# Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware

Ruigang Yang and Greg Welch

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina

**Abstract.** We present a novel use of commodity graphics hardware to perform real-time image segmentation and image morphology operations. Our preliminary results show a performance increase of over 30% using an nVidia GeForce4 when compared to an implementation using Intel MMX optimized code on a 2.2 Ghz Intel P4 CPU.

## 1 Introduction

Modern commodity graphics hardware systems offer increasingly powerful performance in terms of speed and programmability. In this paper, we introduce the use of such graphics hardware to perform real-time image segmentation—one of the central tasks in computer vision. While there have been many noteworthy hardware advances, we make use of *register combiner* and blending technology in particular. A register combiner, first proposed and implemented by nVidia [7], is a texture compositing unit that can perform fixed function arithmetic on a per-pixel basis [6]. The blending functions are used to control how the *source* color (new color fragment from the OpenGL pipeline) and the *destination* color (usually the pixels in the frame buffer) are combined. Using these features, we can perform image segmentation and subsequent image morphology operations, completely within the graphics hardware. Our basic implementation, running on an nVidia GeForce4, is 3–5 times faster for image thresholding and over 30% faster for image morphological operations, compared to a highly optimized software implementation running on a 2.2 Ghz Intel P4 CPU. Our method, which is limited by the AGP bus bandwidth, offers the greatest advantage for applications where the segmented images are to be used as textures for subsequent rendering.

## 2 Methods

The task of image segmentation is to classify pixels as belonging to foreground or background objects [8]. A common approach is to use pixel thresholding. Such techniques, which make decisions based on local pixel information, can be effective when the pixel intensities are clearly above or below the threshold [2]. However things are rarely so clear, in particular around object boundaries. Furthermore, thresholding is quite sensitive to image noise. As such, thresholding is usually followed by morphological operations to smooth the object boundaries and remove spurious pixels. We present here a method to implement these two steps completely on the graphics hardware.

### 2.1 Segmentation

To implement image segmentation we use the *register combiners* available in nVidia's graphics board to compute the squared difference between pixels, then use the *alpha test* to perform pixel thresholding. We use two stages of register combiners to compute the squared difference between an input and background image as follows. Let $C_i$ be the color of a input pixel, and $C_b$ be the color of the corresponding background pixel. We want to compute

$$\Delta = ||C_i - C_b||^2 = (C_i - C_b) \cdot (C_i - C_b). \quad (1)$$

We render the two images using orthogonal projection and program the register combiner to compute the squared difference $\Delta$ and store in the alpha channel of the frame buffer. The pseudo code to set up the register combiner is shown in Algorithm 1.

**Algorithm 1** Set up the register combiners to compute the squared difference

**Require:** The input image is stored in texture unit 0, while the background image is stored in texture unit 1.

```
{ \\FIRST stage of general combiner,
    \\compute tex0-tex1
    spare0.rgb = tex0 + signed_invert(tex1);
}

{\\ SECOND stage of general combiner,
    \\compute dot product
    state0.rgb = dot(spare0, spare0);
}

\\final combiner stage,
\\output the delta in the alpha channel
out.rgb = tex0, out.alpha = spare0.b;
```

Then we enable the standard alpha test to only accept pixels with a high enough alpha value (the squared difference). The exact value of the threshold is a user defined parameter. The resulting frame buffer contains the segmented image.

The entire segmentation process only requires two texture units and two register combiner stages, which are available even on relatively inexpensive commodity graphics boards. Note that it is possible to use a per-pixel threshold based on the local statistics of the background image. This requires an additional texture unit and an additional register combiner stage. The standard deviation of the background is stored in the extra texture unit, and the extra register combiner stage is used to compare $\Delta$ with the standard deviation of the background. Typically if $\Delta$ is greater than three times the standard deviation (the standard "$3\sigma$" rule), the pixel will be classified as a foreground pixel. The alpha test is not necessary using this approach.

### 2.2 Image Morphology

The morphological operations [8] we implemented are simple erosion and dilation of an image. The erosion operation is defined as follows: the output pixel is set to the *minimum* of the corresponding input pixel and its 8 neighboring pixels. If the input image is a color image, then each channel is treated independently. The dilation operation is very similar except it sets the output pixel to the *maximum* of the corresponding input pixel and its 8 neighboring pixels.

OpenGL Version 1.2 supports new blending functions to determine how the source and destination colors are combined. Of particular interest are GL_MIN and GL_MAX, which implement a component-wise minimum (or maximum) of the source and destination colors. To achieve dilation we set the blending function to GL_MAX, then shift and redraw the image 8 times, each with a one-pixel offset in a different direction. To achieve erosion we use GL_MIN. The idea of shifting images has been used to achieve a number of effects, most notably scene antialiasing and depth of field [9].

While the above algorithm is relatively easy to implement, using eight passes is quite slow, even on the latest graphics board. We use multi-texture support to combine multiple passes into one. In our current implementation, we use two texture units to reduce the rendering passes by one half. Because the texture blending function does not support min-max style blending, we have to program the register combiner to do the trick. We use the multiplex function in the register combiner to select the minimum (or maximum) of two pixels. Note that the register combiner cannot select the three RGB channels independently. The code presented in Algorithm 2 selects the pixel based on the alpha value. It is designed to work with the image segmentation routine in Algorithm 1, where the squared difference is stored in the alpha channel. It will also work with single channel (monochrome) images.

**Algorithm 2** Select the minimum

```
{   // FIRST combiner stage;
    // spare0 = tex1 - tex0 + 0.5
      spare0.alpha = tex1 - half_bias(tex0);
}

{ // SECOND combiner stage
    // select the color with the smaller alpha;
    // spare0 =(spare0.alpha < 0.5)?(tex1):(tex0);
        spare0.rgb = mux();

    // select the smaller alpha value
    spare0.alpha = mux();
}

{ // final output
    out = spare0;
}
```

Note that if we want to perform dilation (selecting the maximum value), we only need to swap texture 0 and texture 1 in the first combiner stage.

The imaging subset defined in the OpenGL Specification 1.3 [1] includes a set of filtering functions, however we choose not to use these for two reasons. First, image morphology applied to *intensity* images is not a linear operation. Second, even if we restrict ourself to binary images where dilation can be performed in a linear fashion by applying a $3 \times 3$ all-ones filter, the filter functions (`glConvolutionXX`) are not hardware accelerated in any existing commodity graphics board. We have tested the filter functions and they are over ten times slower than the texture-shifting method.

## 3 Results

We implemented our methods using OpenGL under Microsoft Windows 2000. For comparison we also implemented a software-only version using Intel's Image Processing Library (IPL) [4]. IPL provides a set of low-level image manipulation functions in the form of standard DLLs and static libraries. Functions in IPL are optimized for each of Intel's Pentinum class processors, allowing the users to obtain optimal performance with a specific processor. While IPL has direct support for image dilation and erosion, we are unaware of any background segmentation routine. As such we used the IPL routines `iplSubtract()` and `iplSquare()` to compute the squared difference, and wrote a tight loop to perform the thresholding. We tested both the graphics hardware implementation and the software implementation on a number of processors and graphics board.

We first show some qualitative results using the graphics hardware implementation. Figure 1 shows thresholding results with different intensity thresholds in the second row, and the corresponding final smoothed images (two erosion iterations, three dilation iterations, followed by one final erosion iteration) in the third row. Figure 2 presents color and binary examples after some morphological operations using our hardware implementation. Results from the software implementation are *identical*.

We tested our implementation on four different PCs, timing the image segmentation and morphological operations with input images of different sizes. For each image, we ran the segmentation followed by five erosions and dilations, using both the software and hardware implementations. We report the average times for 50+ repetitions in Table 1 and Figure 3. Our experiments indicate that (a) the time elapsed increases almost linearly with respect to the size of the image, and (b) the performance of the graphics hardware implementation is almost completely independent of the CPU speed. (Compare the numbers on the first PC and the third PC.) In Figure 4 we compare the timing for different combinations of CPUs and graphics boards using VGA image resolution. Our hardware version of the image segmentation is an obvious winner, over five times faster than our software version.[1] For image morphology operations, the hardware version using multi-textures on a GeForce4 is over 30% faster than the software version on a 2.2 Ghz CPU.

[1] We suspect that the threshold loop in the software implementation could be improved by writing in assembly code.

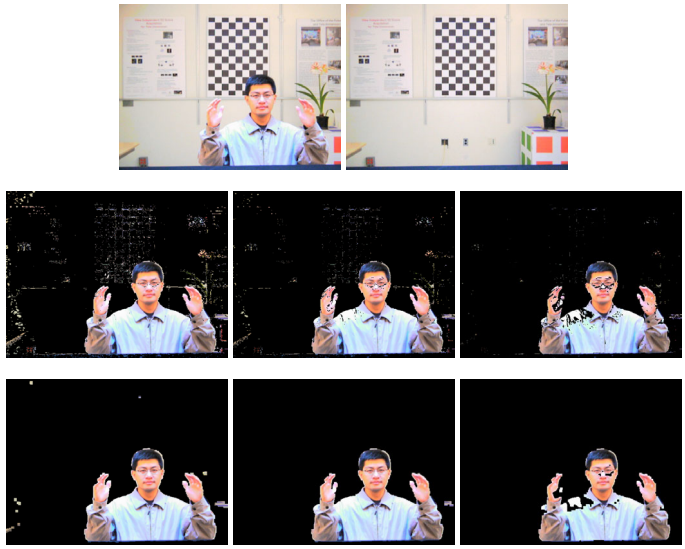


**Fig. 1.** Image Segmentation. The input and background images are shown on the left and right (respectively) of the top row. The segmented images with thresholds of 10, 15, and 20 (from left to right) are shown in the second row. The corresponding images after morphological operations are shown in the third row.

We also measured the time to load images from main memory to the video memory, butt did not include this in our previous comparisons since the image had to be uploaded to the graphics board to be rendered. Doing segmentation in the hardware actually reduces the memory bandwidth requirement by 25% since there is no need to send the alpha channel. We plot the results in Figure 5. The normalized curve is the transfer time divided by the number of pixels and then multiplied by $640 \times 480$ (using VGA resolution as the baseline). From the plot we can see that the transfer rate is almost independent of the CPU clock rate, and linear with respect to the number of pixels (the normalized curve is flat). Our measurements confirm that the AGP bus is still the bottleneck for displaying live images. The advantage of doing processing on the graphics hardware will be diminished if the image data has to be read back for future analysis.

## 4 Future Work

While our existing implementation already shows a performance increase of over 30% using an nVidia GeForce4 compared to an implementation using Intel MMX optimized code on a 2.2 Ghz Intel P4 CPU, it can be improved in both quality and performance.

First, It is possible to convert from RGB space to a different color space and perform the segmentation there [3]. For example, we can convert to a YCrCb color space and only use the Cr or Cb channel for segmentation. This approach reduces the effect of luminance changes and makes the segmentation less susceptible to shadows. With the help of the register combiner, we can segment the image in a different color space on


**Fig. 2.** Color and binary images after morphological operations. The top row shows the original images. The bottom row shows the results of hardware-based morphological operations on each original image. The first and third images show the results of three dilation iterations, the second and fourth images show the results of three erosion iterations (on the original images). Results from the software implementation are *identical*.

the fly. Here we show an example to use the Cr channel for segmentation. The formula to convert from RGB to YCrCb is [5]

$$
\begin{aligned}
Y &= 0.257 * R + 0.504 * G + 0.098 * B + 16 \\
Cr &= 0.439 * R - 0.368 * G - 0.071 * B + 128 \\
Cb &= -0.148 * R - 0.291 * G + 0.439 * B + 128
\end{aligned}
$$

We only need to change the code in the first register combiner stage (in Algorithm 1) to the following:

```
const0 = {0.439, 0.368, 0.071};
spare0.rgb = tex0*const0 + signed_invert(tex1)*const0;
```

This is possible because the register combiner actually computes $E = A*B+C*D$ at each stage. The code in Algorithm 1 is implicitly using a `const0` of $\{1, 1, 1\}$.

It is also possible to provide adaptation of the background differencing model to changes of lighting conditions and background scenes. The static background image can be replaced with running average of the input images with foreground excluded. That only requires one extra rendering pass.

With respect to performance, we see more opportunities for improvement even with the *existing* graphics boards. For example, for backward compatibility reasons, we only used two texture units in our implementation, while the GeForce4 card has four texture units. Using all of them can further reduce the number of rendering passes. The other place for improvement is to use the *render-to-texture* method, which has recently been made available in OpenGL. We can render the shifted images into two textures in alternating fashion, thus avoiding copies from the frame buffer to texture memory.

| | 1.2Ghz GF4 | | | 1.5Ghz GF2 | | | 1.9Ghz GF4 | | | 2.2Ghz GF3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Seg. | Morph | Xfer | Seg. | Morph | Xfer | Seg. | Morph | Xfer | Seg. | Morph | Xfer |
| IPL | 4.09 | 10.94 | | 1.55 | 12.52 | | 1.78 | 10.17 | | 0.94 | 8.2 | |
| **QVGA** HW | 0.29 | 9.95 | 1.78 | 0.9 | 26.28 | 1.85 | 0.82 | 8.98 | 1.73 | 0.38 | 15.32 | 1.75 |
| HW MT | | 6.88 | | | 17.9 | | | 6.57 | | | 10.45 | |
| IPL | 20.46 | 58.47 | | 6.12 | 48.01 | | 6 | 38.25 | | 4.89 | 34.84 | |
| **VGA** HW | 1.2 | 39.19 | 7.12 | 3.66 | 108.62 | 7.27 | 1.2 | 39.45 | 6.66 | 1.63 | 61.84 | 6.99 |
| HW MT | | 27.88 | | | 76.04 | | | 26.47 | | | 42.39 | |
| IPL | 31.96 | 87.49 | | 9.42 | 73.87 | | 9.59 | 60.04 | | 7.33 | 52.97 | |
| **SVGA** HW | 1.91 | 63.16 | 10.96 | 5.37 | 166.28 | 11.31 | 1.91 | 62.36 | 10.34 | 2.54 | 96.86 | 10.76 |
| HW MT | | 43.72 | | | 114.72 | | | 41.6 | | | 66.47 | |
| IPL | 41.95 | 146.95 | | 16.25 | 118.75 | | 15.59 | 97.48 | | 11.93 | 85.81 | |
| **XGA** HW | 3.14 | 109.85 | 17.74 | 8.56 | 270.34 | 18.49 | 3.09 | 105.51 | 16.91 | 4.19 | 158.7 | 17.69 |
| HW MT | | 72.89 | | | 186.07 | | | 70.86 | | | 109 | |

**Table 1.** Performance data using different methods on four PCs. The numbers are in milliseconds. "HW MT" stands for hardware implementation using multi-texture functions. The numbers for image morphology ("Morph") reflect the total time for ten iterations.

## References


1. OpenGL Specification 1.3, August 2001. http://www.opengl.org/developers/documentation/version13/glspec13.pdf.
2. Bernard Chazelle. Application challenges to computational geometry: Cg impact task force report. Technical Report TR-521-96, Princeton University, April 1996.
3. I. Feldmann, S. Askar, N. Brandenburg, and O. Schreer P. Kauff. Real-Time Segmentation for Advanced Disparity Estimation in Immersive Videoconference Applications. In *Proceeding of WSCG 2002, 10th Int. Conference on Computer Graphics, Visualization and Computer Vision*, Plzen, Czech Republic, Feb. 2002.
4. Intel. Image Processing Library. http://www.intel.com/software/products/perflib/ipl/.
5. Keith Jack. *Video Demistified: A Handbook for the Digital Engineer*. Brooktree, first edition, 1993.
6. Mark J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. In *Game Developers Conference 2000*, San Jose, California, March 2000.
7. Nvidia. http://www.nvidia.com.
8. R.C.Gonzalez and R.E.Woods. *Digital Image Processing*. Prentice Hall, second edition, 2002.
9. Mason Woo, Jackie Neider, and Tom Davic. *OpenGL Programming Guide*. Addison-Wesley, second edition, 1996.

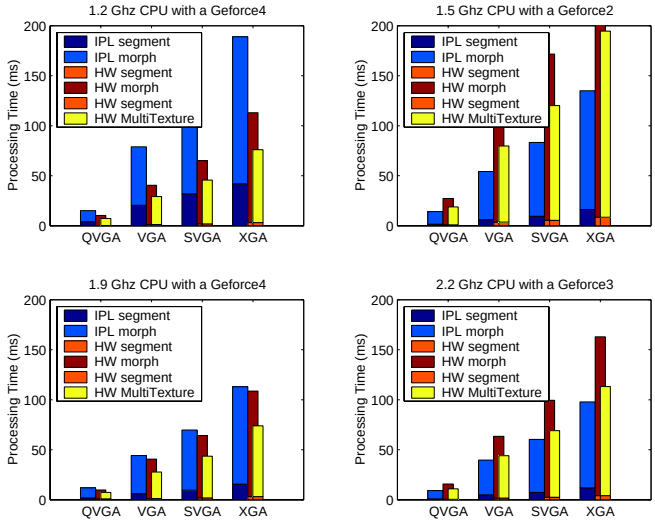
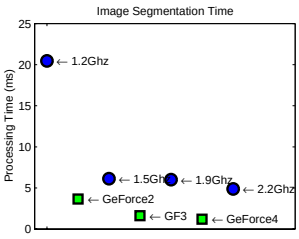



**Fig. 3.** Visual plot of the performance data



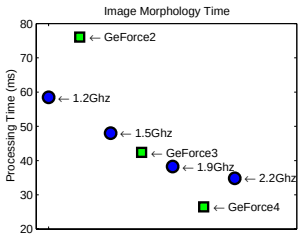




**Fig. 4.** Performance data using different methods on four PCs

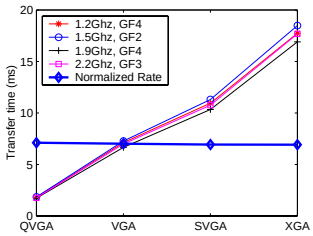



**Fig. 5.** Texture transfer time