# Light Graffiti video effect optimisation

*Simon A. Eugster*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

## ABSTRACT

This paper describes optimisation steps for the *Light Graffiti* video effect which simulates long-time exposure in post-processing. Faster processing of video frames allows more fluent video editing and rendering; finally after optimising a maximum speedup of $10\times$ is achieved.

## 1. INTRODUCTION

*This section explains what Light Graffiti is and discusses previous work.*

**Motivation.** *Light Graffiti*, or *Light Painting*, is a term commonly used for «painting» with light sources while photographing with long-time exposure: The sensor accumulates the light from the light source, and the patterns drawn stay on the final picture.

For video this approach does not work since the frame rate gives a hard lower bound for the shutter speed: While a photograph can be exposed for several minutes, allowing the light painting artist to draw whole landscapes, a video with a frame rate of 24 fps, as used in cinema, can expose each frame for at most $\frac{1}{24}s$ – so the only option would be to do it in post.

As of 2011 I found a video[1] that did just this in hardware, and another guy who used an additional camera with long-time exposure (i.e. the photograph served as an overlay that was uncovered by and by)[2].

I then started writing my own Light Graffiti effect, which was the first publicly available video effect (and still is, as of today).

The effect is rather slow to use while editing since it involves expensive computations. The goal therefore is to make it run faster to allow more fluent editing.

**Related work.** The original implementation of the Light Graffiti algorithm is described on [3] and is available via `http://code.dyne.org/frei0r/`.

The optimised version of the effect is up to 10 times faster. The optimisations will be described in this paper.

## 2. BACKGROUND: FUNCTIONALITY

*This section describes how the Light Graffiti algorithm works, and explains what the difficulties are.*

Extracting light from an image sounds easy: Light is bright, so we can simply use a threshold to decide between light and the rest of the image $(3 \cdot 255 - (r + g + b) < k)$.[1]

This works fine for dark environments – which usually they are not! On the street by night there are street lights in the background, or another object in the image may look close to white for the camera. So the sensitivity would have to be decreased (to e.g. $k = 10$) in order to not match those.

Then, however, coloured lights, like a light blue one with $r = 181, g = 220, b = 251$, fail to be detected with $765 - (r + g + b) = 115$.

My algorithm additionally calculates the differences $\Delta r$, $\Delta g$, $\Delta b$ to the unlighted background, similar to a temporal derivative. Additional thresholds for

$$\text{diffSum} = \sum (\Delta r, \Delta g, \Delta b) \tag{1}$$

$$\text{diffMax} = \max(\Delta r, \Delta g, \Delta b) \tag{2}$$

allow to precisely detect light.

The light itself is stored in a `float` light mask which can be dimmed over time linearly or with a sine function (which looks more natural). It allows values much greater than 1 (the `uchar` pixel values get normalized to $[0 \ldots 1]$ before they are added to the light mask) to simulate overexposure: A light source staying in place for some time leaves a bright spot that does not faint immediately but only after a while. The overexposure reduction option allows to avoid extreme values by applying a log function.

Finally the saturation of lights can be increased when they are painted back on the input image as coloured lights tend to lose their colour when fading.

[1]Colours for red, green, and blue lie in the range $\{0 \ldots 255\}$.

```
process(image) :
for all $pixel \in image$ do
    $mean \leftarrow$ update($mean, image$)
    $lightmask \leftarrow$ dim($lightmask$)
    if isLight($pixel, mean$) then
        $mask \leftarrow mask +$ getLight($pixel, mean$)
    end if
    if $mask \neq 0$ then
        $pixel \leftarrow pixel +$ lowerOexp($mask$)
        $pixel \leftarrow$ saturate($pixel$)
    end if
end for
```

**Cost Analysis.** I'm considering two types of cost measures for the LightGraffiti code. The first one comes from optimisation with the goal to run the program as performant as possible; This is measured in Flops[2] per cycle (Performance) and, for memory traffic, Flops per Byte transferred between CPU and RAM (Operational Intensity). The second one is runtime which is of importance for the user as editing should be as close to realtime as possible.

As the algorithm pseudo-code shows, the cost raises if light is detected in the image and with increasing coverage of the light mask.

I counted the flops and cycles with Intel's VTune.

The complexity of the problem, as solved by this algorithm, is $\mathcal{O}(n)$.

## 3. OPTIMISATIONS

*This section will discuss the important optimisation steps. Small ones which did not have a big effect or were already implemented are not described here.*

The original algorithm is abbreviated as OA, the different optimisation steps numbered from O1 to O7.

**Single Loop (O1–O4).** The original algorithm used two additional loops just for updating the mean image and dimming the light mask. This avoided branching misprediction at the cost of additional memory transfer – which was more expensive. The size of both mask and mean image is, for a FullHD frame of $1920 \times 1080$px, 24 MB.[3] As this is far larger than the L3 cache,[4] they have to be read from and written back to RAM, which gives a memory transfer overhead of 96 MB per frame, with up to 96 MB transferred between RAM and CPU in the main loop.[5]

With the single loop at most 120 MB will be transferred; less is not possible. The change in the Roofline Model (Fig-

[2]Floating-Point Operations

[3]For each pixel three floats with 4 Bytes each need to be stored.

[4]On the i7-2620M the L3 cache has a size of 4 MB.

[5]24 MB for reading the input and writing the output image, plus reading mean and light mask and possibly updating the light mask.

```
dim(lightmask) :
if $linear$ then
    $light \leftarrow light * (1 - x_0)$
else
    $light \leftarrow$ pow(sin($light$), $x_0$)
end if
saturate(pixel)
$pixel \leftarrow$ pow(log($pixel$)$/x_1, x_2$)
```

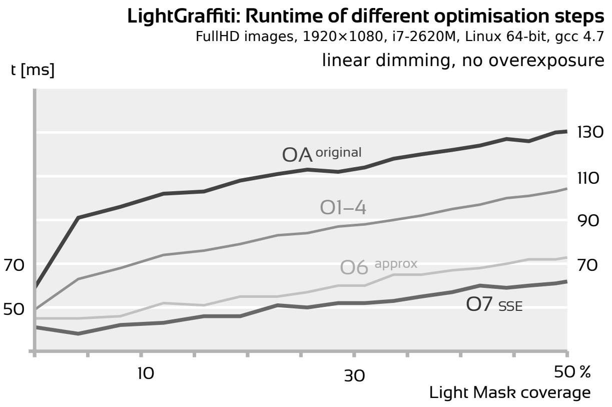

**Fig. 1**. Runtime for the LightGraffiti effect with linear dimming and with disabled overexposure reduction. These settings do not use any pow/log functions and have around 60 flops per pixel.

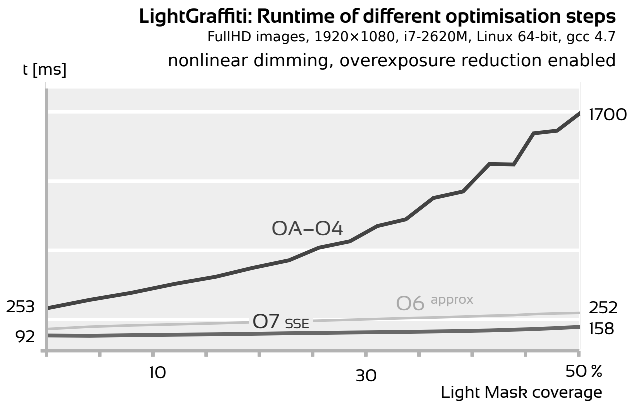

**Fig. 2**. Same measurement with nonlinear dimming and overexposure reduction. This adds approximately 3 sin and log functions (each) and 6 pow functions per pixel.

ure 3) indicates this as well (less Bytes per Flop), and the runtime improved (Figure 1).

**Approximative Functions (O6).** So far the code used a total of about 60 additions and multiplications per pixel. Enabling nonlinear dimming of the light mask and overexposure reduction added 3 sine, 6 power, and 3 log functions – and increased the runtime of the effect 15 times, when comparing the plots for O1 in figures 1 and 2. It is highly undesirable for a video effect to take 1.7 seconds for processing a single frame; Although the effect is (so far) applied in post-production only, this is nevertheless far too long for comfortable editing.

Perfect vectorisation would only decrease the runtime by a factor of at most 4 with SSE. Since the videos only have 8-bit precision though, faster (but less accurate) functions gave the same result (except for rounding errors, which were negligible). The most costly function was the power function which took 260 cycles less with the approximate function [4], the logarithm function was 86 cycles faster ([5]).

Oddly I measured +100 ms per frame with the STL[6] log and +450 ms with the STL pow, but +1150 ms (instead of the expected +550 ms) with both STL log and pow! The only possible reason I could imagine is that $\mathrm{pow} \circ \log$ pollutes the L1/L2 caches, causing more expensive L3 access[7] which contributes to the additional 500 ms per frame (144 cycles per pixel).

Manual taylor expansion for the sine function did not give any speedup; it was generally the fastest of all those functions and I stayed with the STL function.

**Vectorisation (O7).** The roofline model clearly shows that the calculation is still compute bound (i.e. not bound by memory traffic) on the i7.[8] For the last optimisation I used 4-fold loop unrolling and vectorized the code, also using approximate (vectorized) functions ([6, 7]). Although using single precision instructions (meaning that I could process 4 floats at a time) the achieved speedup was not even $2\times$. One reason is that I had to replace conditions by computation (compute the result of both branches and then `blend` them together). The other reason is that the vectorized implementations of the power and log functions are not identical.

**Failed attempts.** Since the code uses a lot of branches, I tried replacing them with computations in the scalar case:

```
**if** $a > 1$ **then**
    $a \leftarrow ax_0$
**else**
    $a \leftarrow \sin(a)$
**end if**
```

becomes

$a \leftarrow (a > 1)ax_0 + (1 - a > 1)\sin(a)$

The branch predictor proved to be better however, and the computational overhead was too large.

Re-ordering computations for better ILP[9] was not possible either since the computations depended too much on each other.

For O7 I tried using a struct of vectors instead of a vector of structs, as this can be read more easily by SSE instructions, but required an additional `_MM_TRANSPOSE_PS`; this code did not give any improvements.

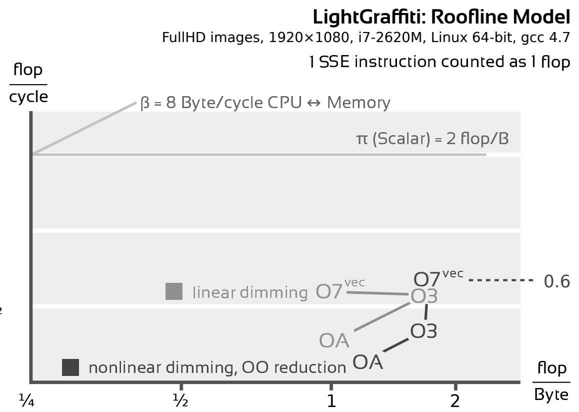

**Fig. 3**. The roofline model indicates that the effect is in the compute bound region, as the Sandy Bridge architecture can transfer 8 Bytes per cycle between RAM and CPU. O1 to O4 have been collected in the point O3 since their positions did not mentionably differ.

## 4. EXPERIMENTAL RESULTS

*This section is so short that a summary would be just as long as the section itself.*

**Experimental setup.** The optimizations were tested and measured on an Intel i7-2620M with disabled SpeedStep and Powersave features to get stable timing results. The L3 cache has 4 MB. The program was compiled with gcc 4.7.0 with the flags `-O3 -std=c++0x -march=corei7 -mavx -g -fno-tree-vectorize`.

As input I used generated FullHD images ($1920 \times 1080$ px) with the light mask covering between 0 and 50 % of the image (this affects the amount of computation required).

**Results.** In the linear case (see Figure 1) the speedup regarding runtime is constant at $2\times$, for nonlinear dimming and overexposure reduction (Figure 2) it raises up to $10\times$. It is still not realtime, but better than the original algorithm.

For the linear original algorithm I counted 36 additions and 28 multiplications. In the ideal case, if additions and

[6]Standard Template Library, here from GCC 4.7

[7]L3 cache latency is between 12 and 38 cycles for the Sandy Bridge.

[8]In each cycle 8 Bytes can be transferred between CPU and RAM (if accessed sequentially), which would permit up to 8 Flops per cycle.

[9]Instruction Level Parallelism

multiplications always appear alternately (and therefore can be executed in parallel), the (theoretical) peak performance is 1.55 flops per cycle. For the linear case OA reaches 24 %, O3 reaches 35 %, and O7 reaches 27 % of peak.

The roofline model (Figure 3) shows an inconsistency: The nonlinear O3 should be more to the right than O7 since it still uses the slow STL functions. Since I have measured the floating-point operations with VTune and they did not differ between the linear and the nonlinear case in O3, I assume that the STL functions use other operations (i.e. not floating-point) which are not reflected in this model. The model also shows that there is still room for improvement for O7.

## 5. CONCLUSIONS

This video effect was not a typical optimisation problem since, first, runtime was the most important criterion and, secondly, the biggest effect was the result of using the approximate functions.

The roofline model suggests that after those changes some classical optimisation can still be possible; i.e. for example checking for data hazards to get better ILP.

Important to remember is that especially the power function takes hundreds of cycles (Intel's ICC is said to be better for those functions, but the video effect did not compile with it). Approximate functions cannot be used everywhere due to their limited precision, but 8-bit video processing seems like a good place for them.

## 6. REFERENCES

[1] Tak+Pipslab, "Light painting – light graffiti – ford kuga (table)," http://www.youtube.com/watch?v=WVaxuIKPKvU, 2008.

[2] MARKO93, "Paris by light (legal lights graffiti)," http://www.youtube.com/watch?v=MZf2W3S7gV0, 2007.

[3] Simon A. Eugster, "Writing a light graffiti effect ...," http://kdenlive.org/users/granjow/writing-light-graffiti-effect, 2011.

[4] Martin Ankerl, "Optimized approximative pow() in c / c++," http://martin.ankerl.com/2012/01/25/optimized-approximative-pow-in-c-and-cpp/, 2012.

[5] Laurent, "Fast log() function," http://www.flipcode.com/archives/Fast_log_Function.shtml.

[6] José Fonseca, "Fast sse2 pow: tables or polynomials?," http://jrfonseca.blogspot.ch/2008/09/fast-sse2-pow-tables-or-polynomials.html, 2008.

[7] zogzog, "Simple sse and sse2 (and now neon) optimized sin, cos, log and exp," http://gruntthepeon.free.fr/ssemath/.