harder to operate on multiple exemplars at once. Some of the asset actions are not automated due to the lack of an easily accessible programming interface for those actions in Unreal. A prime example of this is the necessity to manually save the generated texture array before cooking the game, as the automation script is unable to perform this trivial action for the user.

The scene actors can be assigned a specified exemplar bulk using the common eye dropper interface. This triggers a script that changes the custom scene depth value of the selected actor which is then used by the post process material. This interface suffers

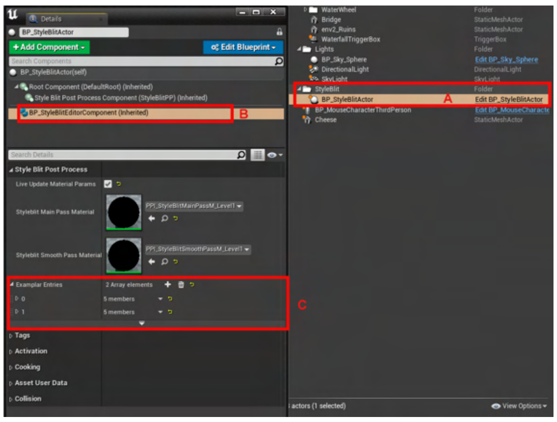


Figure 39 StyleBlit editor component in Unreal: *a* - StyleBlit actor in the level outliner, *b* - editor component added to the StyleBlit actor, *c* - the editable list of exemplar bulks.

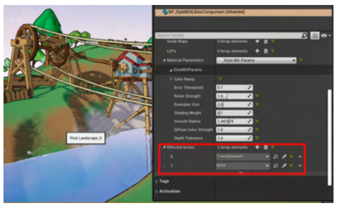


Figure 40 Assignment of an exemplar bulk to an actor via the eye-dropper interface.

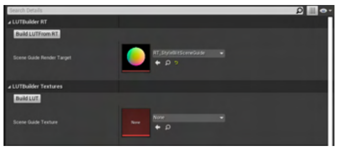


Figure 41 Interface for LUT generation.

from the same flaw of the common object details user interface and requires the user to perform a lot of scrolling in the property list when working on multiple exemplar bulks. Additionally, the current solution does not reliably handle operation undos. A potentially better approach to this user interface aspect would be to contain a reference to the exemplar bulks in the scene actors themselves. This would reduce the number of fields in the array and make it more intuitive to edit the stylization parameters for an actor inside the actor settings window.

The scene guide generation component is also a part of the editor side of the plugin. It contains functions for LUT generation based on a supplied texture or render target. The result of the calculations is saved on disk automatically. The associated user interface is fairly straight-forward, but the process of LUT creation involves a lot of trial and error, which means that many files are generated and it can be time consuming to sort the files afterwards.

The overall state of the StyleBlit editor user interface is a raw prototype. It has a number of bugs and some trivial actions are not automated. The plugin can be improved to enhance the user experience and help more users utilize its features.

Another important aspect of the implementation is the computational cost of the stylization. While many image processing algorithms may perform in real time in minimal demonstration cases, they can still be too expensive to use in a real-world project where the computational budget is split between multiple operations. The performance of the Unreal StyleBlit implementation is tested on the proposed levels included in the demonstration project.

The empirical assessment of the implemented algorithm's computational cost is complicated by its dependency on the configuration of the stylized scene and the specific parameters. When applied on the scene with no visible objects to stylize the algorithm performs no additional computations and the added complexity is equal to the cost of passing the pixel values through a stencil test. The computational complexity grows with the portion of the screen covered by the stylized objects. Additionally the StyleBlit parameters such as the blending radius also influence the number of operations and memory reads and writes the algorithm performs during the stylization. Furthermore, the computational time can be affected by the surrounding Unreal Engine rendering pipeline architecture. For instance adding a post processing stage to the pipeline requires it to perform synchronization and has a computational cost even for a simple pass-through post process material that does not perform any additional operations. For this reason the assessment of the implementation performance was executed as follows.

A user performed a series of roughly equivalent actions in the game and the frame time statistics were recorded using the built-in Unreal Engine profiling tools. This was executed for different level configurations to demonstrate the effect each of the changed parameters has on the overall performance. The selected sequence of actions the player had to perform was rather short (around 40s) for simplicity of the subsequent data analysis and due to the relatively small size of the tested level. During that time the player performed camera movements to capture the stylization performance for different ratios of the screen space covered by the stylized objects. The player also interacted with the level's logic and moved around the level to measure the performance under different occlusion queries results.

The sequence of actions was repeated for the playable level with different combinations of enabled post process materials. The performance was measured in Unreal Engine 4.27.2 in FullHD resolution 1920x1080 on a laptop PC with the following specifications: OS: Windows 10, CPU: GenuineIntel Intel(R), Core(TM) i5-7300HQ CPU @ 2.50GHz, GPU: NVIDIA GeForce GTX 950M. The results are listed figures 42 and 43. Unreal executes the majority of the computations asynchronously (e.g. the CPU-GPU communication is executed in the render thread) and allows to track the performance of different threads separately. The time a frame took to finish then roughly corresponds to the maximum of the individual threads' frame times with additional latency for cross-thread communication. In the case of the tested project the most significant contributor to the frame time by a large margin was rendering (i.e. GPU time), thus the results of the measurement for other threads were left out of the visualizations. The total frame time measured by the Unreal profiling tools does not include the implicit delay caused by the synchronization between CPU and GPU. For this reason the listed total frame time may in some cases be displayed as lower than the GPU time.

The complexity of a post process material is strongly related to the number of arithmetical operations per pixel and number of shared memory accesses by all the GPU threads, however, high branching of the executed shader program may lead to low shader performance with relatively few total operations per pixel. The combination of StyleBlit and cel-shading post process materials proves to be the most computationally complex. The cost of applying both stylization approaches is more than the sum of StyleBlit and cel-shading applied separately due to the price of adding more post processing stages to the rendering pipeline. The StyleBlit itself proved to be computationally cheap in the test, however, the parameters used during the test did not describe

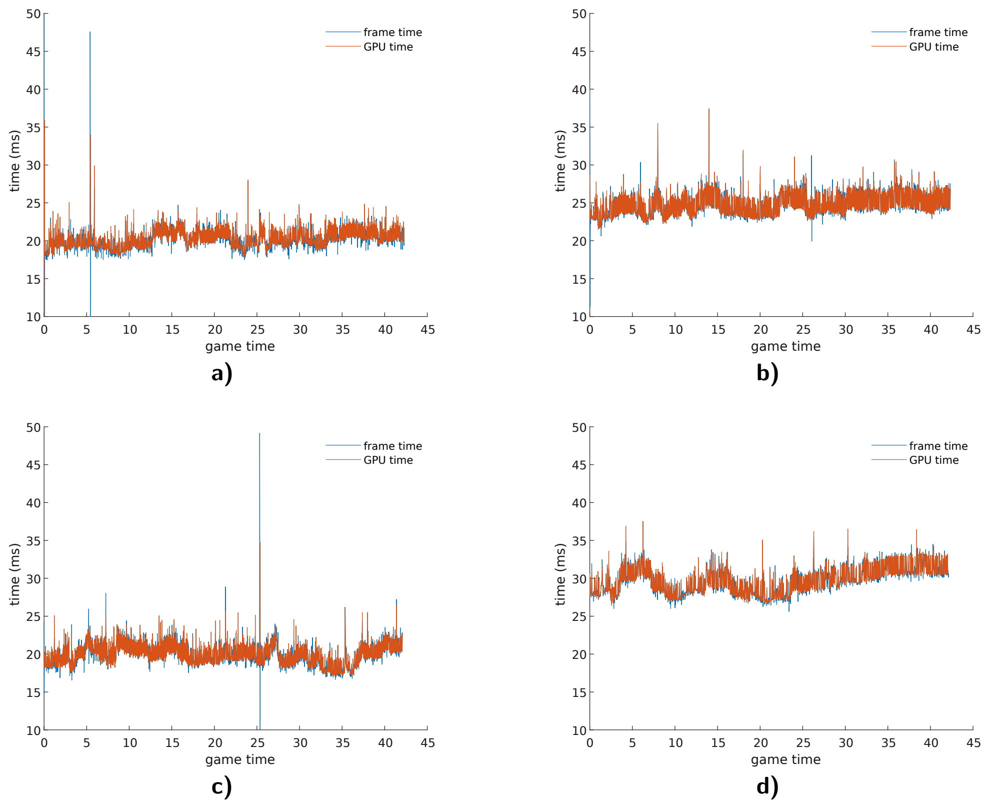


Figure 42 Frame time measured over a small duration (40 s) of the game play on the demonstrational level with different stylization configurations: *a* - no stylization (post process materials are not enabled), average frame time: 19.97 ms, *b* - only cel-shading (i. e. toon shading and outline) are enabled, average frame time: 24.71 ms, *c* - only StyleBlit enabled, average frame time: 20.15 ms, *d* - StyleBlit and cel-chading enabled, average frame time: 29.7 ms.

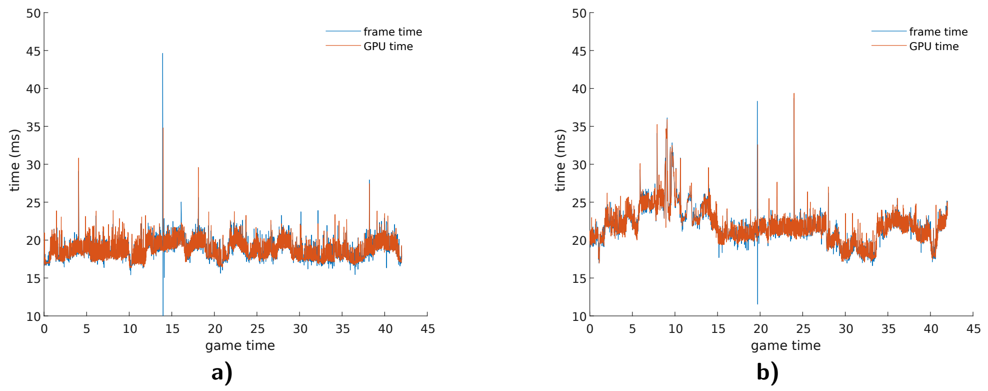


Figure 43 Frame time for different blending radii: *a* - $r = 0$, effectively no blending, the frame time deeps slightly below the reference test with no post processing due to inconsistencies in the player actions, average frame time: 18.93 ms, *b* - $r = 4$, significant blending on all stylized objects, the performance oscillates significantly with view position relative to the stylized objects, average frame time: 22.1 ms.

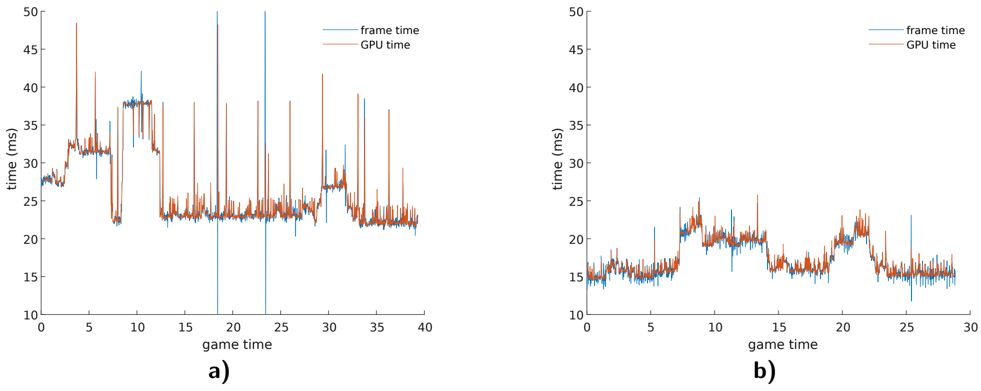


Figure 44 Frame time for showcase level: *a* - both StyleBlit and outline are enabled, average frame time: 26.25 ms, *b* - only StyleBlit is enabled, average frame time: 17.26 ms. The plateaus in the first half of the graphs corresponds to the user changing the blending radius from 2 to 5 and back, the smaller plateaus in the later half of the run correspond to camera zooming in to the object.

the most computationally difficult set up (a small blending radius was used). The figure 43 demonstrates the cost of StyleBlit with different blending radii. The blending radius influences the performance significantly.

The same test was executed on the showcase level to compare the performance of the algorithm with the measurements in the previous test. The number of actions the player can make was smaller, thus the repeated sequences of actions were slightly shorter (around 30 s). Aside from the camera manipulation the player controlled the stylization parameters. The resulting measurements contain obvious frame time plateaus and valleys that are formed by the portion of the screen occupied by the rendered objects, and the blending radius parameter value. The results are visualized in figure 44. The outline applied for the level requires more operations per pixel due to the larger line width setting selected, thus the performance impact of the shader is more significant than in the playable level.

The implemented StyleBlit post process material is able to stably perform virtual scene stylization in real time for a wide variety of setups and configurations. It is not too computationally expensive and when used correctly can be efficiently combined with other post process materials and integrated into practical projects. The cel-shading implementation that supplements StyleBlit in the plugin performs a number of computationally heavy operations and texture reads and could be optimized to achieve a more stable performance.

5 Conclusion

In this thesis the process of virtual scene stylization was explored. The background for exemplar-driven stylization algorithms was established and several existing exemplar-based stylization approaches for virtual scenes were overviewed. The proposed discussion was aimed to provide a sufficient basis and terminology to perform an in-depth analysis of the main focus of the thesis: StyleBlit stylization algorithm.

StyleBlit was described based on the original article that proposed the algorithm and the introductory section's discussion. The description contains a detailed look over the algorithm's individual steps and required data structures. The algorithm was stated to contain two general stages: initial style transfer and chunk border blending. The contribution of each stage to the final result was analyzed. Additionally, the nearest neighbor search used during the algorithm's initial style transfer was dissected and analyzed for its specifications' impact on the output result in different usage scenarios. The StyleBlit strengths and weaknesses were summarized. The algorithm was stated to work best for stylization of organic rounded shapes and to be difficult to effectively apply to flat surfaces or environments due to its structure.

For the sake of the algorithm's subsequent implementation the existing implementations were explored. The focus of the implementations' analysis was the integration of StyleBlit into a modern rasterization pipeline The OpenGL implementation supplied as with the original StyleBlit article was stated to exemplify the analytical solution of the nearest neighbor search problem solution. The exemplar texture files from the application were used to evaluate the results of the thesis implementation. The Unity StyleBlit plugin was analyzed as an example of the lookup texture nearest neighbor solution usage. It was also inspected for the possible ways StyleBlit can be used as a modular solution in a game engine.

The results of the analysis were then used to create an Unreal Engine plugin containing the reusable StyleBlit implementation along with other supplementary features. For this a review of the Unreal rendering pipeline and general asset structuring concepts was given. The different options of integrating StyleBlit into the engine were explored and the final solution was proposed: the algorithm was implemented as a number of post-process materials which handle the corresponding stylization stages. Additionally several StyleBlit extensions were proposed to widen the range of possible use cases of the technique. The extensions enable the algorithm to utilize information about the stylized objects color and shading during the style transfer.

Along with the StyleBlit implementation itself supplementary editor utilities to handle asset manipulations were included in the developed plugin. Two alternative stylization approaches were also implemented and included in the plugin. These approaches are outline rendering and toon shading, they were briefly studied and demonstrated in the corresponding sections of the thesis. The additional stylization methods can be used together with StyleBlit to stylize a wider range of virtual scenes.

In the latter part of the thesis the implementation results were evaluated. The outputs of the Unreal plugin solution were compared with those of the existing StyleBlit implementations. The results were assessed to be sufficiently similar to the reference solutions. Further the visual qualities of the StyleBlit outputs were compared with other stylization approaches. The algorithm was proposed to work best when used in combination with other methods to output the best looking results.

The suggested plugin was also evaluated from a developer's standpoint, assessing how usable the proposed interface for StyleBlit management is in a practical scenario. The evaluation yielded several possibilities for the interface improvement, but it was stated to be non exhaustive and further testing could be beneficial for further plugin development. Finally, the performance of different implementation aspects was tested in a game scenario. The implementation performed reasonably well to be used in a practical setting. Some room for improvement was found for the outline and toon shading implementations.

Bibliography

[1] Jan Eric Kyprianidis et al. “State of the ”Art”: A Taxonomy of Artistic Stylization Techniques for Images and Video”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.5 (2013), pp. 866–885. DOI: 10.1109/TVCG.2012.160. URL: https://hal.inria.fr/hal-00781502.

[2] Daniel Sýkora et al. “StyleBlit: Fast Example-Based Stylization with Local Guidance”. In: *Computer Graphics Forum* 38.2 (2019), pp. 83–91.

[3] Aneta Texler et al. “FaceBlit: Instant Real-time Example-based Style Transfer to Facial Videos”. In: *Proceedings of the ACM in Computer Graphics and Interactive Techniques* 4.1 (2021), p. 14.

[4] Holger Winnemöller, Sven C. Olsen, and Bruce Gooch. “Real-Time Video Abstraction”. In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 1221–1226. ISSN: 0730-0301. DOI: 10.1145/1141911.1142018. URL: https://doi.org/10.1145/1141911.1142018.

[5] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. “Image Style Transfer Using Convolutional Neural Networks”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2414–2423. DOI: 10.1109/CVPR.2016.265.

[6] Yongcheng Jing et al. “Neural Style Transfer: A Review”. In: *CoRR* abs/1705.04058 (2017). arXiv: 1705.04058. URL: http://arxiv.org/abs/1705.04058.

[7] Aaron Hertzmann et al. “Image Analogies”. In: *Proceedings of ACM SIGGRAPH* 2001 (June 2001). DOI: 10.1145/383259.383295.

[8] Peter-Pike Sloan et al. “The Lit Sphere: A Model for Capturing NPR Shading from Art”. In: *Proceedings of the Graphics Interface 2001 Conference, June 7-9 2001, Ottawa, Ontario, Canada*. June 2001, pp. 143–150. URL: http://graphicsinterface.org/wp-content/uploads/gi2001-17.pdf.

[9] Carlos Zubiaga et al. “MatCap Decomposition for Dynamic Appearance Manipulation”. In: (June 2015).

[10] Jakub Fišer et al. “StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings”. In: *ACM Transactions on Graphics* 35.4 (2016).

[11] Ondřej Jamriška et al. “LazyFluids: Appearance Transfer for Fluid Animations”. In: *ACM Transactions on Graphics* 34.4 (2015).

[12] Yangdong Deng et al. “Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques”. In: *ACM Comput. Surv.* 50.4 (Aug. 2017). ISSN: 0360-0300. DOI: 10.1145/3104067. URL: https://doi.org/10.1145/3104067.

[13] Scott Schaefer, Travis McPhail, and Joe Warren. “Image Deformation Using Moving Least Squares”. In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 533–540. ISSN: 0730-0301. DOI: 10.1145/1141911.1141920. URL: https://doi.org/10.1145/1141911.1141920.

[14] Buyue Zhang and Jan P. Allebach. "Adaptive Bilateral Filter for Sharpness Enhancement and Noise Removal". In: *IEEE Transactions on Image Processing* 17.5 (2008), pp. 664–678. DOI: 10.1109/TIP.2008.919949.

[15] Bruce Lucas and Takeo Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI)". In: vol. 81. Apr. 1981.

[16] Alexandru-Lucian Petrescu et al. "Analyzing Deferred Rendering Techniques". In: *Control Engineering and Applied Informatics* 18 (Mar. 2016), pp. 30–41.

[17] *Unity Manual: Rendering paths in the Built-in Render Pipeline.* 2021. URL: https://docs.unity3d.com/Manual/RenderingPaths.html (visited on 01/09/2022).

[18] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[19] *Nanite Virtualized Geometry: Overview of Unreal Engine 5's virtualized geometry system to achieve pixel scale detail and high object counts.* 2022. URL: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/ (visited on 08/05/2022).

[20] *Render Dependency Graph: An immediate-mode API which records render commands into a graph data structure to be compiled and executed.* 2021. URL: https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/ (visited on 08/05/2022).

[21] Kostas Anagnostou. *INTERPLAY OF LIGHT HOW UNREAL RENDERS A FRAME.* 2017. URL: https://interplayoflight.wordpress.com/2017/10/25/how-unreal-renders-a-frame/ (visited on 08/05/2022).

[22] Morgan McGuire, Michael Mara, and David Luebke. "Scalable Ambient Obscurance". In: *Proceedings of ACM SIGGRAPH / Eurographics High-Performance Graphics 2012 (HPG '12)* (June 2012). High-Performance Graphics 2012. URL: https://casual-effects.com/research/McGuire2012SAO/index.html.

[23] William T. Reeves, David H. Salesin, and Robert L. Cook. "Rendering Antialiased Shadows with Depth Maps". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 283–291. ISBN: 0897912276. DOI: 10.1145/37401.37435. URL: https://doi.org/10.1145/37401.37435.

[24] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered deferred and forward shading". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics.* Citeseer. 2012, pp. 87–96.

[25] Artem Kovalovs. "Volumetric Effects of The Last of Us: Part Two". In: *ACM SIGGRAPH 2020 Talks.* SIGGRAPH '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450379717. DOI: 10.1145/3388767.3407393. URL: https://doi.org/10.1145/3388767.3407393.

[26] Austin Robison and Peter Shirley. "Image Space Gathering". In: *Proceedings of the Conference on High Performance Graphics 2009.* HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 91–98. ISBN: 9781605586038. DOI: 10.1145/1572769.1572784. URL: https://doi.org/10.1145/1572769.1572784.

[27] Eric Bruneton and Fabrice Neyret. “Precomputed Atmospheric Scattering”. In: *Computer Graphics Forum.* Special Issue: Proceedings of the 19th Eurographics Symposium on Rendering 2008 27.4 (June 2008), pp. 1079–1086. DOI: 10.1111/j.1467-8659.2008.01245.x. URL: https://hal.inria.fr/inria-00288758.

[28] Hoshang Kolivand, Mohd Shahrizal Sunar, and Ali Selamat. “Real-Time Light Shaft Generation for Indoor Rendering”. In: *Intelligent Software Methodologies, Tools and Techniques.* Ed. by Hamido Fujita and Guido Guizzi. Cham: Springer International Publishing, 2015, pp. 487–495. ISBN: 978-3-319-22689-7.

[29] gishi523. *kd-tree: A c++ implementation of k-d tree.* 2017. URL: https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/ (visited on 08/06/2022).

[30] Vlastimil Havran. “Heuristic Ray Shooting Algorithms”. Ph.D. Thesis. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov. 2000. URL: http://www.cgg.cvut.cz/~havran/phdthesis.html.

[31] Nathan Reed. *NVIDIA DEVELOPER: Depth Precision Visualized.* 2015. URL: https://developer.nvidia.com/content/depth-precision-visualized (visited on 08/06/2022).

[32] James T. Todd. “The visual perception of 3D shape”. In: *Trends in Cognitive Sciences* 8.3 (2004), pp. 115–121. ISSN: 1364-6613. DOI: https://doi.org/10.1016/j.tics.2004.01.006. URL: https://www.sciencedirect.com/science/article/pii/S1364661304000233.

[33] Linjia Hu, Saeid Nooshabadi, and Majid Ahmadi. “Massively parallel KD-tree construction and nearest neighbor search algorithms”. In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS).* 2015, pp. 2752–2755. DOI: 10.1109/ISCAS.2015.7169256.

[34] Yasir Ali et al. “Tone mapping of HDR images: A review”. In: June 2012, pp. 368–373. ISBN: 978-1-4577-1967-7. DOI: 10.1109/ICIAS.2012.6306220.

[35] Amy Gooch et al. “A Non-Photorealistic Lighting Model For Automatic Technical Illustration”. In: *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics* (Sept. 1999). DOI: 10.1145/280814.280950.

[36] K.I. Anjyo and K. Hiramitsu. “Stylized highlights for cartoon rendering and animation”. In: *IEEE Computer Graphics and Applications* 23.4 (2003), pp. 54–61. DOI: 10.1109/MCG.2003.1210865.

[37] Pascal Barla, Joëlle Thollot, and Lee Markosian. “X-Toon: An Extended Toon Shader”. In: *NPAR2006.* Ed. by Doug DeCarlo and Lee Markosian. New York: ACM Press, 2006, pp. 127–132. DOI: http://doi.acm.org/10.1145/1124728.1124749. URL: http://artis.inrialpes.fr/Publications/2006/BTM06a/.

[38] Michael P. Salisbury et al. “Orientable Textures for Image-Based Pen-and-Ink Illustration”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH ’97. USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 401–406. ISBN: 0897918967. DOI: 10.1145/258734.258890. URL: https://doi.org/10.1145/258734.258890.

[39] Doug DeCarlo et al. “Suggestive Contours for Conveying Shape”. In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 848–855. ISSN: 0730-0301. DOI: 10.1145/882262.882354. URL: https://doi.org/10.1145/882262.882354.

[40] Forrester H Cole. "Line drawings of 3D models". In: (2009).

[41] D. Kang, D. Kim, and K. Yoon. "A study on the real-time toon rendering for 3D geometry model". In: *Proceedings Fifth International Conference on Information Visualisation.* 2001, pp. 391–396. DOI: `10.1109/IV.2001.942087`.

[42] Yunjin Lee et al. "Line Drawings via Abstracted Shading". In: *ACM Trans. Graph.* 26.3 (July 2007), 18–es. ISSN: 0730-0301. DOI: `10.1145/1276377.1276400`. URL: `https://doi.org/10.1145/1276377.1276400`.

[43] Ben Golus. *The Quest for Very Wide Outlines: An Exploration of GPU Silhouette Rendering.* 2020. URL: `https://bgolus.medium.com/the-quest-for-very-wide-outlines-ba82ed442cd9` (visited on 08/11/2022).

[44] Guodong Rong and Tiow-Seng Tan. "Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform". In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games.* I3D '06. Redwood City, California: Association for Computing Machinery, 2006, pp. 109–116. ISBN: 159593295X. DOI: `10.1145/1111411.1111431`. URL: `https://doi.org/10.1145/1111411.1111431`.

[45] Andreas Aristidou and Joan Lasenby. "FABRIK: A fast, iterative solver for the Inverse Kinematics problem". In: *Graphical Models* 73.5 (2011), pp. 243–260. ISSN: 1524-0703. DOI: `https://doi.org/10.1016/j.gmod.2011.05.003`. URL: `https://www.sciencedirect.com/science/article/pii/S1524070311000178`.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Isaiev Mykola**	Personal ID number:	**466358**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Graphics**		

II. Master’s thesis details

Master’s thesis title in English:

Example-based Stylization of 3D Models Using Unreal Engine

Master’s thesis title in Czech:

Přenos výtvarného stylu na 3D model s využitím Unreal Engine

Guidelines:

Explore algorithms for transferring style from a hand-drawn exemplar of a simple scene to a more complex 3D model [1, 2, 3]. Implement the StyleBlit [3] algorithm in the Unreal Engine using a set of specialized shaders and multipass rendering to allow the stylization of even more complex scenes with several styles in real-time. Compare the results of the resulting implementation with the outputs presented in the original paper [3]. Evaluate the quality of the resultant style transfer and its computational overhead. Try to design a simple computer game that will further validate the practical applicability of the implemented algorithm.

Bibliography / sources:

[1] Sloan et al.: The Lit Sphere: A Model for Capturing NPR Shading from Art, Proceedings of Graphics Interface, pp. 143-150, 2001.
[2] Fišer et al.: StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings, ACM Transactions on Graphics 35(4):92, 2016.
[3] Sýkora et al.: StyleBlit: Fast Example-Based Stylization with Local Guidance, Computer Graphics Forum 38(2):83–91, 2019.

Name and workplace of master’s thesis supervisor:

prof. Ing. Daniel Sýkora, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **06.02.2022** Deadline for master's thesis submission: **15.08.2022**

Assignment valid until: **30.09.2023**

prof. Ing. Daniel Sýkora, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature