Accelerating Image Processing in Flash using SIMD Standard Operations

Chamira Perera†, Daniel Shapiro‡, Jonathan Parri‡, Miodrag Bolic‡, Voicu Groza‡
†*Systems and Computer Engineering, Carleton University*
‡*Computer Architecture Research Group, University of Ottawa*
†*cperera@sce.carleton.ca,* ‡*{dshap092, jparr090, mbolic, groza}@site.uottawa.ca*

***Abstract*— Flash applications have played an integral role in shaping the interactivity of the Internet. Desktop Flash applications feature vector-based processing such as image and video processing to enhance the user experience. In response to these needs, Adobe has added graphics card based acceleration for vector processing in Flash applications starting with Flash Player 10. This solution is limited to computer systems that have the proper graphics card. In this paper, we investigate the possibility of making explicit use of Single Instruction Multiple Data instructions, specifically SSE in the Intel x86-64 platforms, to accelerate vector operations in a Flash application. We also discuss certain limitations of the Flash virtual machine. The data reveals that a 90-92% speedup can be achieved by using SSE instructions to accelerate the alpha blending image processing algorithm in a Flash application. The SSE instructions are accessed by providing a standardized limited native interface to the Flash application.**

Keywords-SIMD; image processing; native code interface; image processing acceleration; virtual machine; Flash

I. INTRODUCTION

Since the advent of Web 2.0, Adobe Flash has played an important role in making websites interactive and fun to use. An example of this is the well-known YouTube service, which allows users across the world to share and stream videos. ActionScript, currently in version 3, is the programming language used to create Flash programs. Flash's ActionScript, just like Java, is an interpreted language. The ActionScript Virtual Machine (AVM) performs this interpretation. The advantage here is that the AVM allows Flash applications to run in a platform independent manner.

The inclusion of various image, video, and audio type processing makes Flash applications feature rich. The data input to the application must be processed in a timely manner. Otherwise, the application will not be considered to be enjoyable. Flash applications are interpreted and so they are not expected to perform as quickly as applications that are compiled and linked for a particular hardware platform, but are often expected to meet a user's perceived real-time perspective.

The Single Instruction Multiple Data (SIMD) instructions allow programs that feature vector-based processing such as image, video, and audio to be accelerated. The Intel x86 and x64 platforms support SIMD instructions inside the Central Processing Unit (CPU), and these instructions are known as Streaming SIMD Extensions (SSE). SSE instructions have been added to the instruction sets of modern CPUs to offer fast vector processing possibilities.

Currently, image and other types of vector processing can be accelerated in Flash programs using hardware acceleration from the Graphics Processing Unit (GPU). This acceleration is only available when used with certain graphics cards from Nvidia and ATI [1]. In addition to the GPU, SIMD instructions available on the CPU can also be used to accelerate vector processing for Flash applications. Unlike the expensive GPU, which is a computer system add-on, CPUs with SSE are more widely available and have no extra price tag.

In this paper, we show that SSE instructions available on an Intel x86 CPU can be used to accelerate the processing of images in a Flash application. This work follows closely in the footsteps of [2], which used SSE to accelerate Java applications using available SIMD instructions. The acceleration for Flash applications is performed by providing a limited native interface to a standalone Flash application so that it can access SSE instructions directly. This study focuses on accelerating the alpha blending image processing algorithm for color images in the RGB color space using SSE instructions.

Figure 1 shows the high-level view of the components involved in accelerating image processing in Flash applications using SSE. The native interface opened by the Flash application allows it to invoke functions in a Dynamic Linked Library (DLL), which in turn uses SSE instructions to accelerate the processing of images. We found no explicit support in the AVM to make use of SSE instructions from within Flash applications and it is not discussed in the available AVM literature.

The remaining sections of the paper are organized as follows: Section 2 provides background information on Flash applications, SSE, and alpha blending, Section 3 discusses related work, Section 4 discusses the implementation work of this study, Section 5 discusses the experimentation that was performed to evaluate the effectiveness of the proposed method and results, and we conclude by summarizing our results in Section 6.

 ISBN: 978-1-61208-129-8

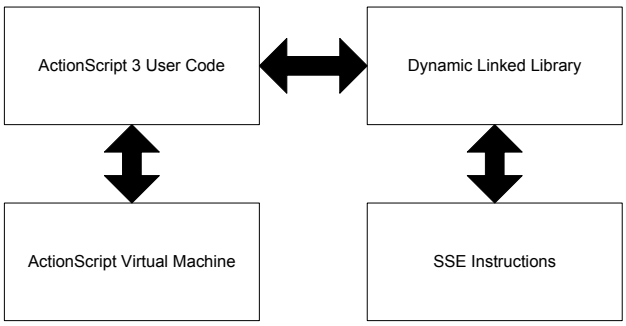

Figure 1. High-level organization of the method to accelerate image processing in Flash applications.

II. BACKGROUND

This section provides background information on the technologies used in the study. Section 2.1 briefly outlines Flash, Section 2.2 discusses Intel's SIMD version: SSE, and finally Section 2.3 illustrates the alpha blending image processing algorithm.

2.1 Flash

An Adobe Flash program compiled into a .SWF file (pronounced as "swiff") runs on any web browser that has a Flash Player plug-in installed. Flash media can also run as a standalone application outside a web browser when packaged into an executable file on various operating systems (OS). These are known as Flash projectors. Having Flash Player allows Flash programs to run on any processor architecture and OS, which is similar to how Java programs run on top of a Java Virtual Machine.

The low level processing in a Flash program can be implemented using ActionScript 3. The ActionScript 3 code is compiled into ActionScript Byte Code (ABC) and packaged into a .SWF file, which is interpreted by the AVM when the Flash application is executed [3].

The Flash platform does not provide a native interface for Flash applications to invoke native code. The lack of native access can be attributed to the perceived security risks that it imposes, and the cross-compatibility that may result from calls to native code. For example, a rogue Flash application running in a web browser could hack the target machine, and the Flash VM of a cell phone is likely not equipped to execute native code for a desktop PC. We leave the problem of security and cross-compatibility for future work. The Zinc 3.0 Flash builder tool by MDM provides the ability to create a standalone Flash projector from a .SWF file and includes functionality in the projector to load a DLL [4]. This is the basis for our native interface.

2.2 SSE

SSE is Intel's version of SIMD instructions for their current and recent x86-64 CPUs and is the evolution of MMX. AMD processors also feature support for SSE. Support for SIMD initially started with the introduction of MMX and until today SSE has had many revisions and currently it is at revision 4.2 (SSE4.2) [12]. The SSE architecture features eight 128-bit wide vector registers. In 2010, Intel showcased its Advanced Vector Extensions (AVX), which feature 256-bit wide vector registers [6]. The first version of SSE provided more support for floating point operations compared to integer operations. In addition, it does not allow an instruction to pack smaller units of data (8-bit values) to vector registers.

SSE2 enhanced SSE by providing more support for integer arithmetic including operations to handle 8 and 16-bit data in SSE registers. This is ideal for image processing as the smallest size of data (e.g., a color channel of a pixel) in an image is usually an 8-bit value [5]. The net result of using SSE2 for image processing rather than SSE is that more pixels can be loaded into a single vector register and more parallelism can be achieved and exploited. For this study, SSE2 instructions were used to accelerate image processing.

2.3 Alpha-Blending

Simply put, alpha blending is an image processing algorithm that allows two images to be blended together [7]. An example is blue screen matting, where a newscaster can be superimposed in front of a particular background to give the illusion that he/she is actually in front of the background. To perform alpha blending, Eq. (1) can be used. FG and BG correspond to foreground and background images respectively. F is the resulting image from the blending, and α ranges from 0 to 1 inclusively. By changing the value of α, the contributions of the pixels from the foreground and background to the blended image can be controlled. To apply this equation to color images, all three color channels, i.e., red, green, and blue values have to be updated in the same manner.

$$F\ pixel = \alpha \times FG\ pixel + (1.0 - \alpha) \times BG\ pixel \quad (1)$$

An optimized version of the alpha blending equation is shown in Eq. (2). This equation was derived from an equation shown in [10]. Unlike in (1), the values of α in this equation range from 0-255. Therefore, this equation assumes that the α values are read from an image. This image is known as a matte.

$$F\ pixel = \alpha \times (FG\ pixel - BG\ pixel) >> 8 + BG\ pixel \quad (2)$$

Note that the right shift by 8 (divide by 256) introduces a certain amount of error in the blending as the value of α should be divided by 255 to ensure that the values are within the proper range (0 to 1). This error cannot easily be detected by the human eye. An example

 ISBN: 978-1-61208-129-8

of a matte image is shown in Figure 4. Figure 5 shows an example of how alpha blending was applied to blend the background shown in Figure 2 and foreground shown in Figure 3 using the matte available in Figure 4. SIMD can easily be applied to accelerate alpha blending because the same operation is applied to all pixels and each color channel in the pixel.

Figure 2. Example background image.

Figure 3. Example foreground image.

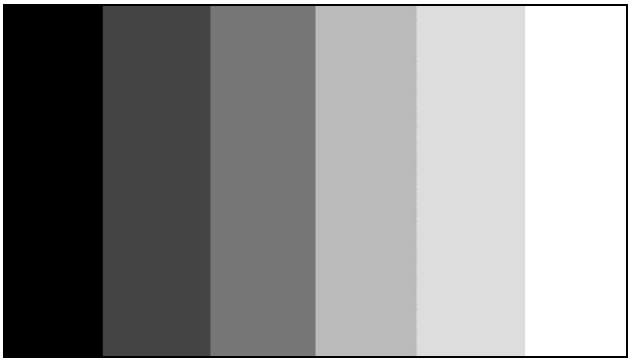

Figure 4. Example custom matte.

Figure 5. The alpha blended image.

III. Related Work

The work done in this paper is in line with the work of [2] to accelerate Java applications that perform vector operations using SSE instructions. Their work used the Java Native Interface (JNI) to access SSE instructions and accelerate vector operations in Java. Their vector operation was completely implemented using SSE instructions and wrapped in a function, which is part of a DLL and presented as a standardized API. Their results showed a 1.0x-4.0x speed up of the Java application with SSE acceleration over an application that implements the operation only in Java.

Usage of SSE instructions to accelerate image processing is not a new concept. This has been demonstrated in the Intel whitepaper [5], which makes use of SSE, SSE2 and AVX instructions to accelerate cross-fade and sepia filters in image processing. Since processors that support AVX instructions are still not mass produced, Intel used a processor simulator to simulate the AVX instructions. In [9], SSE2 instructions were used to speed up a Harris corner detector used in image processing.

Adobe provides the ability to accelerate image and other vector processing algorithms using the GPU in their Pixel Bender Toolkit [1, 11]. Using this toolkit a developer can implement a vector processing algorithm and compile it into byte code. In the Adobe literature, the vector processing algorithm produced by the toolkit is known as a shader [11]. This byte code is imported into the Flash application by the ActionScript code and is used to perform the vector processing. This capability was added starting with Flash Player 10. The usage of the GPU is opaque to the developer and there is no way to say for certain that specific operations are performed in the GPU or not. Another disadvantage of this approach is that the acceleration can only be achieved with certain GPUs: only a subset of those available from Nvidia and ATI. Overcoming this weakness, the work in this paper shows that the vector unit is a viable alternative to the GPU. Hence, a user should not have to invest in an additional

 ISBN: 978-1-61208-129-8

graphics card but instead experience optimal acceleration from the already available vector unit in the processor.

IV. IMPLEMENTATION

To invoke SSE instructions in a Flash program, a limited native interface was added to Flash applications (see Figure 1). This approach is similar to the JNI, which allows Java applications to call native functions in a DLL written in other languages such as C++. The Flash platform does not provide a native interface that Flash programs can use. To overcome this issue the MDM Zinc 3.0 Flash Builder tool was used.

Prior to implementing an image processing algorithm, a generic vector processing algorithm was implemented in a manner similar to [2]. In their work, the data that is processed by the Java application is generated by the Java application and is processed by the native function. Due to severe limitations in the native interface provided via the Zinc 3.0 tool, no speedup of the algorithm was achieved when the Flash application relied on SSE instructions to accelerate the vector operation. This is because the native interface only allows the programmer to pass either primitive types (i.e., different size integers) or a string of characters, therefore, if a vector of integers have to be passed to the DLL to process and another vector has to be returned then each element in the vector has to be converted to a string of characters. Furthermore, this string of characters must be decoded and the original vector has to be reconstructed so that the data can be used. This operation of encoding and decoding of the elements of the vector consumes a large amount of time and this time alone was enough to offset the execution time improvement, such that the implementation of the vector operations in Flash was faster. We look to greatly improve on this in future work.

Instead of operating on vectors using data that are generated in the Flash application, a different approach was applied where the Flash application performs vector processing on file based data such as image files or audio files. The native code in the DLL performs the vector processing as before. The Flash application does not load the image file into its local memory space but instead it delegates that task to the DLL by passing it a path to the location of the file. After the images are loaded, the Flash application invokes a function that performs the image processing, i.e., the alpha blending. The native function saves the resulting blended image to a location specified by the Flash application. Once the control returns back to the Flash application, it loads the blended image to its own local memory space and displays the image to the user.

The native code to perform alpha blending was implemented using SSE2 instructions. The pseudo implementation of the algorithm using SSE2 instructions is shown in Figure 6. The underlying assumption in the implementation is that the total number of the pixels in each image (i.e., the foreground, background, and matte images) should be a multiple of 16. This is a reasonable assumption as almost all of the mainstream image resolutions (e.g., 720p, 1080p) have this property. The matte image is usually gray-scale but it was converted to an RGB image so that the algorithm could be implemented with less complications. Note that the RGB channel values for a gray-scale image are the same.

Initially, 16 bytes worth of pixels of each image is loaded into three SSE registers. The Portable Image Library (PIL) was used as an image handling library [8]. The PIL stores a pixel as a 3-byte value (each color channel occupies one byte), as it loads an RGB image and returns a handle to the location that it is stored in memory [8]. Because 3 bytes per pixel are needed, there can be at most five full pixels and one partial pixel in an SSE register.

All three images have to be interleaved with zeros because the SSE2 multiply operation only multiplies 16-bit data. Note that, because of the interleaving, 8 bytes of pixel data will be in one SSE register and the other 8 bytes will be in another register. After the operations required for the algorithm are performed, the bytes that correspond to the pixel data have to be extracted one by one and written to the buffer that holds the resulting image.

At first glance, this operation may seem very inefficient due to 16 individual memory writes by software. However, the performance of this approach was compared against another implementation that makes use of an SSE3 instruction (`PSHUFB`). This instruction can be used to shuffle the bytes in an SSE register around. With this instruction the two sets of interleaved data can be placed into one single SSE register and the contents of it can be written to memory using a single SSE2 instruction. When the performance was evaluated, the execution time was exactly the same as the original solution, and since SSE3 is supported only on newer Intel CPUs (Core 2 Duo, Core i7), the SSE3 approach was not pursued any further.

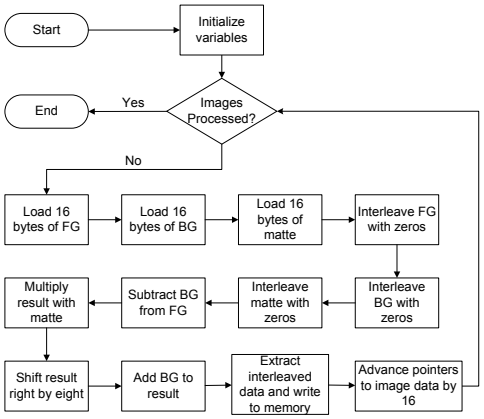

Figure 6. Pseudo SSE2 implementation of the alpha blend algorithm.

V. EXPERIMENTAL EVALUATION AND RESULTS

To evaluate the effectiveness of accelerating alpha blending in a Flash application using SSE instructions, the performance was measured for four different implementation possibilities, and the results compared. The approaches evaluated were: implementation of the algorithm with our accelerated Flash approach, only in Flash, only in C, and in C using SSE instructions. In the latter implementation, the DLL that was compiled for the Flash and SSE implementation was reused. The latter two implementations represent the best possible implementation in terms of performance as the program is compiled to run directly on the hardware without any interpretation. Alpha blending was performed on a pair of eight images with different image resolutions. In addition, eight matte images were also used for the blending. Only mainstream image resolutions were used for the evaluation as the total number of pixels is a multiple of 16.

The evaluation of all four implementations of alpha blending was done on an Intel Centrino Duo machine with 2.5GB of RAM. The OS used on the machine is Windows Vista. The C implementation and the DLL were compiled with optimization for speed enabled. In the Flash-only implementation, the entire set of pixels were extracted and stored in an instance of a vector class in ActionScript 3. This method provides the quickest possible implementation.

Fifty runs were performed on each resolution for each implementation and the execution time for each run was recorded. Note that only the image resolution and not individual pixel values affect the performance of the given operation. The average and the standard deviation were computed at the end of each run. Figure 7 shows pseudo code for the method that was used to measure the execution time of each implementation. The average execution times (in milliseconds) for each resolution used in the performance evaluation are shown in Table 1, while Table 2 shows the standard deviations of the execution times. Figure 8 depicts the average execution times shown in Table 1 in a graphical format.

```
start = get_sys_time_stamp()
alpha_blend()
end = get_sys_time_stamp()
execution_time = end - start
```

Figure 7. Method of measuring execution time.

According to the results shown in Table 1 and Figure 8 it is evident that there is a speedup when a Flash program makes use of SSE instructions to perform vector operations over an implementation that solely relies on Flash to perform the computations. This speedup is summarized in Table 3. In addition, it shows the speedup of the implementation of the algorithm in C that uses SSE over the implementation that uses both Flash and SSE. We found that the Flash with SSE implementation has a speedup of 90-92% over a Flash-only implementation and a speedup of negative 7%-34% over a C and SSE implementation. The results were in line with expectations, as the native code was faster than the code running through the AVM. Since the standard deviations of the execution times were low it is evident that external factors such as garbage collection in the AVM and OS context switching have not influenced the variability of the runtimes in a drastic manner.

TABLE 1. AVERAGE EXECUTION TIMES IN MS FOR DIFFERENT IMAGE RESOLUTIONS.

Image Resolution	T_{Flash} *(ms)*	$T_{Flash+SSE}$ *(ms)*	T_{C} *(ms)*	T_{C+SSE} *(ms)*
640x480	30.16	2.9	3.04	2.04
768x576	43.84	3.9	4.22	3.06
800x600	47.76	4.78	5.02	3.14
1024x600	60.42	5.22	6.14	4.04
1280x720	89.82	7.86	10.02	6.08
1366x768	102.56	8.58	11.02	7.18
1680x1050	173.06	13.82	19.1	12.13
1920x1080	201.66	16.2	23.44	15.06

TABLE 2. STANDARD DEVIATIONS OF THE EXECUTION TIMES FOR DIFFERENT IMAGE RESOLUTIONS.

Image Resolution	σ_{Flash}	$\sigma_{Flash+SSE}$	σ_{C}	σ_{C+SSE}
640x480	0.65	0.416	0.282	0.282
768x576	1.076	0.364	0.418	0.24
800x600	1.733	0.932	0.141	0.405
1024x600	1.071	0.545	0.405	0.34
1280x720	1.119	0.808	0.141	0.274
1366x768	1.981	1.247	0.141	0.482
1680x1050	1.931	0.748	0.364	0.598
1920x1080	1.303	1.125	2.011	0.314

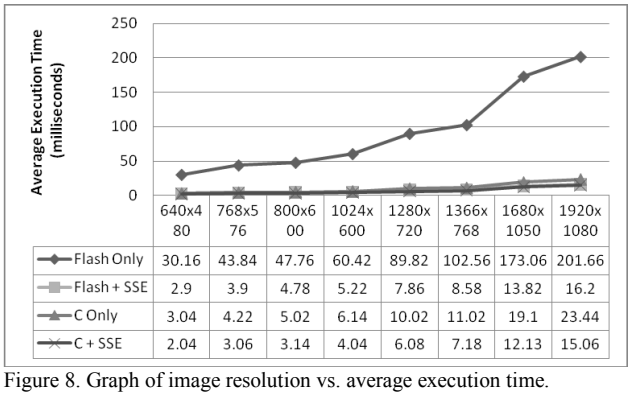

Figure 8. Graph of image resolution vs. average execution time.

TABLE 3. SPEED OF FLASH VS. FLASH+SSE AND FLASH+SSE VS. C+SSE.

Image Resolution	*Speedup Flash+SSE vs. Flash (%)*	*Speedup Flash+SSE vs. C+SSE (%)*
640x480	90.38	29.66
768x576	91.1	21.54
800x600	90	34.3
1024x600	91.36	22.61
1280x720	91.25	22.65
1366x768	91.63	16.32
1680x1050	92.01	12.22
1920x1080	91.97	7.03

VI. CONCLUSION

The purpose of this work was to prove that a Flash application that performs vector processing can be easily accelerated using SIMD instructions such as the SSE instruction sets on common x86-64 CPUs. Adobe already has a solution to accelerate vector-based processing using GPUs, however, this solution is only available to a limited number of GPUs, and moreover only to consumers who pay extra to have a GPU in their system. If the acceleration is provided via the vector unit in the CPU then it is available without having to invest money in a special GPU. The minimum system requirements to run the alpha blending algorithm outlined in the paper is a Intel Pentium 4 machine since it requires support for SSE2 and 128MB of RAM to run Flash Player 10.

This acceleration was proven only in a limited case where the data to be processed can be loaded from a file and the resulting data can be saved back into a file. In addition, the Flash application can only be executed as a standalone desktop application (Flash projector) and not in a web browser in its current state. The reason for this limitation is due to the fact that the current Flash platform does not have a native interface to call native functions in a DLL unlike Java, which has a fully featured JNI, and the fact that the Zinc 3.0 Flash builder only provides support to create standalone Flash desktop applications that feature a very limited native interface. The native interface is provided via loading a DLL and being able to invoke functions in it.

For this work, we implemented the alpha blending image processing algorithm using SSE2 instructions, compiled it into a DLL, and the Flash application invoked the algorithm using the native interface provided by the Zinc 3.0 tool. The experimental results show that there is a speedup of 90-92% when alpha blending is performed by a Flash and SSE implementation over a Flash-only implementation. The size of the image was orthogonal to the speedup. Even though only one image processing algorithm was considered in this study, due to the nature of most image processing and vector processing algorithms it can be concluded that by using Flash applications that have vector operations can be accelerated using SSE instructions. Such inherent parallel support should be included in the ActionScript specification and made available in the AVM.

The solution outlined in this paper neglects that a Flash based web application calling native code via a native interface can introduce a security risk and compatibility issues. To alleviate these security concerns the available options are to:

- Make Pixel Bender generate SSE instructions when it cannot detect the proper GPU on the computer system.
- Incorporate SSE directly into the AVM and augment ActionScript specification for better vector operation support.

In our upcoming work, we will address the security and compatibility concerns posed by exposing the vector instructions directly through generic calls to the VM. Mapped and complete SSE integration into the ActionScript specification and AVM will move ahead with an SSE bypass in the open source ActionScript virtual machine, Tamarin [13], produced by Mozilla and Adobe.

REFERENCES

[1] Adobe Systems Inc., "Pixel Bender release notes." [Online]. Available: http://www.adobe.com/devnet/pixelbender/articles/releasenotes.html. [Accessed: Feb 1, 2011].

[2] J. Parri, J.M. Desmarais, D. Shapiro, M. Bolic, and V. Groza, "Design of a Custom Vector Operation API Exploiting SIMD Intrinsics within Java," 23rd Canadian Conference on Electrical and Computer Engineering, pp. 1-4, May 2010.

[3] *ActionScript Virtual Machine 2 (AVM2) Overview,* Adobe Systems Inc., San Jose, CA, USA, 2007.

[4] MDM, "Zinc™ 3.0 - Rapid Application Development for Adobe® Flash." [Online]. Available: http://www.multidmedia.com/software/zinc/. [Accessed: Feb. 1, 2011].

[5] *Image Processing Acceleration Techniques using Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions,* Intel, Santa Clara, CA, USA, 2009.

[6] Intel, "Picture the future now Intel® AVX." [Online]. Available: http://software.intel.com/en-us/avx/. [Accessed: Feb. 1, 2011].

[7] J.F. Blinn, "Compositing. 1. Theory," *Computer Graphics and Applications, IEEE*, vol. 14, no. 5, pp. 83-87, September 1994.

[8] A. Whitehead, "Portable Image Library (PIL)." [Online]. Available: http://iv.csit.carleton.ca/~awhitehe/PIL/. [Accessed: Feb 1, 2011].

[9] J. Skoglund and M. Felsberg, "Fast image processing using SSE2," in *Proc. SSBA Symposium on Image Analysis,* Malmö, Sweden, 2005.

[10] W. Shao, "Tip: An Optimized Formula for Alpha Blending Pixels." [Online]. Available: http://www.codeguru.com/cpp/cpp/algorithms/general/article.php/c15989/Tip-An-Optimized-Formula-for-Alpha-Blending-Pixels.htm. [Accessed: April Feb. 1, 2011].

[11] *Programming ActionScript 3.0 for Flash,* Adobe Systems Inc., San Jose, CA, USA, 2009.

[12] S. Siewert, "Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms." [Online]. Available: http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms/. [Accessed: Feb. 1, 2011].

[13] Mozilla, "Tamarin Project." [Online]. Available: http://www.mozilla.org/projects/tamarin/. [Accessed: Feb. 1, 2011].

 ISBN: 978-1-61208-129-8