Jitblt: Efficient Run-time Code Generation for Digital Compositing

Daniel Amelang

VPRI Technical Report TR-2008-002

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Jitblt: Efficient Run-time Code Generation for Digital Compositing

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Daniel James Amelang

Committee in charge:

Professor James Hollan, Chair
Professor William Griswold
Professor Alan Kay

2008

VPRI Technical Report TR-2008-002

The thesis of Daniel James Amelang is approved and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____
Chair

University of California, San Diego

2008

EPIGRAPH

I'd rather write programs to write programs than write programs. —Richard Sites

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I wish to acknowledge the many sacrifices that my wife, Layna, made to make this thesis possible. Her constant support stands in stark contrast to my infrequent expressions of appreciation. My two children have been invaluable, contagious examples of carefree living. They were both a much-needed source of stress therapy and inspiration during the writing of this thesis.

I acknowledge that I will never know, let alone appreciate, everything that my parents have done for me. The enormous debt I owe them continues to grow through the years as they continue to help my family and me in many ways.

Viewpoints Research Institute graciously provided my financial support for the past year. Everyone at Viewpoints has been very encouraging and helpful, especially Ian Piumarta and Kim Rose. Ian provided valuable feedback on several early drafts.

I am grateful to the members of my thesis committee for their time and patience. In particular, I am grateful to Jim Hollan for being my advisor and placing his confidence in me. Bill Griswold frequently shared wise counsel with me. I owe a lot to Alan Kay for not only being a helpful member of my committee, but for establishing and perpetuating the intellectual environment that is Viewpoints Research Institute.

The idea behind this thesis grew out of conversations between Carl Worth, Keith Packard, and Viewpoints Research Institute. This thesis would not have happened had Carl not recommended me to Viewpoints as the one to implement the idea.

I am grateful to those behind the software that I used to write this thesis, particularly LaTeX, Inkscape, Vim, Ubuntu Linux, OpenOffice, and the GIMP.

I am indebted to Navdeep Raj for his freely-available icon library which, because he distributed it in source form, allowed me to create Figure 1.1. Figure 2.1 and Figure 3.2 contain a modified version of “Arena y Viento” by Pablo Arroyo. Figure 2.1 also features “Catch the Sun” by Evan Leeson. Both of these images are released under the Creative Commons Attribution-Noncommercial-Share Alike 2.0 license. As required by this license, my figures are also available under this license.

This material is based upon work supported by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

ABSTRACT OF THE THESIS

Jitblt: Efficient Run-time Code Generation for Digital Compositing

by

Daniel James Amelang

Master of Science in Computer Science

University of California, San Diego, 2008

Professor James Hollan, Chair

Digital image compositing is a fundamental operation of computer graphics. For the sake of performance, current implementations shun many important software qualities such as brevity, readability, and extensibility. This thesis presents a novel implementation that uses a powerful programming environment to maintain these software qualities while still allowing for high performance. The approach is based on run-time code generation, using a simple object pipeline architecture to manage software complexity. The resulting implementation, named "Jitblt," is compared to an existing, popular compositing library. The results show that large gains in software quality can be had, but further work is necessary to make this new approach practical.

Chapter 1

Introduction

1.1 Concepts

Digital compositing is the combination of two or more digital images into one [7]. Compositing is a fundamental operation of computer graphics that is used frequently during real-time and offline rendering. The display of modern graphical user interfaces involves the successive composition of various windows and widgets. Text is typically displayed by compositing glyphs onto a background. In state of the art video production, multiple visual sources (e.g., live video streams, computer generated imagery) are composited together to create the final scene [31].

Essential concepts of real-time digital compositing include pixmaps, pixel formats, and compositing operators. Pixmaps (or "color bitmaps") are rectangular 2D images represented in digital form as an array of point samples. Each sample is called a pixel ("picture element") and can be represented in computer memory in many different formats. For example, ARGB formats are commonly used today, storing each pixel as four values representing the intensity of the channels alpha, red, green and blue. A8R8G8B8 is an example ARGB format that stores each channel value as an unsigned 8-bit integer packed together as 32 bits. Pixel formats vary in color format (e.g., Red-Green-Blue vs. Luma-Chrominance), numerical representation (e.g., integer

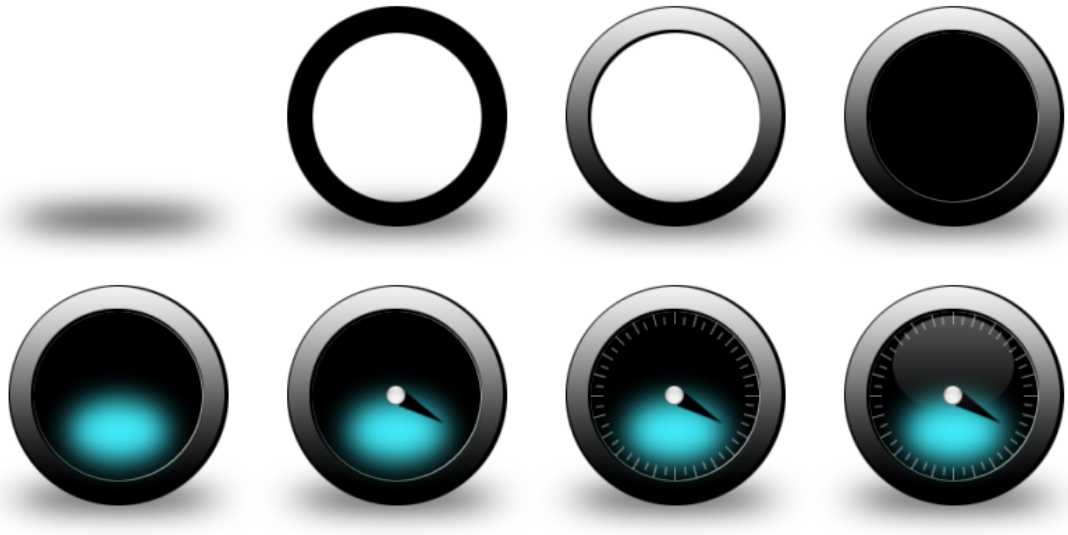

Figure 1.1: User Interface Icon Created Through Successive Compositing

vs. float-point), channel order (e.g., RGB vs. BGR) and precision (e.g., R5G6B5 vs. R11G11B10).

Compositing operators (or "blend modes") determine how two images are combined into one. These operators are often described numerically through simple arithmetic on the values of the input pixel channels. For example, the common compositing operator `OVER` could be expressed simply as `m + n * (1.0 - m.a)`,[1] where `m` is a pixmap being combined "over" the pixmap `n`.

Real-time digital compositing demands high performance. The throughput of an animation, or the latency of a user interface, is often bounded by the performance of the compositing step. Many computer systems, both now and in the future, will not have graphics hardware capable of accelerating image compositing. This need for performance drives CPU-based real-time digital compositing systems to abandon many important qualities of software implementation. To implement high-performance compositing in a high-level, succinct and extensible way is the software engineering challenge that is the topic of this thesis.

[1] The syntax `m.a` is used to reference the alpha channel of the image `m`.

1.2 Related Work

The compositing process began as an analog technology used in the film industry for combining two or more frames into one [2]. Later, the XEROX Alto was introduced with a very basic form of real-time digital compositing implemented via a special routine called "Bit BLT" [12, 13]. The digital images on the Alto were very primitive 1-bit "bitmaps" without color or concept of partial transparency. Bit BLT was capable of four combination "functions" named Replace, Paint, Invert and Erase.

Catmull and Smith developed the concept of the alpha channel for representing partial transparency in digital images [29]. This concept was further explored and formalized by Porter and Duff in the seminal paper on the algebra of digital image composition [27]. A comprehensive treatment of the history, art and theory of digital image compositing can be found in Brinkmann [7].

Later, new compositing parameters such as the alpha channel, image transformations, and the Porter-Duff compositing operators were incorporated into the compositing functions used for the display of user interfaces [3, 14]. Around the same time, several papers were devoted to the many tricks and traps involved in achieving adequate compositing performance on commodity hardware [4, 5].

Reiser was the first to use dynamic code generation to elegantly implement high-performance compositing [22]. Previous approaches required generating specialized code paths *a priori*, often by hand. Dynamic code generation (also called run-time code generation) had been used much earlier by Thompson to accelerate regular expression matching [30]. These approaches to optimization are special cases of the general concept of partial evaluation (also called program specialization) [15].

Run-time code generation for partial evaluation was suggested and implemented successfully for several uses [17, 26]. An attempt was made to perform specialization automatically at run-time with the specific application of pixel manipulation [8]. Unfortunately, the approach proved impractical for many reasons [9] that are discussed in Section 2.1.

Currently, high-performance compositing (when performed on the CPU) is still a mixture of hand-written special cases and explicit run-time code generation. On the most recent version of Apple's OS X, LLVM intermediate code is generated and compiled at run-time to perform fast compositing [18]. Elliott et al. [10] describes a system built on a dialect of Haskell that compiles compositing expressions at run-time. In a recently released book, Petzold describes the use of the run-time code generation capabilities of the .NET framework to perform fast image compositing [21].

Unfortunately, none of the above approaches achieve both high performance and elegance. The ideal solution would not only be fast, but also expressed in a way that makes the source code highly understandable and extensible. Severely optimized code is often thrown out and rewritten because only the original author, likely no longer involved in the project, can comprehend the magic involved. Furthermore, the nature of computer hardware changes rapidly, making the optimizations of yesterday less effective today.

The contribution of this thesis is the presentation of a novel approach for implementing fast digital compositing gracefully. The key to the approach is the use of a new programming environment with powerful metaprogramming capabilities. Chapter 2 describes a compositing library implemented using this system. An analysis is made of this library in Chapter 3, followed by conclusions about the result in Chapter 4. Technical details of the new programming environment are provided in Appendix A.

Chapter 2

Implementation

"Jitblt"[1] is the name of the digital image compositing library that was implemented for the purpose of this thesis. The unusual name is a portmanteau of "JIT" (from just-in-time compilation) and "Bit BLT". It utilizes explicit run-time code generation to perform compositing, and can do so for 20 pixel formats and 15 compositing operators. The main objective of Jitblt is to provide fast compositing while maintaining a small, high-level implementation. Jitblt was implemented using a new programming system named COLA. Details of this system can be found in Appendix A. COLA was ideal because it provides powerful metaprogramming facilities, making code generation simple and elegant. Jitblt, like COLA, is open-source software.[2][3]

2.1 Approach

Compositing systems typically provide an interface centered around a single function, method or object that is highly parameterized. These parameters include references to pixmaps, compositing operators, pixmap offsets, and the dimensions of the region to be composited. It is easy to process the numeric parameters like offsets and

[1] Apologies to Andreas Raab who did some earlier, related work under the same name.
[2] git://people.freedesktop.org/~dan/pixman
[3] http://vpri.org/pipermail/jitblit

dimensions, but it is more difficult to handle the many possible permutations of pixel formats and compositing operators.

A naive attempt to support these parameters would be to create a pixel unpack function for each pixel format, in addition to a pixel pack function for each format, and also a pixel combination function for each compositing operator. To perform compositing, the system would determine the appropriate functions for the given input parameters, and iterate through each pixel of each pixmap, calling the unpack, combine and pack functions.

Unfortunately, this approach is not feasible for real-time compositing systems because the overhead of the function calls dominates the performance profile. This is due to the fact that these functions perform very little computation, yet are called once for every pixel. It is not uncommon for a compositing system to be expected to process millions of pixels per second.

The function call overhead can be eliminated using inline expansion. For each permutation, a single function can be created that contains the pixel iteration loop with all the pack, unpack and combination code together. These specialized functions can be written manually or generated programmatically.

If the number of possible permutations is large, statically generating all corresponding functions *a priori* is not feasible. Jitblt supports 20 formats and 15 operators. Pixman-Jitblt (a Jitblt-based system presented in Section 3.1) can apply 2 operators to 3 formats in a single compositing request. This means that 1,800,000 functions would need to be generated. The sheer size of the generated code makes this approach impractical. A hybrid approach is possible, where functions for only the most frequently requested permutations are generated, while the rest are serviced by a slow, general code path. The trouble here is that 1) the most common permutations might not be known and 2) the number of common permutations might still be too large.

These problems can be solved by dynamically generating the specialized functions as needed. Typically, only a small number of permutations are used for a given software and hardware combination. Thus, dynamic generation allows for minimal gen-

erated code while maintaining high performance. This introduces another problem: code complexity. Programs that employ run-time code generation are often awkward to implement and difficult to understand.

One solution to this problem is to create a compiler that performs automatic program specialization [8]. This compiler would take as input the general, slow solution first proposed, plus the parameters on which specialization is to be performed. Once given concrete values for the parameters, the compiler would be able to automatically produce the specialized code paths as needed. This approach is extremely elegant in theory, because the source code of the compositing system can be kept simple and general, and the overhead of function calls and branches can be ignored. Yet, all the performance benefits of specialization are still had, thanks to the compiler.

No compiler of this sort is known to exist, at least not in any practical form. Draves [9] documents the most successful effort. Draves' system, which happened to also focus on pixel manipulation, was fraught with problems. It appears that shifting the responsibility of specialization to a general tool makes the task intractable. For this reason, Jitblt does not try to perform specialization automatically. Code generation for the specialized functions is done explicitly. The difficulty of the task is managed instead by using a programming environment that provides simple but powerful run-time code generation facilities.

2.2 Pipeline Architecture

The code generation functionality of Jitblt is distributed among several small software objects. Each pixel format and compositing operator is represented by a single object whose primary purpose is to generate code fragments. These objects are created within the prototype-based object system of COLA. Organizing code generation behavior into prototype hierarchies has proven effective in reducing repetition and increasing clarity.

The code fragments generated by the objects must be assembled together to cre-

ate the specialized function. This is done by connecting the code generation objects together to form a pipeline. This configuration mirrors the data flow of the compositing process. Pixel unpacking occurs at the head(s) of the pipeline, compositing operators are applied in the middle, and pixel packing is done at the bottom.

Figure 2.1 illustrates the structure of a typical pipeline. In this example, an image with pixel format R5G6B5 is combined with an A8 image using the IN compositing operator. The result is then combined with another R5G6B5 image using the OVER operator. The final result is stored into an R5G6B5 image.

This pipeline of objects generates the actual compositing code when the `store` method is invoked on the last object in the pipeline. This method doesn't pack and store any pixels. Instead, it returns a code fragment that, when compiled and executed, performs the pack and store. In addition, this code fragment will contain all the code fragments from the objects above it in the pipeline. Thus, code generation follows a "pull" model where objects below request code from those above and incorporate the result in their own code. The request is made by calling the methods `a`, `r`, `g` and `b` on the parent object(s). These methods correspond with the channels alpha, red, green and blue, and the return values are the code generated by the parent for that particular channel.

Figure 2.2 exhibits the COLA code generated by the pipeline in Figure 2.1. Pixman-Jitblt (introduced later in Section 3.1) generates this code when invoked through the function call: `(jitblt-compile R5G6B5 IN A8 OVER R5G6B5)`.

In addition to pixmaps, Jitblt also supports solid (i.e., uniform) colors as pixel sources. A solid color can be understood as a special case pixmap that produces the same pixel value across iterations. The pixel source concept could be generalized further to include color gradients or even programmatic pixel sources, although Jitblt currently does not provide anything beyond solid colors and pixmaps.

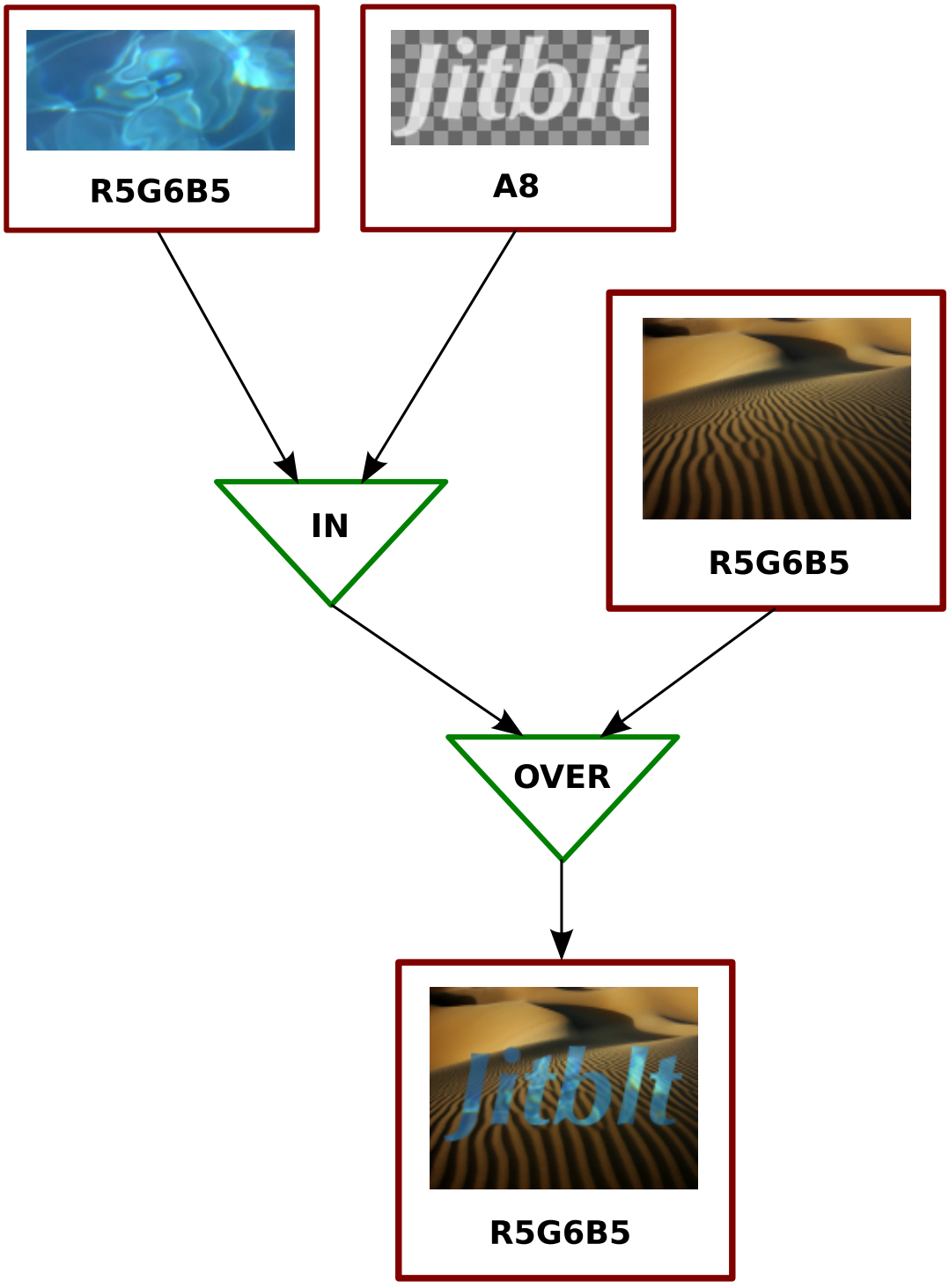

Figure 2.1: Sample pipeline 1. Red boxes represent pixel formats. Green triangles represent compositing operators. Arrows specify direction of data flow. Example input and output images are included to illustrate the effect of the pipeline.

```
(set (short@ d) (|
  (<< (& (+ (let ((t (+ (* (let ((__r (& (>> (short@ s) 8) 248)))
                             (| __r (>> __r 5)))
                           (uchar@ m)) 128)))
                (& (>> (+ (>> t 8) t) 8) 255))
              (let ((t (+ (* (let ((__r (& (>> (short@ d) 8) 248)))
                               (| __r (>> __r 5)))
                             (- 255 (uchar@ m))) 128)))
                (& (>> (+ (>> t 8) t) 8) 255))) 248) 8)
  (<< (& (+ (let ((t (+ (* (let ((__g (& (>> (short@ s) 3) 252)))
                             (| __g (>> __g 6)))
                           (uchar@ m)) 128)))
                (& (>> (+ (>> t 8) t) 8) 255))
              (let ((t (+ (* (let ((__g (& (>> (short@ d) 3) 252)))
                               (| __g (>> __g 6)))
                             (- 255 (uchar@ m))) 128)))
                (& (>> (+ (>> t 8) t) 8) 255))) 252) 3)
  (>> (+ (let ((t (+ (* (let ((__b (& (<< (short@ s) 3) 248)))
                             (| __b (>> __b 5)))
                           (uchar@ m)) 128)))
                (& (>> (+ (>> t 8) t) 8) 255))
              (let ((t (+ (* (let ((__b (& (<< (short@ d) 3) 248)))
                               (| __b (>> __b 5)))
                             (- 255 (uchar@ m))) 128)))
                (& (>> (+ (>> t 8) t) 8) 255))) 3)))
```

Figure 2.2: Generated COLA code for the parameters in Figure 2.1. The variables `s`, `m` and `d` are part of the boilerplate code (see Section 2.3.3) and are supplied to the pipeline objects at instantiation.

2.3 Compilation

2.3.1 Fixed-point Arithmetic Transformations

Internally, Jitblt uses 8-bit unsigned integers for all compositing arithmetic, with the range 0 - 255 representing 0 - 100% channel intensity. But, to make the code more readable and general, the compositing operators are expressed in the source code using floating-point numbers in the range 0.0 - 1.0. This way, numeric constants aren't tied to their integer approximations. Furthermore, multiplication and division of channel values do not require manual adjustment as they do in the integer, fixed-point form. For this purpose, a small function was written for transforming COLA S-expressions from floating-point to integer fixed-point form. This function properly scales all constants and inserts the necessary arithmetic adjustments. All pipeline-generated code is processed by this function before being converted to machine code. This function was very little extra work (only 12 lines of code), yet it significantly increased the readability of the compositing operator code.

2.3.2 Optimizations

The COLA compiler used by Jitblt does not perform any compiler optimizations. Because compositing entails repeating the same small number of calculations thousands, sometimes millions of times per second, performance of the compositing code is a critical issue. Soon after Jitblt was functional, it became apparent that compositing performance was poor, and I began to examine the generated code to identify optimization opportunities. I found that the compositing code performed numerous redundant calculations that would easily be eliminated through applying simple, well-known compiler optimizations [1]. Due to the high run-time cost of performing these optimizations, they are only applied to the pipeline-generated code.

The first optimization was algebraic identity removal. Applying this optimization was very straightforward. It took 21 lines of COLA code to implement the identi-

fication and removal of 14 identities. No modifications of the core COLA system were needed. I merely defined a function that performed a source-to-source transformation of COLA S-expressions, applying itself recursively. The second compiler optimization was constant folding. In 12 lines of code, this optimization was performed similarly to the algebraic identity removal. The third was common subexpression elimination (CSE). The implementation of this optimization was never finished due to time constraints.

2.3.3 Machine Code Generation

The code produced by the pipeline of objects is the heart of the specialized function, but it is not complete. The code must be inserted into boilerplate code that defines the function itself, handles pixmap pointer initialization, calculates pixmap strides, and defines nested loops for iterating through the pixels. Once this is all together, the COLA compiler is invoked on the generated code. The compiler produces machine code and returns a C-compatible function pointer to this code. This generated function is a compositing function that is specialized on the given pixel formats and compositing operators. When invoked, one must only provide arguments for the pixmap data, pixmap offsets, and compositing region dimensions.

Chapter 3

Analysis

3.1 Pixman-Jitblt

To establish that Jitblt has certain advantages over the status quo of real-time digital image compositing libraries, I chose to compare Jitblt to an existing, widely-used compositing library. Pixman[1] is the name of the software compositing library used in several software systems today, including the Firefox web browser, and the X.Org implementation of the X Window System. Pixman is an ideal candidate because it is available under a non-restrictive open source license, making it freely available, inspectable and modifiable. Pixman is written in the C programming language and is the product of the work of dozens of developers over approximately seven years of ongoing development.

Once Jitblt was complete, I modified the Pixman version 0.9.3 source code to create a version of Pixman that used Jitblt as a "compositing backend." That is, I wrote a layer of code inside Pixman that delegated the actual compositing functionality to Jitblt, bypassing the usual Pixman compositing code. Thus, I was able to create a drop-in replacement for the Pixman library that was actually a front-end for Jitblt. This proved to be very advantageous because I could reuse the Pixman API to access the compositing

[1] PIXel MANipulation library, part of the Cairo graphics project.

functions of Jitblt. Any performance tests written for the Pixman library could be reused for Jitblt testing, and all applications that use Pixman through a shared library could utilize Jitblt without recompilation. "Pixman-Jitblt" is the name I use to refer to this hybrid system.

3.2 Visual Output

Ideally, Jitblt should produce visual output that is close, if not identical, to Pixman. Performing an exhaustive evaluation, though, was not feasible. Fortunately, the Cairo graphics project[2] has a suite of over a hundred unit tests that exercise the basic capabilities of Pixman. I used these tests to compare the visual output of an unmodified Pixman with my modified Pixman-Jitblt. A few discrepancies surfaced which indicated some numerical errors in Jitblt. Once these errors were corrected, the output of the two versions was identical.

3.3 Functionality

Pixman provides a large range of functionality. Jitblt does not attempt to be a full replacement for Pixman and therefore provides only a subset of Pixman's functionality. To make Pixman-Jitblt a drop-in replacement for Pixman, it was necessary to have Pixman-Jitblt delegate to the original Pixman compositing code when unsupported functionality was requested.

The subset of Pixman that Jitblt does provide includes all 14 of the compositing operators. In addition, Jitblt can perform what Pixman calls "component alpha," which is similar to the `IN` compositing operator, except that it multiplies by a different value for each channel. Jitblt provides component alpha by defining an additional compositing operator called `COMPONENT_IN`. The extensible pipeline architecture of Jitblt made the introduction of component alpha much more graceful than that in Pixman.

[2] http://cairographics.org

Jitblt supports 20 of the 35 pixel formats that Pixman supports. Most of the unsupported formats (e.g., indexed color, grayscale) are not in wide use today. To replace Pixman entirely would require implementing the additional features of color gradients, transformations (e.g., image rotations), convolution filters, alpha maps, read/write callbacks and several repeat modes (e.g., image tiling).

3.4 Lines of Code

The biggest difference between the implementations of Pixman and Jitblt is that Pixman does not use dynamic code generation. Pixman contains a general code path for compositing. This path first unpacks all the pixels for a given scanline into a normalized pixel format (A8R8G8B). Then it calls one or more functions to perform the compositing arithmetic on the whole scanline. Finally, it packs the scanline into the pixel format of the destination pixmap. Thus, Pixman tries to reduce function call overhead by "batching" the pixel processing one scanline at a time.

This general code path proved too slow for many real-world uses. The Pixman developers began to introduce manually-generated specialized code paths for the most common combinations of compositing parameters. These special cases are detected through a long series of nested switch/case statements that default to the general code path when no specialized code path is found. Over time, the number of special cases introduced into Pixman has grown to the point that the code base is awkward to maintain. In Pixman version 0.9.3, the switch statement is 568 lines long and the special cases total approximately 3880 lines of code. In total, Pixman dedicates approximately 8940 lines of code to digital image compositing.

In contrast, Jitblt is only 473 lines of code. Even with the additional glue code necessary to embed Jitblt in Pixman, Pixman-Jitblt is only 765 lines of code. Part of the discrepancy is due to the limited functionality that Jitblt offers. Part is due to the lack of carefully optimized code paths in Jitblt. It is hoped, though, that this additional

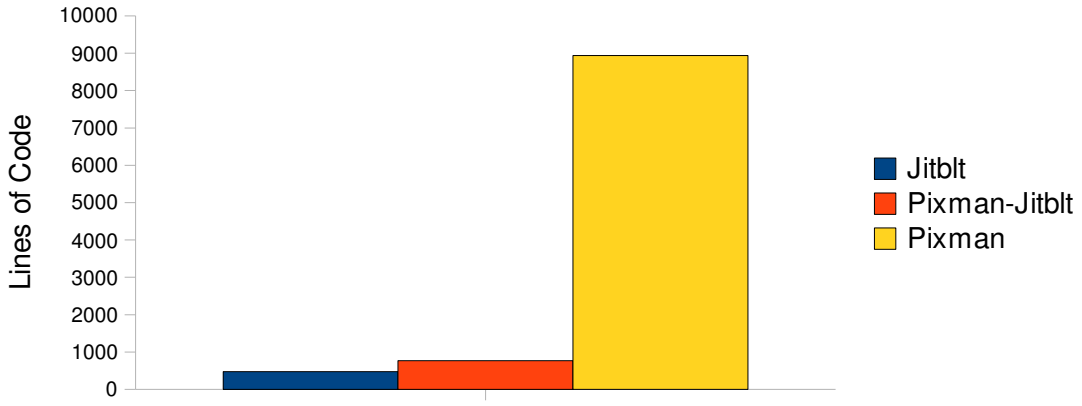

Figure 3.1: Lines of Code Comparison

functionality and performance can be achieved within Jitblt without losing clarity and concision.

3.5 Memory Footprint

Pixman-Jitblt requires significantly more memory than unmodified Pixman. For example, under the setup used for the performance tests, the original `libpixman.so` shared library file was 0.8 MB, while the Jitblt version was 3.3 MB. In addition, the Jitblt version uses run-time memory allocation extensively. The performance test for unmodified Pixman had a stable resident set size of 0.5 MB during execution. The same test using Pixman-Jitblt had a stable resident set size of 6.2 MB. Both memory issues are related to the use of the COLA environment and are discussed further in Appendix A.

3.6 Performance

All performance tests were executed on a 1.8 GHz Pentium M processor with 768 MB RAM running Linux 2.6.22. The processor frequency was fixed at 1.8 GHz to avoid any possible interference by automatic power management. To determine duration of execution, the processor time stamp counter was sampled using the RDTCS instruc-

tion. Each test was executed multiple times and the shortest duration was selected to represent the outcome.

3.6.1 Startup

Without Jitblt, Pixman incurs virtually no startup penalty. That is, applications using Pixman will not notice any significant delay during startup that is a result of Pixman. Pixman-Jitblt, on the other hand, incurs a delay of 3.2 seconds at startup. This delay can be broken down into two parts: 0.9 seconds for COLA bootstrap + 2.3 seconds for Jitblt source code compilation. These issues are further explored in Appendix A.

3.6.2 Function Compilation

The timing of function compilation varies widely depending on the input parameters. For example, the pipeline featured in Figure 2.1 takes 15 milliseconds to compile, while the pipeline in Figure 3.2 takes 256 milliseconds. The reason for the variation is related to the Jitblt-specific code optimizations described in Section 2.3.2. Applying these optimizations can be costly. Thus, pipelines with no applicable optimizations (e.g., no algebraic identity removal possible) are compiled quickly, while those with many applicable optimizations take longer.

Because function compilation is too costly to be performed for each compositing request, Pixman-Jitblt keeps a small cache of recently-compiled functions. The use of this cache effectively amortizes long compilation times.

3.6.3 Function Execution

The most important performance area is function execution. The actual digital image compositing is performed while the function is executed. In a typical application, a function may be expected to process hundreds of thousands of pixels per second. This compositing step is the performance bottleneck in many real-time 2D graphics

applications. Therefore, for any real-time compositing library to be of practical use, it must provide adequate performance.

To gauge performance, I timed the execution of two different specialized functions. The first function was already illustrated and explained in Section 2.2 and featured in Figure 2.1. This permutations of parameters is interesting because the `R5G6B5` format is often the framebuffer format on embedded devices. Thus, this pipeline represents the realistic scenario of an image composited through a mask onto a framebuffer. This permutation is also interesting because Pixman does not have a manually-optimized code path for this combination. Therefore, this case allows for the comparison of the general compositing code path of Pixman with Jitblt. Because the general path is the slowest path in Pixman, this test represents the best advantage scenario for Jitblt.

The second function was created with the parameters illustrated in Figure 3.2. This pipeline is similar to the first except that instead of an image, a uniform color is applied through the mask. This combination is also very common and, unlike the first function, has a manually-optimized code path in Pixman. This optimized path is quite fast in part due to the use of Intel SIMD instructions for 4-way parallelism. Because Jitblt does not use any special SIMD instructions, this test is the best advantage scenario for Pixman.

The group of images used for each test was scaled to three sizes: small (20x8 pixels), medium (144x64 pixels) and large (576x256 pixels). The small size is typical of the area required to composite a glyph (i.e., text character) onto a background. The medium size is typical of a small icon or geometric shape. The large size is similar to the area occupied by digital photograph or application window.

In no case is Jitblt faster than Pixman. For this reason, Figure 3.3 represents the performance in terms of slowdown (Jitblt time / Pixman time). At its best, Jitblt is 1.28 times slower than Pixman. Being that very little time was spent on optimizing Jitblt or COLA, the fact that Jitblt is comparable to Pixman is promising. At its worst, Jitblt is 7.41 times slower than Pixman. This is discouraging, but not unexpected. Suggestions for improvements that narrow this performance gap are discussed in detail in Section 4.2

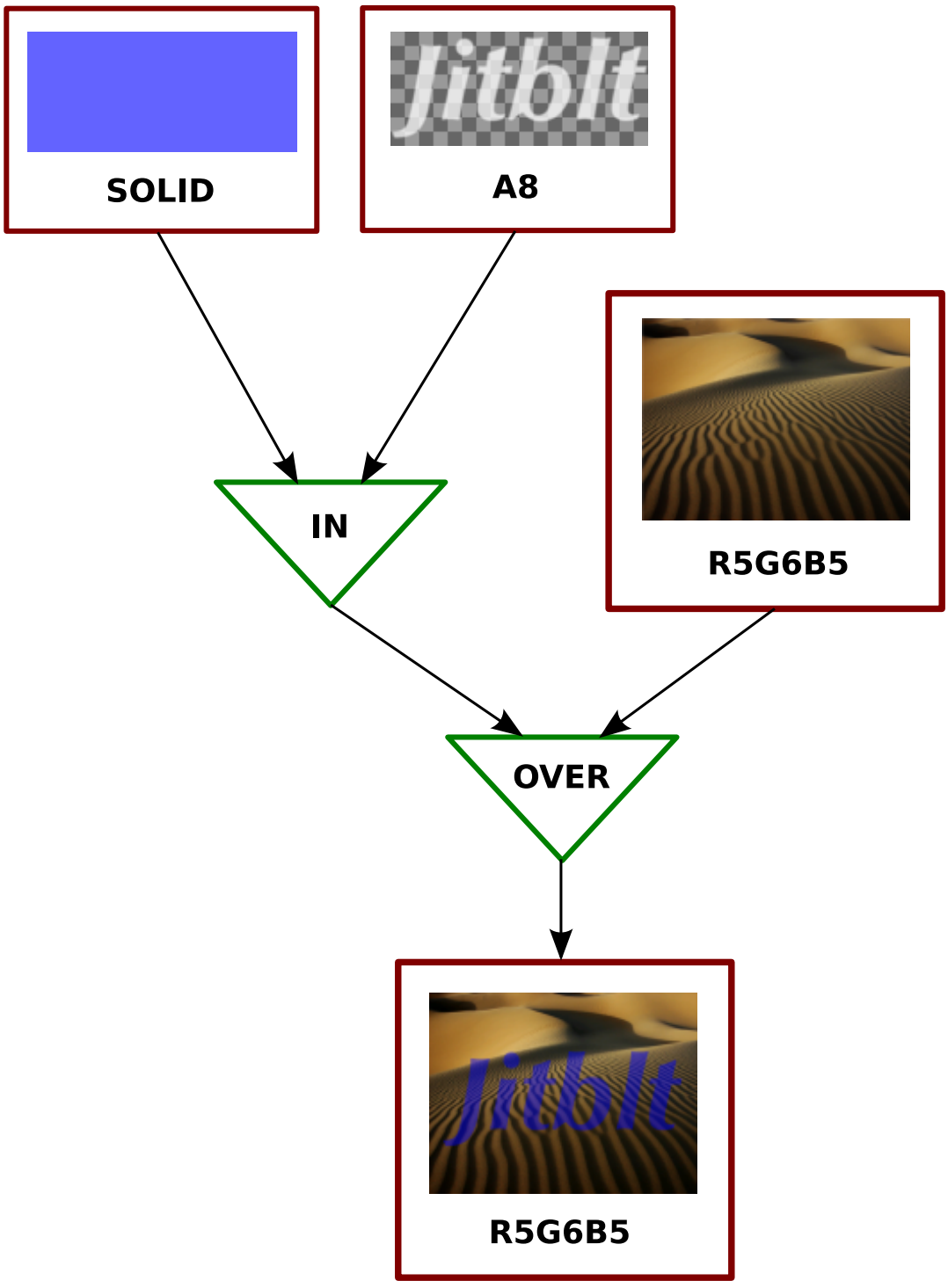

Figure 3.2: Sample Pipeline 2

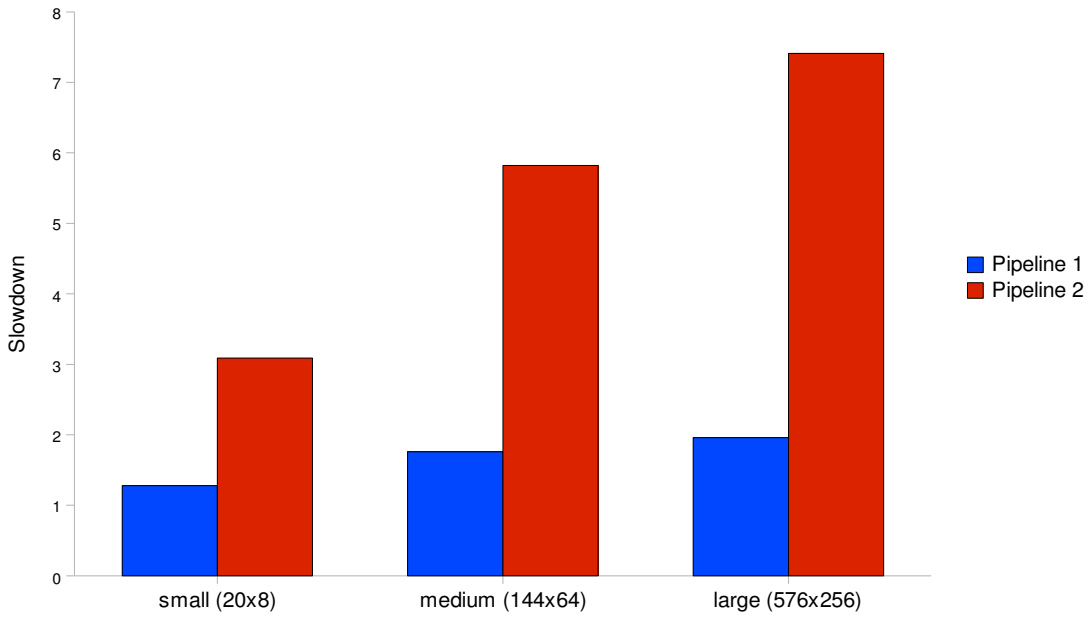

Figure 3.3: Performance Comparison

and Appendix A. It is clear, though, that significant work is necessary to make Jitblt a practical real-time compositing library.

Chapter 4

Conclusion

4.1 Advantages

The most striking advantage of Jitblt, clearly shown in Section 3.4, is how few lines of code its implementation has. This order of magnitude reduction in size leads to better code comprehension and increased maintainability. Fewer lines of code often mean fewer bugs. Extending the functionality of Jitblt scales well, also. For example, adding a new pixel format or compositing operator to Jitblt takes between 1 to 17 lines of code. Adding a new pixel format or compositing operator to Pixman's general compositing path takes about 50 lines of code. Adding a specialized code path in Pixman typically takes 50 to 100 lines for each combination of compositing parameters.

This extensibility is important for the future. When Adobe's Flash Player added new blend modes (Adobe's terminology for compositing operators), several requests were made to add them to Pixman. The effort to do so, especially in optimized form, was too great and the addition was never made. The introduction of new pixel formats has also been slow. High precision pixel formats (e.g., floating-point formats, wide integer formats) are becoming increasingly popular for high quality or high dynamic range images. Their adoption into Pixman has also been delayed for the same reason. High-level declarative approaches like Jitblt make extensibility easier, and thus are better

suited to adapt to future directions in image compositing.

4.2 Disadvantages

Jitblt has several limitations, some of which are related to the current state of the implementation, and some of which are inherent to the overall approach taken by Jitblt. Chapter 3 and Appendix A cover several limitations of the implementation. In addition to those already mentioned, further optimizations include better constant folding, SIMD instruction utilization, and data-dependent optimization. Images tend to have large fully-transparent or fully-opaque regions. Significant speedup can be had by looking ahead several pixels at a time to determine if compositing calculations can be skipped or simplified for a given region. For some composting operator and pixel format combinations, a simple memory copy is sufficient and much faster than the multistage pipeline Jitblt generates.

These optimizations can be difficult to incorporate cleanly into Jitblt. This is due to the high level of abstraction that Jitblt uses. Optimizations that must reach across pipeline stages, or across pipeline invocations are very effective, but go against certain assumptions in Jitblt. SIMD instruction utilization requires analysis across pipeline invocations to determine how to combine computation. Some optimizations can be achieved by introducing additional, more sophisticated compiler optimizations. But some, like the transparency look ahead technique described above, are data-dependent. In brief, it is not clear how to cleanly introduce several important optimizations into a high-level system like Jitblt.

4.3 Current Trends in Real-time Compositing

Since the initial Bit BLT implementation in 1975 [12], real-time compositing has evolved extensively. Features like transparency, image transformations and additional compositing operators have been introduced to create the rich visual results that are

commonplace today. Jitblt was designed to provide these capabilities, yet there remains the question of how to design Jitblt, or any compositing library today, to be as "future proof" as possible.

Like most systems in use today, Jitblt was designed to provide a fixed number of compositing operators (e.g., `OVER`, `ATOP`). In recent years, not only have additional compositing operators been introduced, but more general, programmable ways of manipulating and combining images have arisen. These developments are closely related to, and made feasible by, the development of shading languages for 3D graphics [11, 28, 6]. These languages give the programmer very general, fine-grained control over the output value of each pixel in the image. All the Porter-Duff compositing operators can be easily expressed in these shading languages [20].

Several 2D graphics libraries have adopted a subset of these 3D shading languages for end-user image processing (including custom compositing). Prominent examples include Apple's Core Image Kernel Language[1] and Adobe's Pixel Bender.[2] Like Jitblt, both utilize run-time code generation for high-performance. Both use the LLVM compiler [19] for run-time code generation.[3]

It is clear that much like 3D fixed-function graphics hardware has given way to 3D programmable graphics hardware, 2D graphics compositing operators are being replaced by pixel manipulation languages. The static functionality of yesterday can now be considered a special case or prepackaged piece that will likely remain conveniently available. But, for a compositing library to be relevant in the future, I believe that it must look beyond simple compositing and provide programmable pixel manipulation.

[1] http://developer.apple.com/documentation/GraphicsImaging/Reference/CIKernelLangRef/index.html

[2] http://labs.adobe.com/wiki/index.php/Pixel_Bender_Toolkit

[3] http://llvm.org/ProjectsWithLLVM

Appendix A

The COLA Programming Environment

Jitblt was implemented in a new programming system called COLA (Combined Object Lambda Architecture) [23]. COLA is part of the STEPS project currently undertaken by Viewpoints Research Institute [16]. COLA is a minimalist, dynamic, late-bound software system created by Ian Piumarta. It provides an object model, a small library of objects, and a dynamic compiler.

The COLA object model is named "id," and is described in detail in Piumarta and Warth [25]. Id defines a very minimal structure and mechanism for object communication based on message passing. In addition, it provides garbage-collected memory allocation, dynamic loading, and several fundamental objects. The various pixel formats and compositing operators in Jitblt are represented by id objects. These objects generate code in response to messages sent to them during Jitblt pipeline compilation.

The COLA dynamic compiler generates machine code from S-expressions akin

```
; defines a function that returns the square of the argument
(define square
  (lambda (a) (* a a)))

; applies the function
(square 3)
```

Figure A.1: Example of the COLA S-expression Language

```
; defines a function that returns a dynamically–compiled function
(define compile–binary–function
  (lambda (operator)
    (let ((code `(lambda (a) (,operator a a))))
      [code _eval])))

; compiles a function
(define square (compile–binary–function '*))

; applies the function
(square 3)
```

Figure A.2: Example of Dynamic Code Generation in COLA

to those of the LISP programming language [24]. The basic syntax is shown in Figure A.1. Not only is Jitblt written in this language, Jitblt also uses this representation for its own run-time code generation. The quasiquote feature of LISP, also available in the COLA dynamic compiler, proved invaluable. In fact, approximately 20% of the lines of code in Jitblt are quasiquote expressions. Figure A.2 illustrates a simple use of quasiquote to dynamically compile a binary function.

The square brackets that appear in Figure A.2 are syntactic sugar for message sending similar to method invocation syntax in the Objective-C language. The expression `[code _eval]` expands to `(send '_eval code)` which sends the object referenced by `code` the message `_eval`. In this particular example, the object receiving the message is of type `Expression`, a type provided and used internally by the COLA compiler. Sending this object the message `_eval` will cause it to evaluate itself and return a reference to the generated machine code (i.e., a function pointer).

COLA includes a C API for creating and communicating with a COLA compiler from external languages. This is available in the form of a static library that embeds the entire COLA system (libjolt). The library exposes a single C function for creating a COLA compiler object. Once the compiler object is created, the compiler can evaluate COLA S-expressions encoded in ASCII C strings. This mechanism made it possible to bridge Pixman (written in C) and Jitblt (written in COLA).

The use of COLA to create Jitblt has validated COLA as a powerful metaprogramming system. The principles upon which COLA is built have indeed been shown to make the task of domain-specific run-time specialization much easier. At the same time, work on Jitblt has exposed some aspects of the COLA implementation (in its current state) that prevent Jitblt from being of much practical use.

As mentioned in Section 3.6.1, COLA imposes a large penalty on application start up time. On a 1.8 GHz Pentium M processor, the delay is about 3.2 seconds. On a 330 MHz ARM11, this delay surpasses 30 seconds. Part of the issue is the speed of compilation and code generation, especially when compiling expressions containing quasiquotation. The biggest issue, though, stems from the fact that at every startup, the COLA system builds itself from "first principles". This, in addition to the Jitblt source code being compiled at startup, leads to the lengthy pause. Current work on a COLA static compiler will hopefully alleviate this issue.

Another issue that came up during the development of Jitblt relates to the efficiency of the code that is generated at run-time by Jitblt. COLA generates relatively efficient code for x86 processors. For ARM processors, though, the generated code can perform several times slower than code generated from a C compiler. This makes Jitblt very impractical for ARM-based systems. Unfortunately, these systems have a special need for fast compositing on the CPU since they often lack graphics hardware with decent compositing capabilities.

Even on x86 systems, Jitblt was not able to achieve adequate performance for several reasons. Because COLA does not provide any compiler optimizations (e.g., constant folding, CSE), optimization had to be done within Jitblt itself. Unfortunately, there were several optimizations that could not be completed due to time restraints. COLA's use of a very simple linear scan register allocator means that register spilling occurs more frequently in code generated by Jitblt than in code generated by an optimizing compiler. Even more limiting is the fact that COLA stores all local variables on the stack (never in registers), leading to redundant memory accesses. The small, frequently-executed loops generated by Jitblt are very sensitive to these factors.

As detailed in Section 3.5, COLA can have a large memory footprint. The size of the libjolt static library is 2.7 MB. This isn't a severe problem on desktop computers, but it can be an issue on embedded devices. A more serious problem is run-time memory allocation. Jitblt-Pixman consumes about 12 times more resident memory than Pixman. This is due to issues with the conservative, non-copying garbage collector. Simple changes to the configuration of the garbage collector may solve the out-of-control memory consumption. There are also plans to replace the garbage collector entirely.

References

[1] Aho, A., Sethi, R., and Ullman, J., 1986: *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[2] Beyer, W., 1964: Traveling Matte Photography and the Blue Screen System. *American Cinematographer*, 266.

[3] Blinn, J., 1994: Compositing, Part 1: Theory. *IEEE Computer Graphics and Applications*, **14**(5), 83–87.

[4] Blinn, J., 1994: Composting, part 2: practice. *IEEE Computer Graphics and Applications*, **14**(6), 78–82.

[5] Blinn, J., 1997: Fugue for MMX [parallel programming]. *Computer Graphics and Applications, IEEE*, **17**(2), 88–93.

[6] Blythe, D., 2006: The Direct3D 10 system. *ACM Transactions on Graphics (TOG)*, **25**(3), 724–734.

[7] Brinkmann, R., 1999: *The Art and Science of Digital Compositing*. Morgan Kaufmann.

[8] Draves, S., 1996: Compiler Generation for Interactive Graphics Using Intermediate Code. *Dagstuhl Seminar on Partial Evaluation*, 95–114.

[9] Draves, S., 1997: *Automatic Program Specialization for Interactive Media*. Ph.D. thesis, Carnegie Mellon University.

[10] Elliott, C., Finne, S., and de Moor O., 1999: Efficient Image Manipulation via Run-time Compilation. Technical Report TR-99-82, Microsoft Research.

[11] Hanrahan, P., and Lawson, J., 1990: A language for shading and lighting calculations. *ACM SIGGRAPH Computer Graphics*, **24**(4), 289–298.

[12] Ingalls, D., 1975: Bit BLT (November 19, 1975). Xerox Memo.

[13] Ingalls, D., 1981: The Smalltalk graphics kernel. *Byte*, **6**(168).

[14] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A., 1997: Back to the future: the story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*, **32**(10), 318–326.

[15] Jones, N., Gomard, C., and Sestoft, P., 1993: *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

[16] Kay, A., Ingalls, D., Ohshima, Y., Piumarta, I., and Raab, A., 2006: Steps Toward The Reinvention of Programming. VPRI Research Note RN-2006-002, http://vpri.org/pdf/NSF_prop_RN-2006-002.pdf.

[17] Keppel, D., Eggers, S., and Henry, R., 1991: A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington.

[18] Lattner, C., 2007: LLVM for OpenGL and other stuff. LLVM Developers Meeting, http://llvm.org/devmtg/2007-05/10-Lattner-OpenGL.pdf.

[19] Lattner, C., and Adve, V., 2004: LLVM: a compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 75–86.

[20] Nilsson, P., and Reveman, D., 2004: Glitz: Hardware Accelerated Image Compositing Using Opengl. *Usenix 2004 Annual Technical Conference, Freenix Track*, 29–40.

[21] Oram, A., and Wilson, G., 2007: *Beautiful Code*. O'Reilly Media, Inc.

[22] Pike, R., Locanthi, B., and Reiser, J., 1985: Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software - Practice and Experience*, **15**(2), 131–151.

[23] Piumarta, I., 2005: Accessible language-based environments of recursive theories. VPRI Research Note RN-2006-001-a, http://vpri.org/pdf/colas_wp_RN-2006-001-a.pdf.

[24] Piumarta, I., 2006: Coke programming guide. http://piumarta.com/software/cola/coke.html.

[25] Piumarta, I., and Warth, A., 2007: Open reusable object models. VPRI Research Note RN-2006-003-a, http://vpri.org/pdf/obj_mod_RN-2006-003-a.pdf.

[26] Poletto, M., Engler, D., and Kaashoek, M., 1997: tcc: a system for fast, flexible, and high-level dynamic code generation. *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, 109–121.

[27] Porter, T., and Duff, T., 1984: Compositing digital images. *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 253–259.

[28] Rost, R., 2004: *OpenGL Shading Language*. Addison-Wesley Professional.

[29] Smith, A., 1995: Alpha and the History of Digital Compositing. Microsoft Tech Memo.

[30] Thompson, K., 1968: Programming Techniques: Regular expression search algorithm. *Commun. ACM*, **11**(6), 419–422.

[31] Wright, S., 2006: *Digital Compositing for Film and Video*. Focal Press.