BIWI-TR-269 **August 11, 2005**

Real-Time, GPU-based Foreground-Background Segmentation

Andreas Griesser
griesser@vision.ee.ethz.ch
Computer Vision Lab, Swiss Federal Institute of Technology, Zürich

Abstract

This report presents a GPU-based foreground-background segmentation that processes image sequences in less than 4ms per frame. Change detection wrt. the background is based on a color similarity test in a small pixel neighbourhood, and is integrated into a Bayesian estimation framework. An iterative MRF-based model is applied, exploiting parallelism on modern graphics hardware. Resulting segmentation exhibits compactness and smoothness in foreground areas as well as for inter-frame temporal contiguity. Further refinements extend the colinearity criterion with compensation for dark foreground and background areas and thus improving overall performance.

Contents

1 Introduction **3**

2 Mathematical Model **3**
2.1 The Colinearity Criterion . . . 3
2.2 Bayesian Estimation . . . 4
2.3 Adaptive Threshold, MRF . . . 5
2.4 Darkness Compensation . . . 7
2.5 Final Decision Rule . . . 7
2.6 Iterative, randomized MRF computation . . . 9

3 Implementation Issues **10**
3.1 Programming Language . . . 11
3.2 Drawing Context . . . 11
3.3 Offscreen-Buffers . . . 11
3.4 GPU Data Formats . . . 12
3.5 Bilinear Interpolation . . . 12

4 GPU-based Implementation **14**
4.1 The *CalcNorm*-Shader . . . 14
4.2 The *SumNorm*-Shader . . . 15
4.3 The *NormMixBack*-Shader . . . 16
4.4 The *NormConvert*-Shader . . . 16
4.5 The *MRF-Iter*-Shader . . . 18
4.5.1 Center Area . . . 21
4.5.2 Top Area . . . 22
4.5.3 Bottom Area . . . 24
4.5.4 Shader Programs . . . 25
4.6 The *SegConvert*-Shader . . . 26

5 User's Guide **27**
5.1 Requirements . . . 27
5.2 Distributed files . . . 28
5.3 Library class description . . . 29
5.4 Example Program . . . 32

6 Results **33**

1 Introduction

Robust and accurate foreground-background segmentation is a relatively small but crucial step in several computer vision applications. It is a key element in surveillance, 3D-modelling from silhouettes, motion capture, or gesture analysis for human-computer interaction (HCI). For several of these - surveillance and HCI are cases in point - real-time processing is crucial. Hence, for these applications, foreground-background segmentation should be extremely fast, as the bulk of the computation time on the CPU has to remain available for the subsequent stages of processing and interpretation.

As a result, the type of foreground-background segmentation that can be used on-line has typically been kept as simple as possible, and has led to important constraints on the background. For instance, in their semi on-line user modeling work, Matusik *et al.* [8] had to resort to a rather simple background subtraction. On the other hand, more sophisticated algorithms are available today, like Bayesian pixel classification based on time-adaptive, per-pixel mixture of Gaussians color model [4, 5]. A comprehensive survey of image change detection algorithms is presented in [6]. Recently, Mester *et al.* developed a color similarity criterion [1], which has already performed well in our - offline - gesture recognition setup [7]. However, these sophisticated algorithms lack real-time performance.

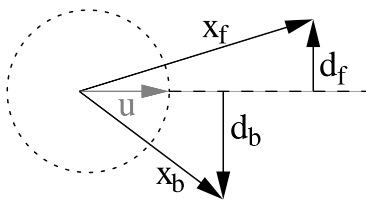
Here we propose a GPU-based implementation of Mester's approach, combined with some refinements to further improve performance. Our implementation takes less than 4 milliseconds per frame and frees the CPU from this preprocessing step altogether. Thus, our approach is especially useful for algorithms already using the GPU in the further processing stages.

2 Mathematical Model

2.1 The Colinearity Criterion

Mester's method compares the color values at pixels in a reference (background) image, and a given image. In particular, all color values within a small window around a pixel, here always a 3×3 neighbourhood, are stacked into row vectors $\mathbf{x}_b$ resp. $\mathbf{x}_f$ for the background resp. the given image, where the latter will typically contain some additional foreground objects.

Under the null hypothesis H_0 these vectors can be written as $\mathbf{x}_b = \mathbf{s} + \epsilon_b$ and $\mathbf{x}_f = k \cdot \mathbf{s} + \epsilon_f$, where ϵ_b and ϵ_f are additive noise vectors and $\mathbf{s}$ is an unknown signal vector. Change detection amounts to assessing whether $\mathbf{x}_b$ and $\mathbf{x}_f$ are colinear. If they are (the null hypothesis H_0), no change is judged to be present and the background is still visible at that pixel in the given image. If not, the pixels are considered to have different colors, and a foreground pixel has been found.



Rather than testing for perfect colinearity, one has to allow for some noise in the measurement process. Indeed, when Gaussian noise is assumed, the unkown 'true signal' direction (represented by the unit vector u) can be estimated by minimizing the sum $D^2 = |\mathbf{d}_b|^2 + |\mathbf{d}_f|^2$.

By defining

$$\mathbf{X} := \begin{bmatrix} \mathbf{x}_f \\ \mathbf{x}_b \end{bmatrix} = \begin{bmatrix} r_f^1 & g_f^1 & b_f^1 & r_f^2 & g_f^2 & b_f^2 \dots r_f^N & g_f^N & b_f^N \\ r_b^1 & g_b^1 & b_b^1 & r_b^2 & g_b^2 & b_b^2 \dots r_b^N & g_b^N & b_b^N \end{bmatrix} \tag{1}$$

with N pixels in the neighbourhood of the considered pixel, Mester et al. [1] pointed out that the test statistic D^2 is identical to the smallest non-zero eigenvalue of the 2×2 matrix $\mathbf{X}\mathbf{X}^T$:

$$D^2 = eig\left(\mathbf{X}\mathbf{X}^T\right) = eig \begin{bmatrix} fore & cross \\ cross & back \end{bmatrix} \tag{2}$$

with three image qualifiers defined as

$$\begin{aligned} fore &:= \mathbf{x}_f \cdot \mathbf{x}_f^T \\ cross &:= \mathbf{x}_f \cdot \mathbf{x}_b^T \\ back &:= \mathbf{x}_b \cdot \mathbf{x}_b^T . \end{aligned} \tag{3}$$

Clearly, in a 3×3 neighbourhood, the matrix $\mathbf{X}$ contains the color values (red, green and blue) of $N = 9$ pixels in the foreground (indexed with f) and background (indexed with b). Equation (2) amounts to

$$0 = \begin{vmatrix} fore - D^2 & cross \\ cross & back - D^2 \end{vmatrix} . \tag{4}$$

Mester [1] empirically showed that D^2 follows a χ^2 probability density function with $3(N-1)$ degrees of freedom and a proportionality factor σ_u^2. Based on the knowledge of this distribution, the null hypothesis test can be reduced to a significance test, whereby D^2 is compared with a threshold t through $Prob[D^2 > t|H_0]=\alpha$ with the significance level α.

2.2 Bayesian Estimation

A Bayesian analysis allows the above decision to be made on a principled basis. The result of this analysis will be a foreground or 'change mask' Q, found by maximizing its a-posteriori probability (MAP). The binary change mask consists of pixels with labels $q_i = u$ (unchanged, background) or $q_i = c$ (changed, foreground). Based on the distance measurement D^2, change labels are assigned following the decision rule:

$$\frac{p(Q_c|D^2)}{p(Q_u|D^2)} \underset{u}{\overset{c}{\gtrless}} t.$$

A 'changed' label is assigned to a pixel if the left term is greater than the threshold t, otherwise it gets 'unchanged' assigned to it. Using Bayes' theorem we get

$$\frac{p(D^2|Q_c)}{p(D^2|Q_u)} \underset{u}{\overset{c}{\gtrless}} t \cdot \frac{p(Q_u)}{p(Q_c)}.$$

In order to calculate the fraction on the left side of the above equation, both conditional probability density functions must be estimated. Within a local neighbourhood comprising N pixels, both pdf's are modeled as zero-mean Gaussian distributions

$$\begin{aligned} p(D^2|Q_c) &= \left(\frac{1}{\sigma_c \cdot \sqrt{2\pi}}\right)^N \cdot e^{-\frac{D^2}{2\sigma_c^2}}, \\ p(D^2|Q_u) &= \left(\frac{1}{\sigma_u \cdot \sqrt{2\pi}}\right)^N \cdot e^{-\frac{D^2}{2\sigma_u^2}} \end{aligned}$$

with variances σ_c and σ_u. As foreground areas typically exhibit color differences of large magnitude w.r.t. the background, the variance σ_c^2 is much larger than the variance σ_u^2 caused by noise. Thus, the decision rule can be rewritten to

$$\left(\frac{\sigma_u}{\sigma_c}\right)^N \cdot e^{-\frac{D^2}{2\sigma_c^2}+\frac{D^2}{2\sigma_u^2}} \quad \underset{u}{\overset{c}{\gtrless}} \quad t \cdot \frac{p(Q_u)}{p(Q_c)},$$

which is equal to

$$e^{\frac{D^2}{2}\cdot\frac{\sigma_c^2-\sigma_u^2}{\sigma_u^2\cdot\sigma_c^2}} \quad \underset{u}{\overset{c}{\gtrless}} \quad \left(\frac{\sigma_c}{\sigma_u}\right)^N \cdot t \cdot \frac{p(Q_u)}{p(Q_c)}.$$

As $\sigma_c^2 \gg \sigma_u^2$, the sum $\sigma_c^2 - \sigma_u^2 \approx \sigma_c^2$ and thus

$$e^{\frac{D^2}{2\sigma_u^2}} \overset{c}{\underset{u}{\gtrless}} \left(\frac{\sigma_c}{\sigma_u}\right)^N \cdot t \cdot \frac{p(Q_u)}{p(Q_c)}.$$

Extracting D^2 yields

$$D^2 \overset{c}{\underset{u}{\gtrless}} \underbrace{2\sigma_u^2 \cdot ln\left(\left(\frac{\sigma_c}{\sigma_u}\right)^N \cdot t\right)}_{T_s} + \underbrace{2\sigma_u^2 \cdot ln\left(\frac{p(Q_u)}{p(Q_c)}\right)}_{T_{adapt}} \tag{5}$$

with a static threshold T_s and an adaptive threshold T_{adapt}.

2.3 Adaptive Threshold, MRF

Without any prior knowledge of the change mask, the adaptive threshold T_{adapt} in equation (5) is 0 because both probabilities $p(Q_u)$ and $p(Q_c)$ would then be equal. This often results in scattered foreground and background segments. To remedy this, one would like to bring spatio-temporal compactness considerations into play. Indeed, foreground objects tend to cover larger and more compact regions. If the object moves slowly compared to the framerate, this adds a temporal smoothness.

Such considerations are now added. A first element is the spatial compactness. A pixel should have a higher chance of being considered foreground if several of its neighbours have this status. This is a bit of a chicken-and-egg problem, however, as this assumes we already have a mechanism to decide on the neighbours first. In practice, this deadlock is solved by designing an iterative scheme. This will start with all pixels as background during the first iteration for the first frame. After that, the results from the previous iteration for that frame are used, or that of the last iteration of the previous frame in case a new frame is started. Note that the latter choice pushes towards temporal smoothness.

Still following Mester [1] and in order to bring the spatial compactness idea to bear, the change mask is considered to be sampled from a two-dimensional Gibbs/Markov random field (MRF). Hereby the a priori probability is expressed by

$$p(Q) = \frac{1}{Z} \cdot e^{-E(Q)} \tag{6}$$

with a normalization constant Z and an energy-term $E(Q)$. The smoother the boundary of the change mask within the considered neighbourhood (window W), the lower the energy-term $E(Q)$ is. Evaluating the smoothness and compactness can be simplified to account for changes between pixel pairs only. A pixel pair consists of two adjacent pixels in either horizontal, vertical or diagonal direction. Within any neighbourhood region, two kinds of pixel pairs can be distinguished: those who comprise the currently considered pixel, denoted as *local* pixel pairs, and all other pairs not comprising the current pixel, denoted as *global* pixel pairs (see figures 1-a,b).

Hence, the energy-term can be split into a local and a global term:

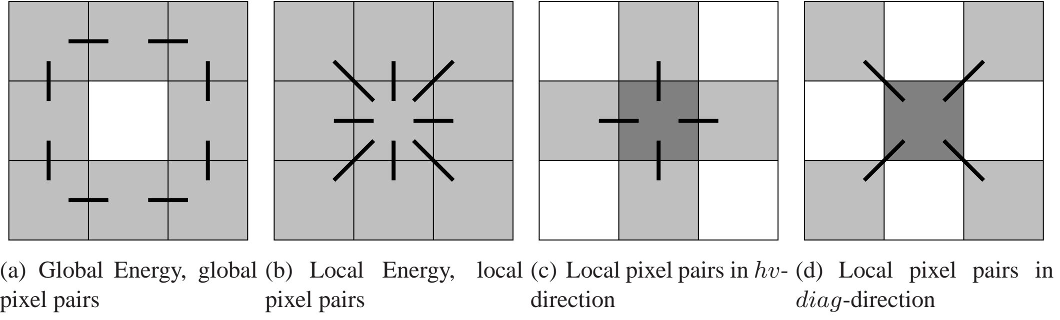
$$E(Q) = E_L(Q) + E_G(Q). \tag{7}$$

Based on a squared image grid, the 8 possible local pixel pairs within a 8-neighbourhood can be divided into two groups: 4 pairs of horizontal/vertical pairs (hv-dir.) and 4 diagonal pairs ($diag$-dir.) (see figures 1-c,d). Designating the number of adjacent pixels in hv direction as ν_B and in $diag$ direction as ν_C wrt. the label q_i, the local energy term can be rewritten as

$$E_L(Q) = \nu_B(q_i) \cdot B' + \nu_C(q_i) \cdot C', \tag{8}$$

whereby B' and C' are constant multiplicative factors influencing the level of compactness. Thus, equation (7) is reformulated as

$$E(Q) = \nu_B(q_i) \cdot B' + \nu_C(q_i) \cdot C' + E_G(Q), \tag{9}$$



(a) Global Energy, global pixel pairs (b) Local Energy, local pixel pairs (c) Local pixel pairs in hv-direction (d) Local pixel pairs in $diag$-direction

Figure 1: Local and global pixel connectivities.

Combining (9) with (6) and accounting for both labels $q_i = c$ (changed) and $q_i = u$ (unchanged) yields to

$$\begin{aligned} p(Q_c) &= \frac{1}{Z} \cdot e^{-(\nu_B(q_i{=}c)\cdot B' + \nu_C(q_i{=}c)\cdot C' + E_G(Q_c))}, \\ p(Q_u) &= \frac{1}{Z} \cdot e^{-(\nu_B(q_i{=}u)\cdot B' + \nu_C(q_i{=}u)\cdot C' + E_G(Q_u))}. \end{aligned}$$

When inserting above formulas into equation (5), the adaptive threshold term is

$$\begin{aligned} T_{adapt} &= 2\sigma_u^2 \cdot ln\left(\frac{p(Q_u)}{p(Q_c)}\right) \\ &= 2\sigma_u^2 \cdot \left[(\nu_B(q_i{=}c) - \nu_B(q_i{=}u)) \cdot B' + (\nu_C(q_i{=}c) - \nu_C(q_i{=}u)) \cdot C'\right]. \end{aligned} \tag{10}$$

One can easily observe that

$$\begin{aligned} \nu_B(q_i{=}c) + \nu_B(q_i{=}u) &= 4 \\ \nu_C(q_i{=}c) + \nu_C(q_i{=}u) &= 4. \end{aligned}$$

By setting

$$C' = B'/2$$

and defining

$$B := \sigma_u^2 \cdot B'$$

we can simplify equation (10) to

$$\begin{aligned} M &:= 2 \cdot \nu_B(q_i{=}c) + \nu_C(q_i{=}c) \\ T_{adapt} &= 12B - 2BM. \end{aligned} \tag{11}$$

Indeed, the lower the amount of foreground pixels in the surrounding change mask is ($q_i{=}c$), the lower M and therefore the higher the adaptive threshold gets, increasing the barrier at which a considered pixel may be assigned to foreground. This intuitive behaviour results in smooth and compact regions, even in small neighbourhoods, i.e. 3×3 pixels.

2.4 Darkness Compensation

Though an intensity-invariant method as Mester's does provide robustness against shadows and lighting changes, such invariance also has drawbacks. Often, part of the foreground or background will be dark, i.e. close to black. As black can be seen as a low intensity version of any color, the current approach will never trigger segmentation in those areas.

Our solution consists of adding an additional component to the vectors $\mathbf{x}_b$ and $\mathbf{x}_f$ in eq. (1). This additional component has a fixed value of $\sqrt{O_{dc}}$. (The awkward use of the square-root has been opted for as this simplifies further notation; e.g. *fore, cross, and back* of eq. (3) are now increased by O_{dc}.) This additional component renders the color similarity measure more sensitive to differences, esp. when dark pixels are involved. Indeed, this additional component has the effect of lifting the $3N$-dimensional ground plane in the enlarged $3N + 1$-space, to height $\sqrt{O_{dc}}$, thereby distinguishing vectors that were colinear but had different norms.

When running these new $3N+1$-dimensional vectors through the criterion, we come to the following observations:

- The resulting distances never decrease (proof is straightforward), thus regions that were previously segmented, remain segmented after the manipulation (for the same threshold T).
- The comparison of equal vectors remains unhampered (D^2 is 0 in both approaches).
- Moreover,when $\|\mathbf{x}_f\| = \|\mathbf{x}_b\|$ (more or less equal intensities), there is no impact whatsoever on the distance measure.
- Vectors that were previously colinear but of different sizes, will not remain colinear. The impact is dependent on the difference in intensity.
- As O_{dc} goes to infinity, the distance measure becomes $(fore + back - 2 \cdot cross)/2$. This equals $\|\mathbf{x}_f - \mathbf{x}_b\|/2$, which still yields a valid distance measure for background segmentation, but totally lacks the illumination invariance property useful to cope with shadows. The choice of O_{dc} determines how illumination sensitive the result is.

The above observations show that the provided manipulation now correctly segments dark coloured areas, compared with bright background. This was previously not the case, as these areas were seen as noise on a 0-vector. Furthermore, normal operation, where foreground looks like background or where foreground was already segmented from the background without the extra compensation, is not impaired.

2.5 Final Decision Rule

We can now convert the final decision rule into a form, which is suitable for computation. First, the determinant in equation (4) has to be solved involving a square root term, which is often a bottleneck in high-speed implementations:

$$D^2 = \frac{1}{2} \cdot \left(fore + back - \sqrt{(fore - back)^2 + 4 \cdot cross^2} \right). \tag{12}$$

Fortunately, the square root must not explicitely be calculated, but D^2 directly compared to a threshold $T = T_s + T_{adapt}$ (see formula 5). Without loss of generality we rewrite the decision rule and get

$$D^2 \overset{c}{\gtrless} T,$$

which assigns the 'changed' label only when the above inequation is fulfilled. Applying this rule to (12) results in

$$\frac{1}{2} \cdot \left(fore + back - \sqrt{(fore - back)^2 + 4 \cdot cross^2} \right) \overset{c}{\gtrless} T,$$

which is equal to

$$fore + back - 2T \overset{c}{\gtrless} \sqrt{(fore - back)^2 + 4 \cdot cross^2}.$$

Taking the square of both sides yields two inequalities

$$\begin{aligned} (fore + back - 2T)^2 &\overset{c}{\gtrless} (fore - back)^2 + 4 \cdot cross^2 \\ fore + back - 2T &\overset{c}{\gtrless} 0. \end{aligned}$$

The former one can be further expanded:

$$fore \cdot back - fore \cdot T - back \cdot T + T^2 \overset{c}{\gtrless} cross^2$$

or in an even simpler form

$$(fore - T) \cdot (back - T) \overset{c}{\gtrless} cross^2.$$

Finally, the decision rule is reduced to testing the following two inequalities only:

$$\begin{aligned} (fore - T) \cdot (back - T) &\overset{c}{\gtrless} cross^2 \\ fore + back &\overset{c}{\gtrless} 2T. \end{aligned}$$

The first inequality shows that either $fore$ and $back$ are both $> T$, or both $< T$, while additionally considering the latter inequality forces $fore > T$ or $back > T$. Therefore we can again simplify the above formulas:

$$\begin{aligned} (fore - T) \cdot (back - T) &\overset{c}{\gtrless} cross^2 \\ fore &\overset{c}{\gtrless} T. \end{aligned}$$

As described in section 2.4, darkness compensation is integrated by adding O_{dc} to each of the three qualifiers $fore$, $back$ and $cross$, which yields

$$\begin{aligned} (fore + O_{dc} - T) \cdot (back + O_{dc} - T) &\overset{c}{\gtrless} (cross + O_{dc})^2 \\ fore + O_{dc} &\overset{c}{\gtrless} T. \end{aligned}$$

We define a new total threshold

$$T_t := T - O_{dc} = T_s + T_{adapt} - O_{dc} \tag{13}$$

and rewrite the two inequalities:

$$\begin{aligned} (fore - T_t) \cdot (back - T_t) &\overset{c}{\gtrless} (cross + O_{dc})^2 \\ fore &\overset{c}{\gtrless} T_t. \end{aligned}$$

Now, when also the adaptive threshold in equation (11) is integrated, the final decision rule set can be formulated:

$$\boxed{\begin{aligned} M &= 2 \cdot \nu_B(q_i{=}c) + \nu_C(q_i{=}c) \\ T_t &= T_s + 12B - 2BM - O_{dc} \\ \\ (fore - T_t)(back - T_t) &\overset{c}{\gtrless} (cross + O_{dc})^2 \\ fore &\overset{c}{\gtrless} T_t \end{aligned}} \tag{14}$$

Influenced by three user-defined parameters, a static threshold T_s, a darkness offset O_{dc} and a compactness value B, a label 'changed' is assigned to a pixel if both inequalities are fulfilled. Otherwise the pixel's label is set to 'unchanged'. Notice that the $back$ qualifier only depends on the background image, which in practice is the average over several background images, and has to be computed only once, while $fore$ and $cross$ need to be updated every frame.

2.6 Iterative, randomized MRF computation

Under consideration of a 3×3 neighbourhood, we can observe that pixels with a chessboard distance ≥ 2 do not directly affect each other and can therefore be handled in parallel.

k	l	k	l	k	l
m	n	m	n	m	n
k	l	k	l	k	l
m	n	m	n	m	n
k	l	k	l	k	l

This processing step, denoted as substep, operates on a subset of the input data, whereby pixels within each subset are mutually independent wrt. the MRF computation. To cover all the input data, several substeps have to be executed in sequential manner. The smallest number of substeps is given by the local partition shown in the left image. Four substeps k, l, m and n are executed one after each other, whereby the order is randomly chosen, ensuring a uniform distribution over time. After all substeps are done, the whole process is repeated several times until convergence in the segmentation wrt. compactness is reached. From the implementation point of view, at each iteration j we select the execution sequence by randomly picking one of the 24 possible permutations of (k, l, m, n) by

$$S_j = rand\left(perm(k, l, m, n)\right). \tag{15}$$

It is important to mention that the result of each substep is written back into the change mask, which serves as input for the next substep. A priori knowledge from the previous frame's change mask is integrated by initializing the change mask with the prior change mask and then start the MRF-iterations. If no prior frame is given, the previous change mask is set to full background, i.e. black color.

The program flow, as depicted in figure 2, starts with the computation of $fore$, $back$ and $cross$ based on the average background image BG and the current foreground image FG. This is followed by one iteration comprising four substeps with the change mask of the previous frame as input and a parameter set 1 (static threshold T_s, darkness offset O_{dc} and compactness value B_1). Finally we run the iteration j times with the second parameter set, where the static threshold and the darkness offset both remain the same while the compactness value differs between between both parameter sets. The outcome of all iterations represents the final segmentation, which serves as input for the next frame.

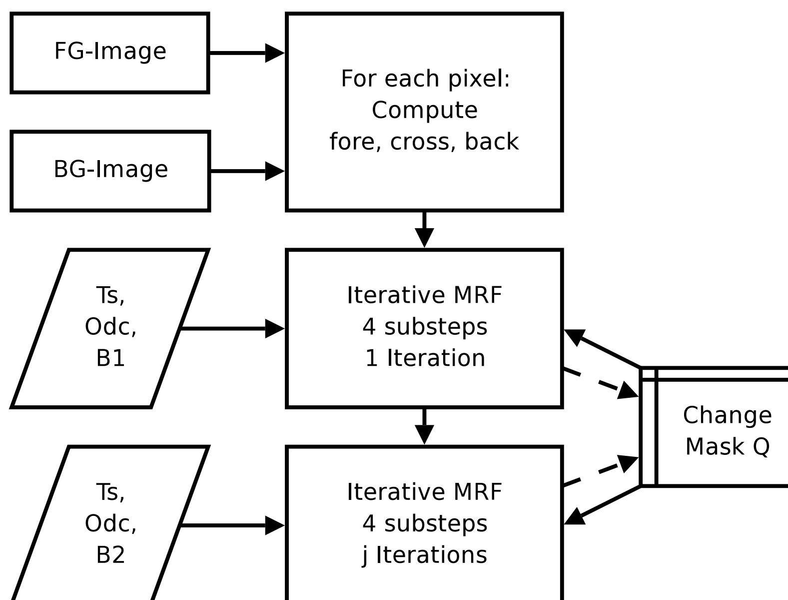


Figure 2: Program flow for the iterative MRF computation.

3 Implementation Issues

Following the decision rules (14) and the iterative, randomized MRF-computation approach, we can now design a GPU-based implementation. As modern graphics hardware consists of multiple independent processing units, they seem to be predistined for parallel pixel operations and thus faster than nowadays CPU's. Although programming the GPU requires some in-depth knowledge of the underlying hardware architecture in order to fully optimize the application for speed, more and more work is done on implementing general-purpose computation on graphics hardware (see [12]). Unfortunately, several limitations are imposed by device drivers, for example hardware manufacturers often prioritize development under Windows operating system over Linux. The following sections describe some hints and useful techniques for general GPU-based programming. The graphics card we used hereby is a NVidia GeForce 6800GT, also known as NV40.

In the last years GPU manufacturer as well as the OpenGL consortium agreed for two user-programmable processing units on the graphics hardware:

- A Vertex Shader,
- A Fragment (or Pixel) Shader.

Figure (3) shows the processing pipeline using these two programmable Shader units. Via OpenGL or DirectX the user defines drawing primitives and textures (geometry stage). For example one wants to draw a rectangle defined by it's 4 corner vertices, whereby each vertex may also correspond to a texture coordinate. These drawing commands are sent to the graphics hardware, where the 3D-geometry is stored. Each of these vertices is processed by the Vertex Shader by assigning per-vertex informations, such as color, texture coordinate, 2D-projected coordinates, and by performing a 3D-2D transformation.

The rasterizer then projects the 2D-rectangle onto the drawing surface, which has similar dimensions as the final framebuffer. Hereby, the intermediate 2D-pixel values as well as the texture coordinates are computed by linear interpolation.

For each pixel within the rasterized rectangle, a separate Fragment (Pixel) Shader is executed, which assigns a color and depth value to the pixel. In case of texture mapping activated, the pixel's color depends on the color of one or more textures, which reside in texture memory of the graphics hardware. Although memory bandwidth is up to approx. 30GB/s, the latency for such texture lookups is still a bottleneck in processing performance.

After a color and depth value is assigned to each pixel, these values are stored in a target buffer, typically the visible framebuffer. Interesting for general purpose computing on the GPU is the usage of so-called offscreen buffers, which allow for writing into a dedicated non-visible buffer. Only if necessary, the final result image is displayed by writing into the framebuffer instead or by copying the PBuffer contents into a texture and project it onto the visible framebuffer.

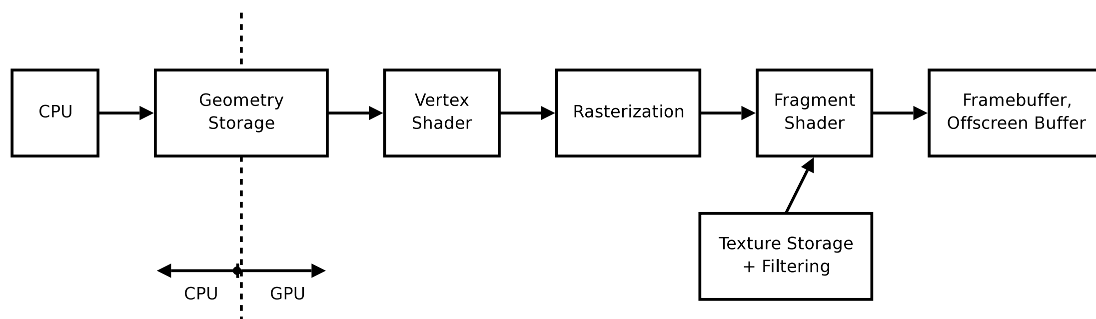


Figure 3: Processing pipeline on graphics hardware using programmable Shader units.

3.1 Programming Language

Among the different programming languages available on the market, the following separation can be made:

- High-Level Shading Language
  - HLSL
  - GLSL
  - Cg
- Low-Level Shading Language (assembler instructions)
- OpenGL 2.0

The High-Level Shading Languages have a C-like command syntax and interface the user application through OpenGL or DirectX. HLSL is constrained to the OpenGL interface, whereas GLSL works only with DirectX. Cg (C for graphics), developed by NVIDIA Corporation together with Microsoft Corporation, provides both interfaces to OpenGL as well as to DirectX and is therefore our choice of programming language. Shader programs can be compiled on beforehand or during runtime, whereby the resulting assembler instructions are executable on the current hardware. Instead of using a high-level language, one might also directly write the Shader application in pure assembler code with the risc of lacking interoperability and hardware compatibility. In our application we use Cg together with the OpenGL interface under a Linux operating system.

3.2 Drawing Context

In order to allow multitasking on the graphics hardware, each application accessing the GPU must have it's own drawing context. Under Linux and OpenGL this is managed by toolkits such as glut, Qt or gtk. Notice that only one context can be active at a time. In the case of multiple contexts this calls for context switches, which are - despite of high memory bandwidth - usually very cost-intensive.

3.3 Offscreen-Buffers

The programmable Fragment Shader computes the color and depth value for an output pixel based on the previous color of the pixel and the assigned textures. The GeForce 6 series can access up to 8 textures within the Shader program, whereas the newer GeForce 7 series can access up to 16 textures. Reading from the framebuffer is not granted in this Shader.

The final color and depth value is written into a buffer, typically the displayed framebuffer. Often it is not required to display intermediate results of an application rendering in multiple steps or using multiple Shader programs in serial. This is where the role of a non-visible (offscreen) buffer comes into play. Under Windows operating system the framebuffer can be switched to a readable mode and thus acting as a normal, readable texture for the Fragment Shader. Under Linux only the latest driver versions support this kind of operability, also known as *framebuffer objects*. Most common is a special *PBuffer* object, which is a pure offscreen and write-only storage w.r.t. the Shader unit. This means that a Fragment Shader program can write into a PBuffer object at one time instance. Afterwards the contents of this buffer may be copied into a dedicated texture memory, which itself can then be read by the Shader programs.

Indeed, this requires some additional memory for the texture storage and, in case of multiple PBuffers, several context switches. In our application we use only one PBuffer object, whereby its contents are copied into different textures, dependent on the currently applied Shader program.

It is important to mention that PBuffer objects require a valid drawing context on beforehand in order to operate safely, which is the task of the application framework on the CPU. PBuffer objects can store

3Byte (RGB) or 4Byte (RGBA) unsigned char data as well as floating point (16bit and 32bit on newer hardware). We found out that the best timing performance is reached by using standard unsigned char data format and that especially floating point texture lookups are rather time consuming.

3.4 GPU Data Formats

As modern graphics hardware not only implements unsigned or signed 8bit data formats but also floating point formats in 16bit or 32bit (newer hardware operates also on 128bit formats), it is important to keep in mind the times needed for texture lookups or buffer copies.

In our algorithm we have to compute the three image qualifiers $fore$, $back$ and $cross$, whereby the hardware imposed each pixel's value ranging from 0 to 1. Obviously, 8bit resolution per value is not sufficient for accurate computation and therefore we experimented with a 16bit floating point format. Unfortunately, the texture lookup times are distinctively bigger than on unsigned char 8bit formats. As the Shader internally operates on 32bit floating point resolution, we looked for a storage type providing acceptable accuracy combined with good timing performance. Cg supports so-called packing functions, which allow for stacking two 16bit float values into four 8bit unsigned char's. Clearly, each image qualifier is truncated from the internal 32bit float to a 16bit float and packed into 2 Bytes of the 4Byte-RGBA unsigned char buffer. This special mode guarantees the fastest texture lookups together with accurate pixel resolution.

3.5 Bilinear Interpolation

Since texture lookups are always time-expensive, some operations on pixel neighbourhoods can be simplified using the bilinear interpolation functionality on graphics hardware. Figure (4-a) shows how bilinear interpolation works on two neighbouring pixels.

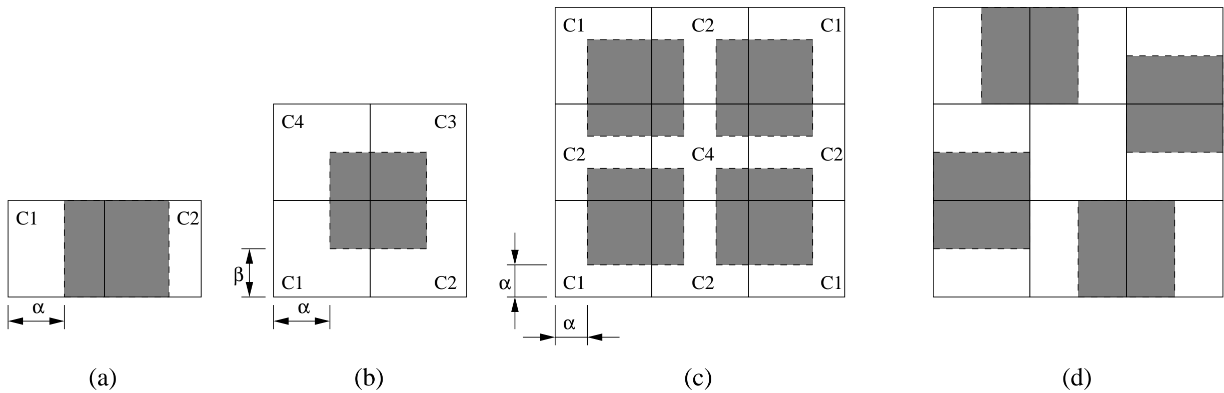


Figure 4: Bilinear Interpolation on the GPU reduces the required number of texture lookups for neighbourhood pixel operations.

Based on the 2D-coordinates of the current pixel C_1, the new texture coordinates are

$$T_c = (u_i + \alpha, v_i), \alpha = 0 \ldots 1$$

whereby u_i is the base coordinate of C_1 in horizontal direction and v_i in vertical direction. The so-defined area covering both texels C_1 and C_2 is shown as dark-grey shaded box in the figure. Notice that OpenGL has its coordinate system origin in the bottom left corner, whereas most image coordinate systems start on the top left corner. The further explanations are based on the OpenGL system.

The texture lookup with the new texture coordinates T_c returns a color value S of

$$S = tex(T_c) = tex(u_i + \alpha, v_i).$$

Since each pixel has a size of 1×1, the left pixel with color value C_1 is covered by the area $(1-\alpha)$ and the right pixel covered by (α). Hence, the interpolated sum, returned by the texture lookup, is

$$S = tex(T_c) = (1-\alpha)\cdot C_1 + \alpha \cdot C2.$$

When calculating the exact sum of both pixels, i.e. $C_1 + C_2$, only one texture lookup with $\alpha = 0.5$ has to be done yielding

$$S_{\alpha=0.5} = \frac{C1 + C2}{2}.$$

Notice that on the GPU when using 8bit unsigned char formats, only fixed point values between 0 and 1 are possible. Greater values will be clamped to 1. The above lookup guarantees that the resulting color value always lies in that range, but a scaling factor of 2 has to be managed for further operations.

In figure (4-b) the bilinear interpolation is extended in v-direction and thus generalizes the texture coordinates to

$$T_c = (u_i + \alpha, v_i + \beta),\ \alpha, \beta = 0 \ldots 1,$$

again based on the bottom left pixel C_1. The resulting lookup generates the following color value:

$$S = tex(T_c) = (1-\alpha)(1-\beta)\cdot C_1 + \alpha(1-\beta)\cdot C2 + \alpha\beta \cdot C3 + (1-\alpha)\beta \cdot C4.$$

When computing the sum of all 4 pixel values, the offsets α and β are both 0.5, which yields

$$S_{\alpha,\beta=0.5} = \frac{C_1 + C_2 + C_3 + C_4}{4},$$

with a scaling factor of 4. Instead of 4 texture lookups and 3 intermediate summations, we reduced the computation of the sum to only one texture lookup.

We further extend this approach to compute the sum of all pixels within a 3× 3 neighbourhood, which is often required in computer vision algorithms. Herefore we need 4 texture lookups, as depicted in figure (4-c). We can observe that the center pixel C_4 is accessed 4 times, each pixel denoted as C_2 2 times and each corner pixel C_1 only once. Each of the interpolated lookups S_i computes the sum of 4 pixels and all 4 S_i together result in the final sum of all 9 pixels. Considering the bottom left texture lookup and, without loss of generality, assuming all C_2 have the same color value, we retrieve

$$S_i = (1-\alpha)^2 \cdot C1 + 2\alpha(1-\alpha)\cdot C_2 + \alpha^2 \cdot C_4.$$

It turns out that C_4 occurs in the final sum 4 times and each C_2 twice, but in the real sum they appear only once. Therefore we have to equal their coefficients by

$$1 \cdot \underbrace{(1-\alpha)^2}_{C_1} \equiv 2 \cdot \underbrace{\alpha(1-\alpha)}_{C_2} \equiv 4 \cdot \underbrace{\alpha^2}_{C_4}.$$

Solving for α results in

$$\alpha = 1/3.$$

Inserting into S_i and summing up all 4 lookups yields

$$\sum_{i=1}^{9} C_i = \frac{9}{4} \cdot \sum_{i=1}^{4} S_i.$$

Based on the center pixel C_4 the texture coordinates can be rewritten to

$$\begin{aligned}T_{c,1} &= (u_i - 1 + \alpha, v_i - 1 + \alpha)\\ T_{c,2} &= (u_i - 1 + \alpha, v_i + 1 - \alpha)\\ T_{c,3} &= (u_i + 1 - \alpha, v_i - 1 + \alpha)\\ T_{c,4} &= (u_i + 1 - \alpha, v_i + 1 - \alpha)\\ \sum_{i=1}^{9} C_i &= \frac{9}{4} \cdot \sum_{i=1}^{4} tex(T_{c,i}).\end{aligned}$$

In figure (4-d) another example is given, computing the sum of all pixels in the 8-neighbourhood of the center pixel without taking the latter into account. Again, only 4 texture lookups and 3 summations are required instead of 8 lookups and 7 summations.

The above techniques gain enourmous speedups in applications, computing the weighted sum of pixels in a local neighbourhood. It is important to mention that bilinear interpolation is activated only when texture filtering is turned on, i.e. the minification filter. This filter is applied whenever a texel (pixel in the texture) is smaller than the resulting pixel in the output buffer. Unfortunately, uploading a texture to the graphics card with minification filter enabled results in higher upload times. For example on a Pentium4 with 3GHz, AGP8x, NVidia GeForce 6800GT, a 640×480 pixel RGB image is uploaded within 1ms without minification filter. Enabling the latter yields approx. 2.5ms upload time. Hardware manufacturer do not explicitely explain this behaviour, but it is important to keep this fact in mind when running Computer Vision algorithms on the GPU.

Despite the Fragment Shader operates on 32bit floating point data, the interpolation in hardware is performed with much lower resolution. NVidia claimes their interpolation is based on 8bit, whereas ATI has only 5 bit resolution. Clearly, the factore α and β are quantized with a resolution of 8bit resp. 5bit. Internally, both manufacturers gain their speed by reading out the texel coverage based on twodimensional lookup-tables, wheres α and β are used as indices. Although rendering results look nice and the user won't recognize a difference, the gathered results are not mathematically exact. This drawback allows the usage of bilinear interpolation only when no highly accurate results are required, such as in the computation of M in equation (14). For the summation of the image qualifiers $fore$, $back$ and $cross$, the interpolation cannot be applied, also because of the special 16bit-packed data format used.

4 GPU-based Implementation

Figure 5 gives an overview of the program flow of the GPU implementation, whereby a fixed neighbourhood of 3×3 is used. The round-shaped boxes in the left column symbolize the different Shader-programs, which are described in the following sections. Each Shader gathers information from one or more inputs and writes into an offscreen PBuffer object. A copy-command afterwards transfers the currently written data into the target texture, as depicted in the right column. Thereby not the full texture object has to be written but just the affected memory areas (grey-shaded in the right column of the figure).

4.1 The *CalcNorm*-Shader

This first Shader steps through the input image, calculating for each pixel $\mathbf{y}_i=[r_i\ g_i\ b_i]$ at location i the three qualifiers f_i, b_i and c_i based on an input image FG and the background average image BG by computing the following three dot products:

$$\begin{aligned}f_i &= \mathbf{y}_{i,f} \cdot \mathbf{y}_{i,f}^T\\ b_i &= \mathbf{y}_{i,b} \cdot \mathbf{y}_{i,b}^T\\ c_i &= \mathbf{y}_{i,f} \cdot \mathbf{y}_{i,b}^T\end{aligned}$$

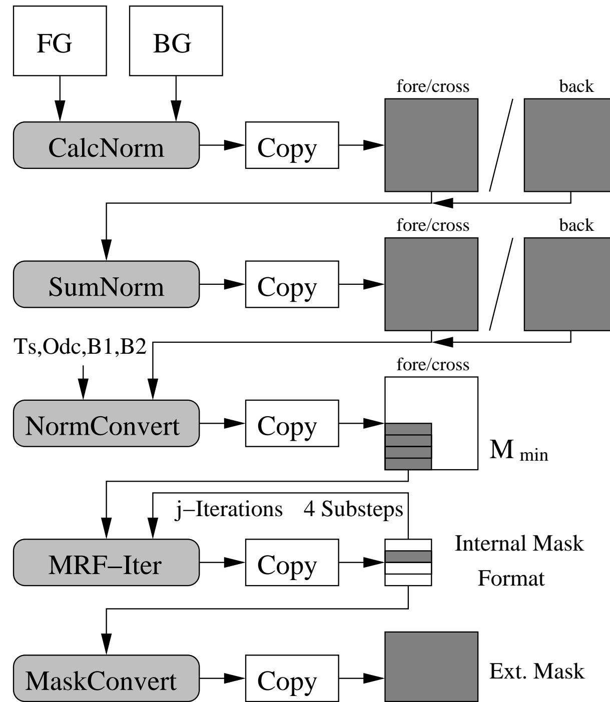


Figure 5: Program flow for GPU implementation.

The indices f and b denote foreground and background pixels. The dot product computation on the GPU is rather trivial, it is executed within one machine operation and thus much faster than a CPU variant. It turns out that decision rule (14) is only applicable when all three qualifiers have at least 16bit resolution. As described in section 3.4, we pack two 16bit floating point data into one 32bit RGBA buffer. Each of the two required buffers has the same dimensions than the input image and a depth of 4Byte, thus being a 1st storage level buffer (see fig. 6 in section 4.4).

Only when a background image has changed, the b-buffer has to be updated, while remaining constant during normal operation. Hence, we pack the values f_i and c_i in one RGBA texture and b_i in a separate RGBA texture.

4.2 The *SumNorm*-Shader

After the dot products are computed per pixel, the three qualifiers $fore$, $back$, and $cross$ can now be derived by following eq. (3). Accounting for f_i, b_i and c_i, the equation is simplified to summing up all dot products in the neighbourhood W_i around the pixel location i:

$$\begin{aligned} fore &= \sum_{j \epsilon W_i} f_j \\ back &= \sum_{j \epsilon W_i} b_j \\ cross &= \sum_{j \epsilon W_i} c_j \end{aligned}$$

As mentioned in section 3.5, hardware accelerated bilinear interpolation cannot be performed because of the packed 16bit data format used. Again $fore$ and $cross$ are stacked in a RGBA texture while $back$ remains in a separate RGBA buffer. Notice that $back$ has to be computed only once during normal operation and needs to updated only when the background image has changed.

4.3 The *NormMixBack*-Shader

As will be seen in the *NormConvert*-Shader in section 4.4, memory conversions are performed in order to optimize the amount of texture lookups and to gain full power of the GPU by operating on 4 color values (RGBA) in parallel. The texture buffer for the $fore$ and $cross$ qualifiers are already optimally used, whereas the buffer for the $back$ values is only filled half, i.e. the first 16bits of the 32bit storage is written. Indeed, by splitting the $back$ texture into two halfs and copying the data of the second half into the unused 16bits of the first half, we can reduce the amount of required texture lookups in the following computation steps. Clearly, by applying a texture lookup, we obtain two $back$ values at once. This quite simple task is performed by the *NormMixBack*-Shader.

4.4 The *NormConvert*-Shader

Now that all parameters for testing the decision rule (14) are known, the iterative MRF computation begins. However, for each substep in an iteration the required inequalities have to be recalculated. As this would slow down the overall process, the ruleset is reformulated to a simpler test. The only variable during a MRF iteration is M and therefore we can rewrite the decision rule based on M. We first define a new parameter S

$$S \quad := \quad \frac{T_s + 12B - O_{dc}}{2B},$$

which simplifies the definition of the total threshold T_t to

$$T_t \quad = \quad 2B \cdot S - 2B \cdot M.$$

The second inequality of decision rule (14) now turns into

$$fore \quad \overset{c}{\gtrless} \quad 2B \cdot S - 2B \cdot M,$$

from which M is extracted

$$M \quad \overset{c}{\gtrless} \quad S - \frac{fore}{2B}. \tag{16}$$

The first inequality of decision rule (14) turns into

$$\begin{aligned} (fore - T_t)(back - T_t) \quad &\overset{c}{\gtrless} \quad (cross + O_{dc})^2 \\ fore \cdot back - fore \cdot T_t - back \cdot T_t + T_t^2 \quad &\overset{c}{\gtrless} \quad (cross + O_{dc})^2 \\ fore \cdot back - (2B \cdot S - 2B \cdot M)(fore + back) + (2B \cdot S - 2B \cdot M)^2 \quad &\overset{c}{\gtrless} \quad (cross + O_{dc})^2. \end{aligned}$$

By defining

$$K \quad := \quad \frac{\sqrt{(fore - back)^2 + 4(cross + O_{dc})^2}}{4B}$$

and solving for M we get

$$M \quad \overset{c}{\gtrless} \quad S + K - \frac{fore + back}{4B}. \tag{17}$$

As a 'changed' label is only applied when the inequalities (16) and (17) are both fulfilled, the conjunction of both formulas yields:

$$M \quad \overset{c}{\gtrless} \quad S + max\left(-\frac{fore}{2B}, K - \frac{fore + back}{4B}\right),$$

which is equal to

$$M \overset{c}{>} \underbrace{S - \frac{fore}{2B} + max\left(0, K + \frac{fore - back}{4B}\right)}_{M_{min}}.$$

We summarize the above formulas and get a decision rule set, suitable for fast implementation:

$$\boxed{\begin{aligned} M &= 2 \cdot \nu_B(q_i{=}c) + \nu_C(q_i{=}c) \\ K &= \frac{\sqrt{(fore - back)^2 + 4(cross + O_{dc})^2}}{4B} \\ S &= \frac{T_s + 12B - O_{dc}}{2B} \\ M_{min} &= S - \frac{fore}{2B} + max\left(0, K + \frac{fore - back}{4B}\right) \\ M &\underset{u}{\overset{c}{\gtrless}} M_{min} \end{aligned}} \tag{18}$$

Notice that during the MRF iteration, the image qualifiers $fore$, $back$ and $cross$ remain constant, as well as the parameters S and K. Thus, M_{min} can be pre-calculated w.r.t. a parameter set B, T_s, and O_{dc}. Due to the fact that M can only vary between 0 and 12 and therefore consumes only 4 bits, both $M_{min,1}$ for the first parameter set and $M_{min,2}$ for the second parameter set are packed into one 8bit color value by $((M_{min,1} \ll 4) \mid M_{min,2})$. As described in section 2.6, the iterative MRF-computation operates on 4 subsets of the input data. These subsets are randomly chosen and sequentially processed. In order to have this processing step independent of the currently selected subset of the input data, i.e. the change mask with labels 'unchanged' (=value 0) and 'changed' (=value 1), we split the latter into 4 equally sized data buffer. Figure 6 gives an overview of the implemented data structures.

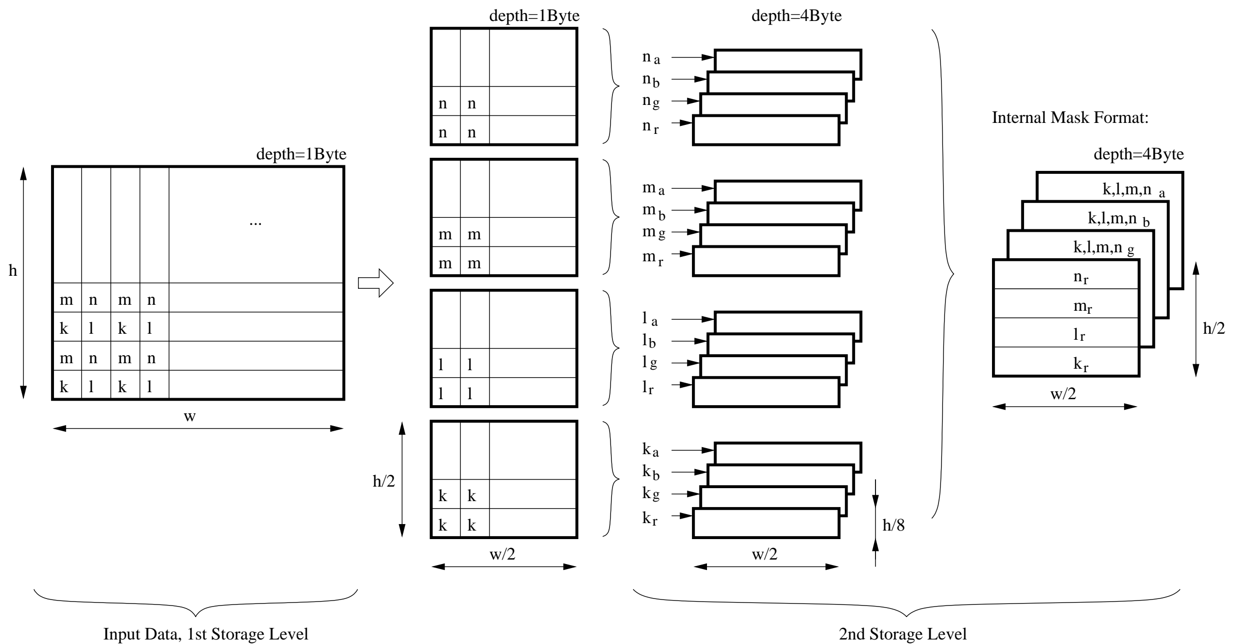


Figure 6: Data formats used on the GPU

The input data obviously has the same dimensions than the segmented image mask, denoted as height h and width w. Since the change mask pixels can only store two values (0, 1), the buffer has a depth of 1Byte. Each 2×2 block of input pixels consists of elements of the 4 subsets k, l, m and n. Starting at the first storage level, the input data is then separated into 4 memory blocks, which each has half the width and half the height of the input data. MRF computation is now performed on one of these equally sized memory blocks.

Due to the fact that the graphics hardware can process 4Bytes at once, we decided to convert the 4 subsets into a second storage level, whereby the resulting depth is 4Byte. This is done by vertically slicing each subset into 4 equally sized subblocks and assigning them to one of the 4 available color channels r, g, b and a. Clearly, the bottommost quarter of the subset k is assigned to the red channel of the second storage level, that is k_r. The topmost quarter of the same subset is stored in the alpha channel (k_a) and so forth. As depicted in figure 6 each subblock has now a depth of 4Byte and dimensions $w/2 \times h/8$. Combining these 4 subblocks into one storage buffer leads to the internal mask format, on which all MRF computations are done by processing 4 pixels in parallel.

The tasks of the *NormConvert* Shader are the following:

1. Fetch $fore$, $back$ and $cross$, corresponding to the current pixel location,
2. Compute $M_{min,1}$ and $M_{min,2}$ for each parameter set and combine both in a 1Byte value M_{min},
3. Store the resulting M_{min} value in a 1st storage level PBuffer.

Since the image qualifiers $fore$ and $cross$ are no longer needed by successing computations, overwriting the texture memory, which stores $fore$ and $cross$, by the values M_{min} reduces the overal amount of texture memory usage. Thus, after the *NormConvert*-Shader has finished, the PBuffer is copied into the first storage level texture, where the two 16bit qualifiers $fore$ and $cross$ have been stored previously.

4.5 The *MRF-Iter*-Shader

As already mentioned in the previous sections, the iterative MRF-computation comprises 4 substeps, each of which solves the MRF for compactness and smoothness. This is done by calculating M per pixel and comparing it with the stored value M_{min}. The different subsets are selected based on eq. (15).

As calculation of M_{min} for a pixel within a subset depends on its surrounding neighbours, special care has been taken on the cutting edges, where the slicing of the subset into the 4 color channels occurs (see figure 6). For example, M_{min} has to be computed for a pixel in the top row of the subset k_r (red channel of k), which, amongst others, requires access to pixels one line above. Indeed, these pixels do not reside in the same color channel, but now in k_g (green channel of k). Therefore we distinguish between 3 processing areas within each subset, as demonstrated in figure 7.

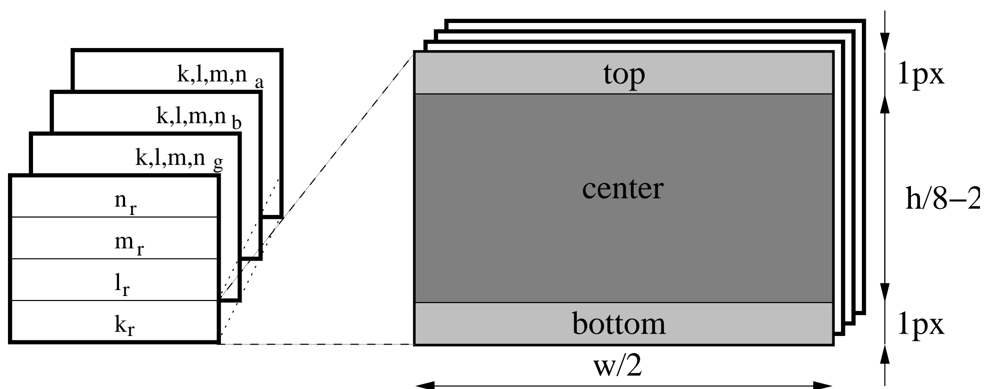


Figure 7: Separation into 3 processing zones for the *MRF-Iter*-Shader

In the shown example, the subset k is separated into 3 vertical pieces. The top- and $bottom$-parts are one pixel high, whereas the $center$-area covers the rest of the data. The MRF-computation is now performed on each area sequentially, whereby again operations are performed in exploit the 4Byte-parallelism. Clearly, within each of the top, $center$ and $bottom$ areas we process the color channels r, g, b, and a of a selected data subset at the same time.

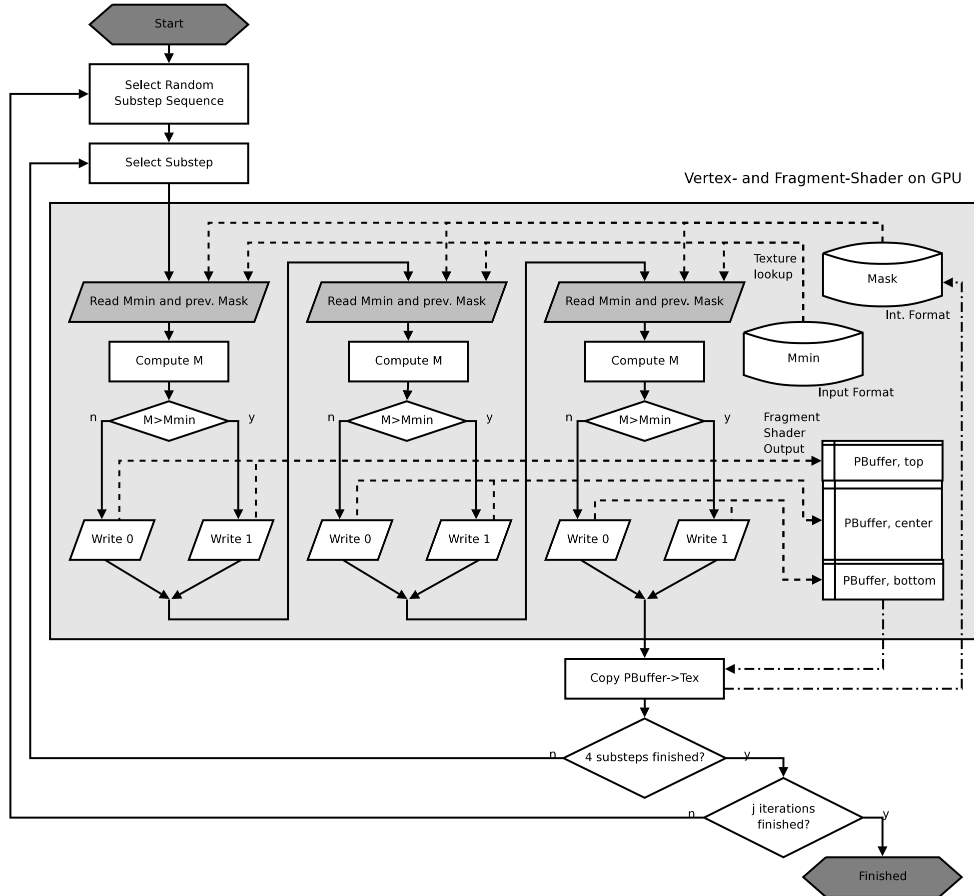


Figure 8: Control-Flow (solid lines) and Data-Flow (dashed lines) of the *MRF-Iter*-Shader

The Shader's control- and data-flow is shown in figure 8. The process starts by choosing the sequence of the 4 substeps k, l, m, and n on the CPU. For each substep in this sequence and the comprising 3 areas top, $center$, and $bottom$, the *MRF-Iter*-Shader solves the MRF. This computation includes the following steps:

1. Read M_{min} from the texture buffer (1st storage level texture),
2. Compute M, dependent on the local pixel neighbourhood, by reading the pixel labels stored in the working, mask buffer
3. Apply the segmentation decision rule by assigning an unchanged label (pixel value 0) or a changed label (pixel value 1) following $M \overset{c}{>} M_{min}$.

The output values stored in the offscreen PBuffer are then copied into the previous mask buffer and thus the old values are overwritten. All computations are repeated until the required number of iterations is reached. Notice that the first iteration operates on the first parameter set B_1, T_s and O_{dc} while successive iterations use the second parameter set B_2, T_s and O_{dc}. At the very first beginning the internal mask buffer is initialized with all pixel labels assigned an 'unchanged' label, i.e. filling the internal mask format with black color.

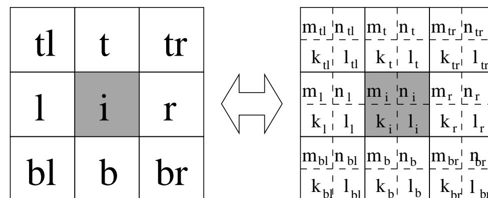
In the following sections a detailed description of the computation of M is given w.r.t. the processing areas top, $center$, and $bottom$.

Keeping in mind that each 2×2 block in the input image consists of the subsets k, l, m, and n, the direct neighbour on the right side of an element k_i is an element of the subset l. This neighbour has the same relative texture coordinates in the 1st storage level. For example, the texture coordinates of pixel k_i are $T_c = (u, v)$. The right neighbour has the same relative coordinates, but since the subsets are vertically positioned in the second storage level, their absolute coordinates in the latter differ. The full pixel correspondence between input pixels and the internal mask format is given by table (1).

$h-1$	$m_a(0,\frac{h}{8}{-}1)$	$n_a(0,\frac{h}{8}{-}1)$	$m_a(1,\frac{h}{8}{-}1)$	$n_a(1,\frac{h}{8}{-}1)$	$\cdots$	$m_a(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$n_a(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$h-2$	$k_a(0,\frac{h}{8}{-}1)$	$l_a(0,\frac{h}{8}{-}1)$	$k_a(1,\frac{h}{8}{-}1)$	$l_a(1,\frac{h}{8}{-}1)$	$\cdots$	$k_a(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$l_a(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
$\frac{3h}{4}+1$	$m_a(0,0)$	$n_a(0,0)$	$m_a(1,0)$	$n_a(1,0)$	$\cdots$	$m_a(\frac{w}{2}{-}1,0)$	$n_a(\frac{w}{2}{-}1,0)$
$\frac{3h}{4}$	$k_a(0,0)$	$l_a(0,0)$	$k_a(1,0)$	$l_a(1,0)$	$\cdots$	$k_a(\frac{w}{2}{-}1,0)$	$l_a(\frac{w}{2}{-}1,0)$
$\frac{3h}{4}-1$	$m_b(0,\frac{h}{8}{-}1)$	$n_b(0,\frac{h}{8}{-}1)$	$m_b(1,\frac{h}{8}{-}1)$	$n_b(1,\frac{h}{8}{-}1)$	$\cdots$	$m_b(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$n_b(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\frac{3h}{4}-2$	$k_b(0,\frac{h}{8}{-}1)$	$l_b(0,\frac{h}{8}{-}1)$	$k_b(1,\frac{h}{8}{-}1)$	$l_b(1,\frac{h}{8}{-}1)$	$\cdots$	$k_b(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$l_b(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
$\frac{h}{2}+1$	$m_b(0,0)$	$n_b(0,0)$	$m_b(1,0)$	$n_b(1,0)$	$\cdots$	$m_b(\frac{w}{2}{-}1,0)$	$n_b(\frac{w}{2}{-}1,0)$
$\frac{h}{2}$	$k_b(0,0)$	$l_b(0,0)$	$k_b(1,0)$	$l_b(1,0)$	$\cdots$	$k_b(\frac{w}{2}{-}1,0)$	$l_b(\frac{w}{2}{-}1,0)$
$\frac{h}{2}-1$	$m_g(0,\frac{h}{8}{-}1)$	$n_g(0,\frac{h}{8}{-}1)$	$m_g(1,\frac{h}{8}{-}1)$	$n_g(1,\frac{h}{8}{-}1)$	$\cdots$	$m_g(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$n_g(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\frac{h}{2}-2$	$k_g(0,\frac{h}{8}{-}1)$	$l_g(0,\frac{h}{8}{-}1)$	$k_g(1,\frac{h}{8}{-}1)$	$l_g(1,\frac{h}{8}{-}1)$	$\cdots$	$k_g(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$l_g(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
$\frac{h}{4}+1$	$m_g(0,0)$	$n_g(0,0)$	$m_g(1,0)$	$n_g(1,0)$	$\cdots$	$m_g(\frac{w}{2}{-}1,0)$	$n_g(\frac{w}{2}{-}1,0)$
$\frac{h}{4}$	$k_g(0,0)$	$l_g(0,0)$	$k_g(1,0)$	$l_g(1,0)$	$\cdots$	$k_g(\frac{w}{2}{-}1,0)$	$l_g(\frac{w}{2}{-}1,0)$
$\frac{h}{4}-1$	$m_r(0,\frac{h}{8}{-}1)$	$n_r(0,\frac{h}{8}{-}1)$	$m_r(1,\frac{h}{8}{-}1)$	$n_r(1,\frac{h}{8}{-}1)$	$\cdots$	$m_r(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$n_r(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\frac{h}{4}-2$	$k_r(0,\frac{h}{8}{-}1)$	$l_r(0,\frac{h}{8}{-}1)$	$k_r(1,\frac{h}{8}{-}1)$	$l_r(1,\frac{h}{8}{-}1)$	$\cdots$	$k_r(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$	$l_r(\frac{w}{2}{-}1,\frac{h}{8}{-}1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
3	$m_r(0,1)$	$n_r(0,1)$	$m_r(1,1)$	$n_r(1,1)$	$\cdots$	$m_r(\frac{w}{2}{-}1,1)$	$n_r(\frac{w}{2}{-}1,1)$
2	$k_r(0,1)$	$l_r(0,1)$	$k_r(1,1)$	$l_r(1,1)$	$\cdots$	$k_r(\frac{w}{2}{-}1,1)$	$l_r(\frac{w}{2}{-}1,1)$
1	$m_r(0,0)$	$n_r(0,0)$	$m_r(1,0)$	$n_r(1,0)$	$\cdots$	$m_r(\frac{w}{2}{-}1,0)$	$n_r(\frac{w}{2}{-}1,0)$
0	$k_r(0,0)$	$l_r(0,0)$	$k_r(1,0)$	$l_r(1,0)$	$\cdots$	$k_r(\frac{w}{2}{-}1,0)$	$l_r(\frac{w}{2}{-}1,0)$
	0	1	2	3	$\cdots$	$w-2$	$w-1$

Table 1: Conversion Matrix between the internal mask format and the input image.

A very important observation on the data formats and storage levels is that neighbouring pixels in the input image correspond to pixels in the internal mask format with equal relative texture coordinates only when they belong to the same 2×2 pixel block. Such block is shown in table (1) (light grey shaded cells), with the relative texture coordinates for each data subset being $(1, 1)$. Neighbouring pixel blocks are assigned indices as defined on the left figure:



4.5.1 Center Area

Based on the decision rule set in equation (18), M is computed as follows:

$$M = 2 \cdot \nu_B(q_i{=}c) + \nu_C(q_i{=}c)$$

Since $\nu_B(q_i{=}c)$ represents the number of pixels in hv-direction with a 'changed' label (pixel value=1), this term can be simplified to the summation of all pixels in hv-dir. The same rule applies to $\nu_C(q_i{=}c)$ and thus yields

$$M = 2 \cdot \sum_{i \,\epsilon\, (hv-dir.)} c_i + \sum_{j \,\epsilon\, (diag-dir.)} c_j,$$

whereby c_i is the color value (0 or 1) of a neighbour pixel in hv-direction, and c_j is the color value of a neighbour in $diag$-direction. Considering an output pixel in the center area of a subset k, denoted as k_i, the corresponding hv-neighbours are l_l, m_i, l_i, and m_b, whereas the $diag$-neighbours are n_l, n_i, n_b, and n_{bl}. Thus, c_i and c_j are derived as

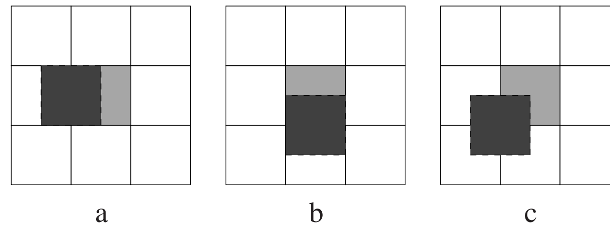
$$\sum_{i \,\epsilon\, (hv-dir.)} c_i = l_l + m_i + l_i + m_b,$$

$$\sum_{j \,\epsilon\, (diag-dir.)} c_j = n_l + n_i + n_b + n_{bl}.$$

Now M is computed by

$$\begin{aligned} M_k &= 2 \cdot (l_l + m_i + l_i + m_b) + (n_l + n_i + n_b + n_{bl}) \\ &= 2 \cdot \underbrace{(l_l + l_i)}_{a} + 2 \cdot \underbrace{(m_i + m_b)}_{b} + \underbrace{(n_l + n_i + n_b + n_{bl})}_{c} \end{aligned}$$

Without further optimization the above step would require 8 texture lookups. Due to the fact that pixels of the same subset in neighbouring pixel blocks are stored as direct neighbours in the internal mask format, i.e. their texture coordinates have a chessboard distance of 1, we make use of the hardware accelerated bilinear texture interpolation (see section 3.5). Hence, the summation terms a, b, and c can be simplified by applying the following texture lookups (the dark grey box represents the texel, which will be interpolated by the graphics hardware):



The a-term linearly interpolates between the pixel l_l and l_i. We denote this interpolation $l_d(-0.5, 0)$, which is equal to the texture lookup $tex(T_c) = tex(u - 0.5, v + 0 + h/8)$ within the internal mask format w.r.t. a considered pixel's coordinates (u, v). In the same way the b-term results in $m_d(0, -0.5)$ and the c-term yields $n_d(-0.5, -0.5)$. The interpolated texture lookups of the 3 terms return color values

$$\begin{aligned} val_a &= l_d(-0.5, 0) = \frac{1}{2}(l_l + l_i), \\ val_b &= m_d(0, -0.5) = \frac{1}{2}(m_i + m_b), \\ val_c &= n_d(-0.5, -0.5) = \frac{1}{4}(n_l + n_i + n_b + n_{bl}). \end{aligned}$$

It turns out that the quantitative relation between the 3 terms is already correct, but just the absolute scale has to be adapted. Thus, applying a global scale factor of 4 leads to

$$\begin{aligned} M_k &= 4 \cdot (2a + 2b + c) \\ &= 4 \cdot (val_a + val_b + val_c) \\ &= 4 \cdot (l_d(-0.5, 0) + m_d(0, -0.5) + n_d(-0.5, -0.5)) \end{aligned}$$

By utilizing the bilinear texture interpolation, we have now reduced the number of texture lookups from 8 to only 3. Indeed, this method increases processing performance distinctively. When repeating the above steps for the remaining subsets l, m, and n, we can summarize the computation of M for the $center$-area:

a b c

$$M_k = 4 \cdot \left(\underbrace{l_d(-0.5, 0)}_{a} + \underbrace{m_d(0, -0.5)}_{b} + \underbrace{n_d(-0.5, -0.5)}_{c} \right)$$

$$M_l = 4 \cdot \left(\underbrace{k_d(0.5, 0)}_{a} + \underbrace{m_d(0.5, -0.5)}_{b} + \underbrace{n_d(0, -0.5)}_{c} \right)$$

$$M_m = 4 \cdot \left(\underbrace{k_d(0, 0.5)}_{a} + \underbrace{l_d(-0.5, 0.5)}_{b} + \underbrace{n_d(-0.5, 0)}_{c} \right)$$

$$M_n = 4 \cdot \left(\underbrace{k_d(0.5, 0.5)}_{a} + \underbrace{l_d(0, 0.5)}_{b} + \underbrace{m_d(0.5, 0)}_{c} \right)$$

4.5.2 Top Area

When looking at the topmost row of a subset in the internal mask format, we observe that direct pixel neighbours do not necessarily reside in the same color channel. For instance, considering an element m_i in the red color channel of the subset m at location $m_r(1, h/8-1)$. This element corresponds to an output pixel at location $(2, h/4-1)$, as can be seen in table (1). The upper neighbour of this pixel maps to an element k_t, which belongs to the subset k but relies in the green color channel. Moreover, the neighbour's v-coordinate within the second storage level is not increased by 1 as expected, but decreased by $h/8-1$ based on the coordinate origin of the subset k.

The same observation can be made on an element n_i at location $n_r(1, h/8-1)$ in the red color channel, which has an upper neighbour l_t at location $l_g(1, 0)$ in the green color channel of the subset l.

We summarize the observations:

- Elements m_i and n_i in the red color channel have their upper neighbours in the green color channel, elements m_i and n_i in the green color channel have their upper neighbours in the blue color channel, elements m_i and n_i in the blue color channel have their upper neighbours in the alpha color channel.

- Elements m_i and n_i in the alpha color channel have no direct upper neighbour since they belong to the image border. Therefore only their values itself are used, which is equal to having a background neighbour.

- Upper neighbours of elements m_i and n_i have coordinates $(u, 0)$ and thus belong to the $bottom$-area of the considered subset.

The computation of M for the elements k and l is equal to the center area, thus

$$\begin{aligned} M_k &= 4 \cdot (l_d(-0.5, 0) + m_d(0, -0.5) + n_d(-0.5, -0.5)) \,, \\ M_l &= 4 \cdot (k_d(0.5, 0) + m_d(0.5, -0.5) + n_d(0, -0.5)) \,. \end{aligned}$$

For an element m we basically get

$$M_m = 2 \cdot \underbrace{(k_i + k_t)}_{M_{m,k}} + 2 \cdot \underbrace{(n_l + n_i)}_{M_{m,n}} + \underbrace{(l_{tl} + l_t + l_i + l_l)}_{M_{m,l}} .$$

Since processing is done on all 4 color channels in parallel, we can separate $M_{m,k}$ into 4 parts:

$$M_{m,k} = \begin{pmatrix} M_{m,k,r} \\ M_{m,k,g} \\ M_{m,k,b} \\ M_{m,k,a} \end{pmatrix} .$$

When looking at the red color channel $M_{m,k,r}$ we can formulate

$$M_{m,k,r} = k_{d,r}(0,0) + k_{d,g}(0, 1-\frac{h}{8}),$$

which describes the sum of the current pixel block's element k in the red channel and the upper neighbour k in the green color channel with a relative v-offset of $(1-h/8)$. The green and blue channels are equally specified:

$$\begin{aligned} M_{m,k,g} &= k_{d,g}(0,0) + k_{d,b}(0, 1-\frac{h}{8}), \\ M_{m,k,b} &= k_{d,a}(0,0) + k_{d,a}(0, 1-\frac{h}{8}). \end{aligned}$$

Since an element m in the alpha color channel belongs to the image's borderline, it does not have any upper neighbour and thus yields

$$M_{m,k,a} = k_{d,a}(0,0).$$

Combining the above formulas results in

$$M_{m,k} = k_d(0,0) \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} + k_d(0, 1-\frac{h}{8}).gbar \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} .$$

This notation means that from the right summation term we read out the color channels in the sequence g, b, r, a instead of r, g, b, a, as it would the general case. Such a 'swizzle'- operator is available on the GPU without time penalty. Any kinds of mixtures are possible, also reading out multiple times the same color channel. The vector $(1, 1, 1, 0)^T$ denotes the multiplication of the red, green and blue channel with 1 and the alpha channel with 0. Clearly, we only take the first channels into account while ignoring the alpha value. For simplification we rewrite

$$M_{m,k} = k_d(0,0) + k_d(0, 1-\frac{h}{8}).gba0,$$

which combines the vector multiplication with the swizzle-operator. The 0 in this new operator represents no color value at all and thus only the color channels g, b, a of $k_d(0, 1-\frac{h}{8})$ are added to the channels r, g, b of $k_d(0, 0)$. The result is written into the r, g, b channels of $M_{m,k}$, whereas in the a channel only the value $k_d(0, 0)$ is present.

Compared with the k and l elements, we now require two texture lookups instead of just one. However, since the top-area only consists of $w/2$ elements, the time penalty is rather small and thus negligible.

The computation of $M_{m,l}$ is quite similar to the above, with the only difference that we need to sum two elements in each color channel. Clearly, the term $l_l + l_i$ can be replaced by an interpolated texture lookup $l_d(-0.5, 0)$ within the same color channel while the term $l_{tl} + l_t$ can be replaced by a lookup $l_d(-0.5, 1-\frac{h}{8})$ in the next color channel. Notice that each interpolated lookup must be multiplied with a scale factor of 2 in order to retrieve the correct sum. Thus we formulate

$$M_{m,l} = 2 \cdot l_d(-0.5, 0) + 2 \cdot l_d(-0.5, 1-\frac{h}{8}).gba0.$$

By again utilizing linear interpolation, the term $M_{m,n}$ is derived:

$$M_{m,n} = 2 \cdot n_d(-0.5, 0).$$

Combining all three parts $M_{m,k}$, $M_{m,n}$ and $M_{m,l}$ we rewrite

$$\begin{aligned}
M_m &= 2 \cdot M_{m,k} + 2 \cdot M_{m,n} + M_{m,l} \\
&= 2 \cdot \left(M_{m,k} + M_{m,n} + \frac{M_{m,l}}{2} \right) \\
&= 2 \cdot \left(k_d(0,0) + k_d(0, 1-\frac{h}{8}).gba0 + 2 \cdot n_d(-0.5, 0) + l_d(-0.5, 0) + l_d(-0.5, 1-\frac{h}{8}).gba0 \right) \\
&= 4 \cdot \left(\frac{k_d(0,0) + k_d(0, 1-\frac{h}{8}).gba0}{2} + n_d(-0.5, 0) + \frac{l_d(-0.5, 0) + l_d(-0.5, 1-\frac{h}{8}).gba0}{2} \right).
\end{aligned}$$

Finally we summarize the computation of M for the top area

$$\begin{aligned}
M_k &= 4 \cdot (l_d(-0.5, 0) + m_d(0, -0.5) + n_d(-0.5, -0.5)) \\
M_l &= 4 \cdot (k_d(0.5, 0) + m_d(0.5, -0.5) + n_d(0, -0.5)) \\
M_m &= 4 \cdot \left(\frac{k_d(0,0) + k_d(0, 1-\frac{h}{8}).gba0}{2} + n_d(-0.5, 0) + \frac{l_d(-0.5, 0) + l_d(-0.5, 1-\frac{h}{8}).gba0}{2} \right) \\
M_n &= 4 \cdot \left(\frac{l_d(0,0) + l_d(0, 1-\frac{h}{8}).gba0}{2} + m_d(0.5, 0) + \frac{k_d(0.5, 0) + k_d(0.5, 1-\frac{h}{8}).gba0}{2} \right)
\end{aligned}$$

4.5.3 Bottom Area

For the computation of M in the bottom area, the same rules can be applied. The only difference is that instead of elements m and n, now the elements k and l need special treatment w.r.t. texture coordinates. The lower neighbour of an element k at position $k_a(u, 0)$ in the alpha color channel is an element m at position $m_b(u, h/8 - 1)$ in the blue color channel. Elements in the red color channel do not have a direct lower neighbour, because they belong to the image border. Thus, similar to the top area, only the element itself is taken into account.

For the bottom area we can formulate:

$$M_k = 4 \cdot \left(\frac{m_d(0,0) + m_d(0, \frac{h}{8}-1).0rgb}{2} + l_d(-0.5,0) + \frac{n_d(-0.5,0) + n_d(-0.5, \frac{h}{8}-1).0rgb}{2} \right)$$

$$M_l = 4 \cdot \left(\frac{n_d(0,0) + n_d(0, \frac{h}{8}-1).0rgb}{2} + k_d(0.5,0) + \frac{m_d(0.5,0) + m_d(0.5, \frac{h}{8}-1).0rgb}{2} \right)$$

$$M_m = 4 \cdot (k_d(0,0.5) + l_d(-0.5,0.5) + n_d(-0.5,0))$$

$$M_n = 4 \cdot (k_d(0.5,0.5) + l_d(0,0.5) + m_d(0.5,0))$$

4.5.4 Shader Programs

From the implementation point of view we have to decide how many Vertex- and Fragment Shader programs we need in order to compute M. A straightforward solution would be to implement 12 different Vertex- and also 12 Fragment Programs (4 subsets, each comprising 3 areas). However, when removing redundancy we can reduce the required number of Vertex Programs to 8 and for Fragment Programs to just 3. The underlying methodology is described in the next two paragraphs.

Vertex Programs (VP). A Vertex Program is executed for each drawn vertex, initiated by OpenGL or DirectX commands. For instance, a rectangle is fully defined by its 4 corner vertices. Each vertex can have attributes, whereas its texture coordinates are among the most important ones for our application. After applying some manipulation, i.e. 3D-transformations, texture coordinate generation, and 3D-2D-transformation, the output of the program is sent to a rasterizer unit on the graphics hardware. The resulting fragment, i.e. the drawn rectangle, is sent to the Fragment Shader units, whereby a Fragment Program is executed for each pixel covered by the considered fragment. Texture coordinates as well as pixel coordinates are linearly interpolated by the GPU.

When looking at the different M-formulations in the previous sections, we can observe that the texture coordinates for M_k in the center area are equal to those in M_k in the top area. Thus, both parts can be merged together into one Vertex Program. The following table lists all required Vertex Programs in the 4 subsets and the 3 areas.

	k	l	m	n
top	VP1	VP2	VP5	VP6
center	VP1	VP2	VP3	VP4
bottom	VP7	VP8	VP3	VP4

Fragment Programs (FP). A Fragment Program is executed for each rastered pixel covered by the considered fragment. It fetches texels from different textures, performs some computation and outputs a final color, i.e. the value M.

For the computation of M within the center area we observe that all 4 subsets undergo the same processing, which is basically the summation of 3 interpolated texture values. The differences in the texture coordinates are of no importance, since they are defined within the Vertex Programs. The same Fragment Program can be applied to M_k and M_l in the top area and to M_m and M_n in the bottom area. The remaining parts require each 5 texture lookups but because of different swizzle-operators, only those within the same processing area can be combined. This yields the following list of required Fragment Programs.

	k	l	m	n
top	FP1	FP1	FP2	FP2
center	FP1	FP1	FP1	FP1
bottom	FP3	FP3	FP1	FP1

4.6 The *SegConvert*-Shader

After the iterative MRF-computation, the internal change mask has to be backtransformed into a 1st storage level format, suitable for further processing. Hence, this last Shader converts the internal change mask into an RGB buffer, having the same dimensions as the input image, with white foreground and black background regions.

The Fragment Shader is executed for each considered ouput pixel, ranging from $(0,0)$ to $(w-1,h-1)$, whereby w is the width and h the height of the input image. Table (1) in section 4.4 lists the pixel correspondences between each output pixel in the 1st storage level. The dark grey boxes indicate the output pixel position. Elements of the 4 input data subsets k, l, m, and n are arranged in a 2×2 block (light grey block in the table), as described in section 4.4.

The Shader's task w.r.t. an output pixel is to fetch the color value from the internal mask format at the position corresponding to the current pixel, as listed in table (1), and transfer it to the output buffer. Although this processing step is rather simple, the implementation is somewhat tricky, since the input pixel coordinates for each consecutive output pixels differ. Clearly, the input texel for an output pixel location $(2,2)$ is $k_r(1,1)$. Based on the internal mask format (second storage level), this refers to a texture coordinate of $T_c = (1,1)$ in the red color channel k_r of the subset k. Thus, the output color is $out = tex(T_c) = tex(1,1)$. The neighbouring output pixel at position $(3,2)$ refers to an input texel with coordinates $(1,1)$ in the subset l, which has absolute texture coordinates $T_c = (1, 1 + \frac{h}{8})$ based on the internal mask format. The texture coordinates for an output pixel at location $(2,3)$ are $T_c = (1, 1 + \frac{2h}{8})$ and for location $3,3)$ they are $T_c = (1, 1 + \frac{3h}{8})$. We can observe that only the v-coordinate of the input texel varies within a 2×2 block of the output buffer.

We store the 4 different v-offsets in a 2×2 texture and activate the REPEAT-mode of the texture lookup. This mode ensures that when accessing a texel at position (u,v), the texture coordinates are converted into $T_{c,repeat} = (u \; mod \; 2, v \; mod \; 2)$.

$(0, \frac{2h}{8})$	$(0, \frac{3h}{8})$
$(0,0)$	$(0, \frac{h}{8})$

Now the flow of operation can be designed w.r.t. an output pixel location (u,v):

1. Fetch the v-offset from the 2×2 texture in REPEAT-mode,
2. Compute the texture coordinate for the input texel by $T_c = \left(\lfloor \frac{u}{2} \rfloor, \lfloor \frac{v}{2} \rfloor + tex_{offs}(T_{c,repeat})\right)$,
3. Fetch the input texel and transfer it to the output buffer.

5 User's Guide

This section gives a detailed description of the distributed software package, providing GPU-based foreground-background segmentation. The specific requirements are listed as well as the delivered files, and implementation details on the public methods of the core library class are given.

The software package includes the core library and an example program demonstrating the usage of the former. Our prototype runs under Linux operating system and supports the latest NVidia cards, such as the GeForce 6 series. The Vertex- and Fragment Shaders are written in Cg with OpenGL bindings. Access to the core library is handled in a thread-safe way, such that multi-threading is supported.

5.1 Requirements

Hardware Requirements. In order to run the provided segmentation algorithm on the GPU, the following hardware-specifications must be reached:

- OpenGL NV30 fragment profile is needed supporting pack/unpack-functions.
- Principally, there is no preferable manufacturer. In general, Nvidia's GPU's have more accurate bilinear interpolation facilities, while ATI's processors sometimes provide a faster memory interface as a result of higher clock-rates.
- The graphics unit must support rectangular textures with non-power-of-two size in each direction as well as the PBuffer extension.

For our prototype we used NVidia's GeForce 6800GT card with 256MB onboard texture memory.

Software Requirements. The following software specification must be met on the executing computer:

- While the PBuffer implementation supports multiple operating systems, our prototype software was written for Linux only. However, porting to Windows or Macintosh systems should be possible without loss of functionality.
- In order to run the application, at least an OpenGL context must be created, as can be seen in the example-application delivered with the library. Any kind of toolkit providing an OpenGL interface may be suitable, i.e. glut, Qt, gtk.
- OpenGL must be installed on the system with the drivers for the graphics board providing the latest OpenGL extensions supported.
- Cg (C for graphics), downloadable from the NVidia's website. The released version does directly effect the runtime, since later compiler support newer hardware capabilities and therefore produces more suitable assembler code. For our prototype, release 1.3 has been used.

Image requirements. Some restrictions on the input images are imposed as follows:

- The height must be a factor of 8, but at least 16 pixels. This directly results from our internal mask format, which divides the height into 4 equally sized subsets with height/8 and depth=4Byte. The *MRF-Iter*-Shader requires at least one pixel in the top-area and one pixel in the bottom-area. Since the center area is not necessarily used, a minimum of 2 pixels in each subset is required, thus yielding a minimum height of 16 pixels.
- The width must be a factor of 2. Again, this directly results from the internal mask format, which has half the width of the input image.

Memory Usage. The amount of required texture memory on the graphic card depends on the size of the input images. Based on the width w and height h, the following table lists all textures used and gives an overview of the memory consumption.

Internal Name	Description	width [px]	height [px]	depth [Byte]
texAvg	Average background image	w	h	3
texSrc	Input (foreground) image	w	h	3
texSeg	Segmentation output	w	h	3
texSeg_int	Internal Mask Format	$w/2$	$h/2$	4
texNormsFC	Storing $fore$ and $cross$, later M_{min}	w	h	4
texNormsB	Storing $back$	w	h	4
texConvert	v-offset matrix, used for the *SegConvert*-Shader	2	2	1

The overall texture memory required can thus be calculated by

$$TexMem = 18 \cdot w \cdot h + 4,$$

which leads to approx. 5.5MBytes for input images of size 640×480 pixels.

5.2 Distributed files

The distributed segmentation package includes the following files:

```
./
   CMakeLists.txt
   main.cpp
   norms.cg
   segment.cg
   segconvert.cg
   imgSrc.png
   imgAvg.png
   imgSeg.png
```

```
./lib/
   CMakeLists.txt
   allshaders_include.h
   libgpuseg.h
   libgpuseg.cpp
   pbuffer.h
   pbuffer.cpp
```

In the toplevel directory, the different Shaders (Vertex and Fragment) are implemented in the `.cg`-files, whereby in `norms.cg` the *NormCalc*-, *NormMixBack*-, and *NormConvert*-Shaders reside. The iterative MRF-computation, split into 8 Vertex- and 3 Fragment Shaders, is implemented in the `segment.cg` file. `segconvert.cg` holds the implementation of the *SegConvert*-Shader. The example `main.cpp`-file demonstrates the usage of the segmentation library by first reading an average input image (`imgAvg.png`) and an actual source image (`imgSrc.png`), performing the segmentation and finally writing the output into an `imgSeg.png` file. The example is compiled using CMake, whereas the `CMakeLists.txt` lists all dependencies and header files required. The core library is found under the `lib`-subdirectory and must be compiled separately, again by using CMake. The declarations of the core segmentation reside in `libgpuseg.h` and the implementation can be found in `libgpuseg.cpp`.

Two ways of linking the Shader code are provided:

1. Dynamically compiling the Shader files (`norms.cg`, `segment.cg`, and `segconvert.cg`),
2. Statically compiling the Shader code by including the file `allshaders_include.h`.

The dynamic linkage obviously has the advantage of online-adaptations to the Shader program without the necessity of recompiling the library itself. Each time the library code is executed, it reads the Shader files, compiles them by sending them to the Cg-compiler, and executes the returned assembler code on

the GPU. However, it is important to mention that the Shader files must be in the executable search path, i.e. in the same directory as the example application. Alternatively, a static linkage is providing a pseudo-compile-time version, which includes the `allshaders_include.h` file. In this header file, the same Shader listing as in the Shader files can be found, but now assigned to different string variables. These strings are compiled at runtime, but of course the library has to be rebuilt after modifications. Pathnames are no longer of importance in the static version. Switching between the dynamic and the static version is done by changing the 3rd codeline in the `libgpuseg.cpp` file:

```
#define DYNAMIC_CGFILES
```

The files `pbuffer.h` and `pbuffer.cpp` provide code for the PBuffer implementation, executable under Linux/Unix, Windows and Macintosh operating systems.

5.3 Library class description

The core library is implemented in the `libgpuseg.h` and `libgpuseg.cpp` files under the `lib-` subdirectory. It contains all code initializing the GPU, creating the textures and PBuffers, setting and modifying the parameter sets, performing the segmentation, providing methods for accessing the resulting segmentation texture, and finally safely cleaning up the reserved memory and instanciated data structures.

In the remaining section the following public methods are described in detail:

```
class CGpuSeg_Base
  CGpuSeg_Base();
  ~CGpuSeg_Base();
  int init(int w, int h, bool bPrintDebug=false);
  int set_background(unsigned char* imgBG, int depth=3);
  int set_foreground(unsigned char* imgFG, int depth=3);
  int segment(bool bWaitFinished);
  int segment(unsigned char* imgFG, bool bWaitFinished, int depth=3);
  int get_segmentation(unsigned char* imgSeg, int depth=1);

  void set_B1(float B1_);
  void set_B2(float B2_);
  void set_Ts(float Ts_);
  void set_Odc(float Odc_);
  void set_MRFiter(int MRFiter_);
  void set_Params(float B1_, float B2_, float Ts_, float Odc_,
                        int MRFiter_);

  float get_B1();
  float get_B2();
  float get_Ts();
  float get_Odc();
  int get_MRFiter();
  void get_Params(float &B1_, float &B2_, float &Ts_, float &Odc_,
                        int &MRFiter_);
  int get_width();
  int get_height();
  GLuint get_SegTex();
```

Functions declared as thread-safe first acquire a class-internal pthread-mutex before executing the requested code and finally releasing the same mutex.

`CGpuSeg_Base():` The constructor of the class creates a pthread-mutex, which allows a thread-safe access to all important internal functions.

`~CGpuSeg_Base():` The destructor cleans up the reserved memory by deleting the required texture memory, the OpenGL display lists, the PBuffers, the Cg-programs, and the pthread-mutex. This function is thread-safe.

`int init(`
`int w, int h, bool bPrintDebug=false):` Before the segmentation can be used, the class instance must be initialized by defining the width `w` and height `h` of the working images under consideration of the limitations given in section 5.1. Afterwards, the Cg-programs are read and compiled, the current existing OpenGL context initialized, the textures created, some useful OpenGL display lists defined, and finally the PBuffers created and initialized. When the parameter `bPrintDebug` is set to `true`, the assembly listing of the Shader programs is output. An alread initialized class instance may be re-initialized. The function is thread-safe and returns -1 in case of any error or 1 otherwise.

`int set_background(`
`unsigned char* imgBG, int depth=3):` With this method a new background image can be uyploaded, defined by the data buffer `imgBG`. The image must be of RGB-type (`depth=3`) with an optional alpha-channel (`depth=4`). The image's origin is on the top-left corner and pixels are stored in a row-wise manner. Each time a new background image is set, the internal working mask is cleared and the first segmentation run takes care of one-time computations, such as the *back* qualifiers. The function is thread-safe and returns -1 in case of any error or 1 otherwise.

`int set_foreground(`
`unsigned char* imgFG, int depth=3):` A new source image is provided to the class instance through this method, whereby the same rules for the image data as for the background image apply. The function is thread-safe and returns -1 in case of any error or 1 otherwise.

`int segment(`
`bool bWaitFinished):` This method initiates a foreground-background segmentation, based on the current source image and the average background image. The computation is split into 3 parts, each operating on a PBuffer object. Care has been taken for future improvements on floating point PBuffers by separating the PBuffers into one storing the image qualifiers (this could be a float-buffer in future versions) and another storing the internal mask format. Notice that in case of two different PBuffer objects, a context switch has to occur, which might slow down the overall performance. If the parameter `bWaitFinished` is set `true`, the function returns only when the GPU has finished the computations. This is only useful when the user's application requires knowledge about ready-to-operate GPU. The function is thread-safe and returns -1 in case of any error or 1 otherwise.

`int segment(`
`unsigned char* imgFG, bool bWaitFinished, int depth=3):` This method combines the functions `set_foreground` and `segment` by first setting the foreground image and then performing the segmentation. The function is thread-safe and returns -1 in case of any error or 1 otherwise.

`int get_segmentation(`
`unsigned char* imgSeg, int depth=1):` In case the final segmentation result is not needed for further compuations on the GPU, the binary segmentation image might be downloaded to a buffer, pointed to by `imgSeg`. The buffer memory must already be allocated on beforehand and can have a

pixel-depth of 4 Bytes (RGBA), 3 Bytes (RGB) or 1 Byte, whereby the latter is the fastest one. Foreground regions are assigned a color value of 255 and background regions have value 0. The readout of the texture memory, storing the segmentation, ensures that all previous OpenGL commands have finished and thus the internal instruction pipeline is emptied. Clearly, when running the segmentation, the flag `bWaitFinished` must not be set in case the `get_segmentation` is called successively. The function is thread-safe and returns -1 in case of any error or 1 otherwise.

`void set_B1(`
`float B1_):` This method modifies the compactness value B of the first parameter set. Typical values range between 1 and 3. The higher B_1 is, the lower the trigger level for a pixel is to be assigned a foreground segment, dependent on the local neighbourhood. Since the first parameter set is used for solving compactness in consecutive image frames, a B-value too high might result in a temporal smearing effect of foreground regions. The function is thread-safe.

`void set_B2(`
`float B2_):` This method modifies the compactness value B of the second parameter set. Typical values range between 100 and 700, dependent on the image contrast. Higher values result in more compact regions and thus smoother foreground areas. The function is thread-safe.

`void set_Ts(`
`float Ts_):` This method modifies the static threshold of both parameter sets. In practice, lower values result in highly cluttered foreground regions, whereas higher values might not detect foreground correctly. However, small outliers in the first segmentation are eliminated by the initial run on the first parameter set, whereas occluded foreground is usually better handled by taking darkness compensation into account as well as the iterative compactness computation using the second parameter set. Typical values range between 50 and 500. The function is thread-safe.

`void set_Odc(`
`float Odc_):` This method writes the darkness offset of both parameter sets. Higher values benefit better segmentations on areas with dark foreground and bright background. Typical values range between 1000 and 30000. The function is thread-safe.

`void set_MRFiter(`
`int MRFiter_):` This method modifies the number of iterations during the MRF-computation. Higher values result in smoother and more compact segmentations, but take more computation time. Typical values range between 4 and 8. The function is thread-safe.

`void set_Params(`
`float B1_, float B2_, float Ts_, float Odc_, int MRFiter_):` This method combines the above functions by setting both parameter sets at once. The function is thread-safe.

`float get_B1():` This thread-safe function returns the currently set compactness value for the first parameter set.

`float get_B2():` This thread-safe function returns the currently set compactness value for the second parameter set.

`float get_Ts():` This thread-safe function returns the currently set static threshold value.

`float get_Odc():` This thread-safe function returns the currently set darkness offset.

`int get_MRFiter():` This thread-safe function returns the currently set number of MRF-iterations.

`void get_Params(`
`float &B1_, float &B2_, float &Ts_, float &Odc_, int &MRFiter_):` This thread-safe function reads both parameter sets at once and stores the result in the submitted reference variables.

`int get_width():` This thread-safe function returns the currently set width of the working images.

`int get_height():` This thread-safe function returns the currently set height of the working images.

`GLuint get_SegTex():` The segmentation result resides in a texture memory on the graphics hardware. In order to allow further GPU-applications to operate on this segmentation, the corresponding texture identifier is required, which can be retrieved by this function. Notice that the texture is only valid during lifetime of the class instance. The function is thread-safe.

5.4 Example Program

The following example program in pseudo-code demonstrated the usage of the core library:

```
Create_OpenGL_Context // QtGL, gtk, glut
CGpuSeg_Base *pSeg = new CGpuSeg_Base();
imgBG = Load_Background_RGBImage;
pSeg -> init(width,height);
pSeg -> set_background(imgAvg);
pSeg -> set_Params(...);
...
imgFG = Load_Foreground_RGBImage;
pSeg -> segment(imgFG, false);
if (Further_GPU_Applications)
  SegTex_id = pSeg -> get_SegTex();
  Execute_Further_GPU_Applications
if (Store_Segmentation)
  imgSeg = Create_Empty_Buffer
  pSeg -> get_segmentation(&imgSeg);
...
delete pSeg;
Destroy_OpenGL_Context
```

First an OpenGL drawing context has to be created, in case none is already defined. Then, after creating an instance of the segmentator class, it is initialized with the width and height of the current working images. The loaded background image is transfered to the GPU and the parameter sets are defined afterwards. On each newly loaded source image the segmentation is performed. If further GPU applications operate on the segmentation result, successive OpenGL instructions may be sent to the graphics hardware without explicitely waiting for the GPU finished all previous commands. The OpenGL state-machine guarantees that the texture memory is only read when previous write operations successively completed. Additionally, the segmentation result could be stored in a dedicated buffer by reading out the texture buffer.

6 Results

The impact of the number of iterations during the MRF computation to the segmentation result is shown in figure 9, whereby the prior frame is completely black. Without iterating at all, some outliers are detected, depending on the threshold level. With increasing number of iterations the segmented regions become more smooth and compact and isolated small foreground pixels are turned into background. The parameters used are $T_s = 310$, $O_{dc} = 5800$, $B_1 = 2$, and $B_2 = 200$. We observed that robustness of segmentation wrt. the static threshold is increased with the MRF iterations.

Figure 9: Segmentation result with different number of iterations j=0,2,4,6 (from left to right) during the MRF-computation.

As the algorithm is mainly designed for indoor applications, we recorded some image sequences in our 3D-scanning setup (see [10]). Accounting for motion blur elimination and thus forcing low exposure times, the grabbed images suffer from very low contrast and high image noise. Therefore darkness compensation is essential for acceptable segmentations even the foreground object and the background are nearly similar in luminance. Figure 10 demonstrates the importance of darkness compensation. All three segmentations use the same parameter set. The fourth image is the result of an optimized CPU-version of the segmentation without MRF-iterations, that is darkness compensation added to the algorithm of Mester. Additionally we tested the algorithm in outdoor environments (see figure 11) and still get good results.

To our knowledge, our GPU-based implementation outperforms any other existing image segmentation wrt. runtime. Table (2) summarizes the measured timings dependent on the number of MRF-iterations. The first two columns are based on images of size 640×512, whereas the rest operates on 640×480 images. The following cases are distinguished:

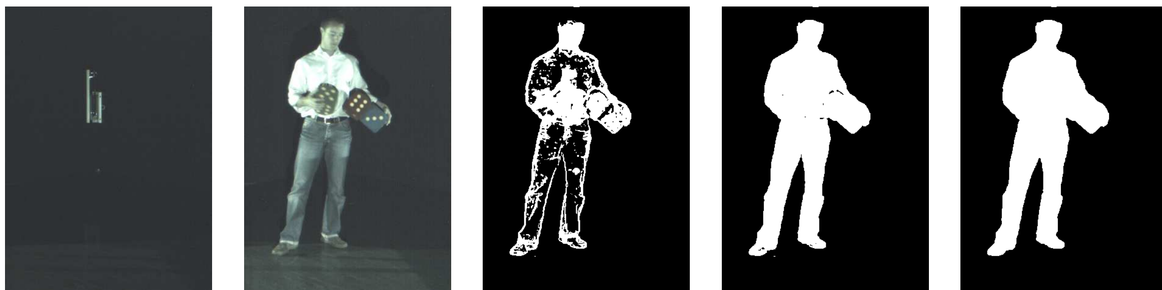


Figure 10: From left to right: BG-image, FG-image, segmentation without darkness compensation (d.c.), CPU-based segmentation with MRF and d.c., GPU-based iterative segmentation with MRF, d.c. and 8 iterations.

$T_{1000,ext}$ is the average runtime of 1000 segmentations without changing the source image. Since each segmentation is performed without explicitely waiting for the result, the internal instruction pipeline of the GPU is always filled. Thus, this mode simulates the usage of our algorithm in combination with GPU-based post-processes. The ext-index denotes the measurment including the thread-safe mutex-locks and -unlocks as well as the PBuffer-context-switches.

$T_{1000,int}$ is similar to the above, except the time is measured without the mutex-operations and without the PBuffer-context-switches. Indeed, this simulates the usage of our algorithm in combination with post-processing GPU-based applications, using the same PBuffer. It turns out that a context-switch typically takes about 400μs.

$T_{1,ext}$ is the time needed for one segmentation run excluding the upload time for the source image. Due to the fact that the internal pipeline of the GPU is no longer optimally filled because the segmentation is performed with `bWaitFinished=true`, these times are slightly larger than in the above pipelined cases.

$T_{1,ext,up}$ again is the time needed for one segmentation run, but now including the image upload. Indeed, this time is highly dependent on the operating system and the graphics hardware driver used.

$T_{1,ext,up+dn}$ measures the time needed for one segmentation run including image upload and the segmentation result download via `get_segmentation`. Again, this time strongly depends on the operating system and the drivers used.

iter	$T_{1000,ext}$	$T_{1000,int}$	$T_{1000,ext}$	$T_{1000,int}$	$T_{1,ext}$	$T_{1,ext,up}$	$T_{1,ext,up+dn}$
	640×512		640×480				
0	2.92	2.88	2.54	2.51	3.01	4.8	6.2
1	3.15	3.10	2.78	2.75	3.25	5.1	6.4
2	3.37	3.33	3.01	2.97	3.50	5.3	6.6
3	3.62	3.57	3.24	3.20	3.73	5.5	6.8
4	3.82	3.76	3.48	3.44	3.94	5.7	7.0
5	4.04	3.99	3.71	3.67	4.2	6.0	7.2
6	4.26	4.21	3.94	3.89	4.4	6.2	7.4
8	4.69	4.66	4.41	4.38	4.82	6.6	7.9
10	5.16	5.11	4.87	4.84	5.33	7.1	8.3

Table 2: Time measurements for the GPU-based segmentation w.r.t. the number of MRF-iterations. The results are given in ms.

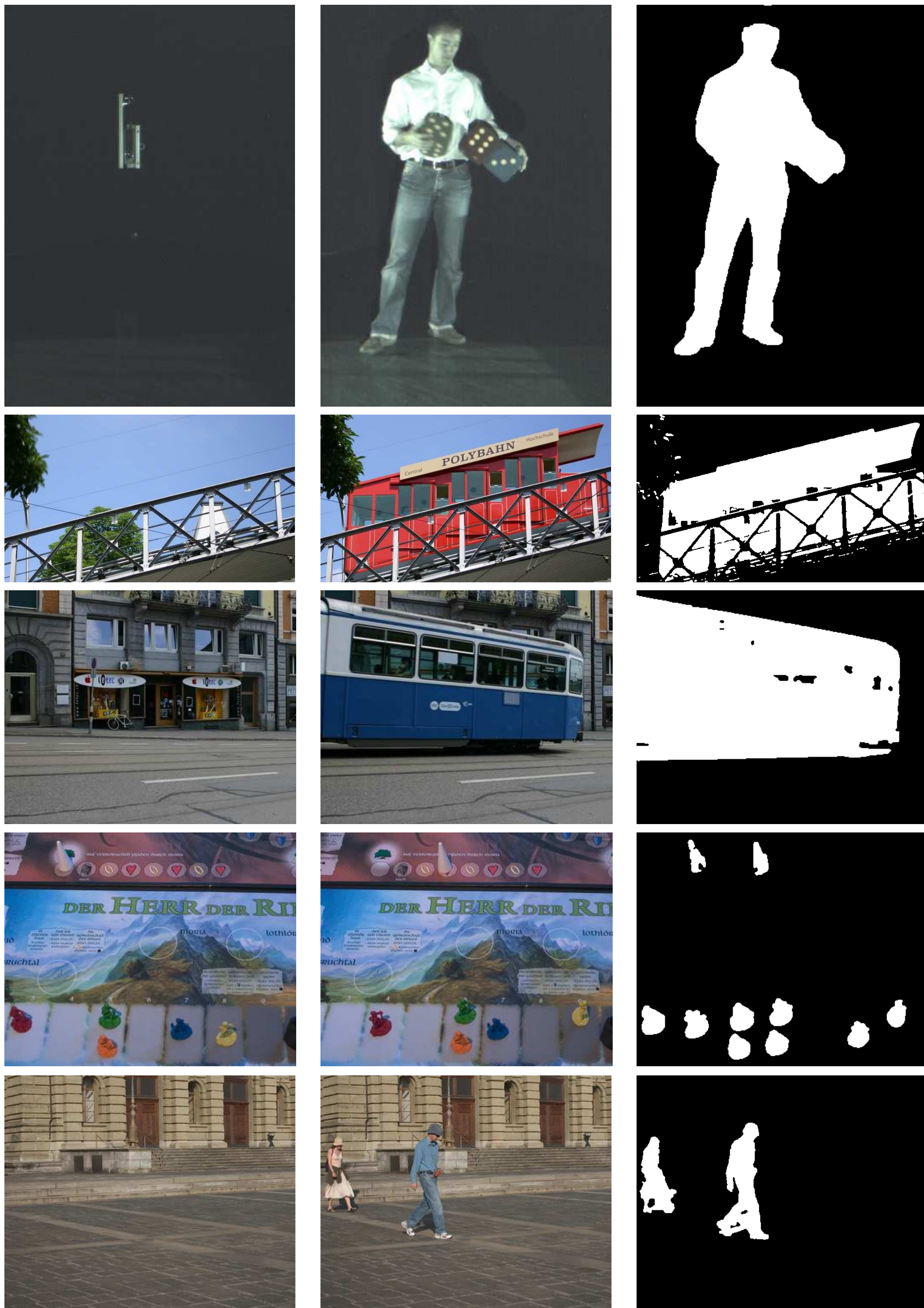


Figure 11: Segmentation results in indoor and outdoor environments. From left to right: BG-image, FG-image, Segmentation usind darkness compensation and 8 MRF iterations on the GPU. Processing time for each image is approx. 3ms

References

[1] R. Mester, T. Aach, L. Dümbgen, "Illumination-Invariant Change Detection Using a Statistical Colinearity Criterion", *Proc. 23rd DAGM Symp.*, 2001.

[2] T. Aach, A. Kaup, "Bayesian algorithms for adaptive change detection in image sequences using Markov random fields', *Signal Processing: Image Communication 7(2)*, 1995.

[3] T. Aach, A. Kaup, R. Mester, "Change detection in image sequences using Gibbs random fields", *IEEE Int. Workshop Intell. Signal Processing Com. Sys.*, 1993.

[4] C. Stauffer, W.E.L. Grimson, "Adaptive Background Mixture Models for Real-Time Tracking", *Proc. CVPR*, 1999.

[5] N. Friedman, S. Russell, "Image Segmentation in Video Sequences: a Probabilistic Approach", *Proc. 13th Conf. on Uncertainty in Artificial Intelligence*, 1997.

[6] R. J. Radke *et al.*, "Image Change Detection Algorithms: A Systematic Survey", *Image Processing 14 (3)*, 2005.

[7] R. Kehl, L. Van Gool, "Real-time Pointing Gesture Recognition for an Immersive Environment", *Proc. 6th IEEE Intl. Conf. on Aut. Face and Gesture Recog.*, 2004.

[8] W. Matusik, C. Bueler, L. McMillan, "Polyhedral visual hulls for real-time rendering", *Proc. EGRW*, 2001.

[9] M. Li, M. Magnor, H.-P. Seidel, "Hardware-Accelerated Visual Hull Reconstruction and Rendering", *Graphics Interface*, 2003.

[10] A. Griesser, T.P. Koninckx, L. Van Gool, "Adaptive Real-Time 3D Acquisition and Contour Tracking within a Multiple Structured Light System", *Proc. 12th Pacific Graphics*, 2004.

[11] N. Cornelis, L. Van Gool, "Real-Time Connectivity Constrained Depth Map Computation Using Programmable Graphics Hardware", *Proc. CVPR*, 2005.

[12] General-Purpose Computation Using Graphics Hardware, http://www.gpgpu.org.