

**DYNAMICALLY GENERATED ASSEMBLY BLITTER FOR S40
MOBILE PHONES**

Heikki Hellgren
Bachelor's thesis
Spring 2011
Information Technology and
Telecommunications
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology and Telecommunications

Author(s): Heikki Hellgren

Title of Bachelor's thesis: DYNAMICALLY GENERATED ASSEMBLY BLITTER FOR S40 MOBILE PHONES

Supervisor(s): Mr. Markku Rahikainen (OUAS), Mr. Mikko Polojärvi (Nokia Corporation)

Term and year of completion: Spring 2011 Number of pages: 49 + 6 appendices

This bachelor's thesis was done for Nokia Corporation S40 Mobile Phones. In this thesis, the problems and possible solutions were discussed for fast two-dimensional software graphics processing. The objective of the study was to design, implement and test a software graphics processing component created by using dynamically generated assembly. The target of the software component created is mobile phones with a processor following the ARM11 processor family architecture.

The objective of this thesis came from the actual needs for faster software graphics processing in S40 Mobile Phones. Before starting this thesis, the author had some previous experience of S40 architecture and computer graphics in general. The problems and solutions of this thesis were researched by studying previous implementations of two dimensional software graphics components and by doing a great deal of pre-studying of computer graphics literature.

The Results of this bachelor's thesis were significant. Most of the graphics processing cases benchmarked were much faster compared to a previous implementation. Further development of the work includes tweaking the performance of the most used use cases, implementation of support for other processor architectures and expanding the software component with new features.

Keywords: Self-modifying code, blitter, S40, mobile phone, computer graphics

CONTENTS

SYMBOLS	1
1 INTRODUCTION	2
2 COMPUTER GRAPHICS.....	3
2.1 Basics of computer graphics	3
2.2 Graphics processing.....	5
2.3 Computer graphics in mobile devices.....	5
2.4 Graphics rendering chain	7
2.5 Block Image Transfer	8
2.5.1 Color conversion	9
2.5.2 Transformations	11
2.5.3 Scaling	14
2.5.4 Alpha compositing.....	17
3 ARM 11 PROCESSOR FAMILY	22
4 DYNAMIC ASSEMBLY	25
5 SPLATTER	28
5.1 Stitcher and code generation component.....	29
5.2 Test environment.....	32
5.3 Pixel pipeline generation	33
5.4 Supported formats and blit types	35
5.4.1 Blit types.....	36
5.4.2 Transformations	38
5.5 Performance tweaking.....	39
6 RESULTS AND CONCLUSIONS.....	40
7 DISCUSSION.....	43
LIST OF REFERENCES	44
APPENDICES.....	48

SYMBOLS

2D	Two-dimensional
3D	Three-dimensional
ALU	Arithmetic logic unit
API	Application programming interface
BLIT	BLock image transfer
BPP	Bits per pixel
CCW	Counterclockwise
CLI	Command-line interface
CPU	Central processing unit
CW	Clockwise
GUI	Graphical user interface
GPU	Graphics processing unit
PC	Personal computer
PDA	Personal digital assistant
RGB	Red, green and blue color model
RISC	Reduced instruction set computing
SIMD	Single instruction, multiple data
YUV	Luma (Y) and chrominance (UV) color space

1 INTRODUCTION

The number of applications and devices using computer graphics has grown exponentially during the last decades. The rapid development in mobile technologies has played a major part in this growth. This bachelor's thesis discusses computer graphics in mobile devices and specializes in two dimensional graphics processing.

Generally speaking every device that has a display needs a way to create a graphical output into the display. This can be done with either software or hardware. Hardware processing means that the device has a dedicated hardware for graphics processing. Software processing is carried out by the CPU (Central Processing Unit) of the device. While software processing is less expensive and easier to implement, it usually lacks the speed of the hardware accelerated graphics. For this reason the software processing approach is usually more interesting, in particular when producing high volume products.

The theoretical part of this study discusses methods, problems and possible solutions when working with software processed two dimensional graphics. The empirical part of the study includes a description and the basic idea of the functionality of an implemented dynamically generated assembly blitter for S40 mobile phones. Blitter, which stands for the component doing BLock Image Transfer, is explained later on in the study. The product of this study was tested against a previous implementation and the results of the work can be seen in the tests presented at the end of this study.

I would like to thank Mr. Mikko Pölojärvi and Mr. Jani Lamberg from Nokia Corporation for sharing their experience in graphics processing and S40 architecture. Without them it would have been impossible to finish this study.

In this study some of the subjects are described at a general level because the actual work contains information that is not intended for the public domain.

2 COMPUTER GRAPHICS

Computer graphics have existed almost from the very beginning of the computer age. The first known GUI (Graphical user interface) was developed in the 1960's by a MIT student, Ivan Sutherland, and it allowed him to use a light pen to draw images onto the screen of a computer (1). For a long time, computer graphics were related to personal computers. Today computer graphics are also included in PDAs, mobile phones and other handheld devices. Generally speaking computer graphics are a part of every electronic device that has a display.

2.1 Basics of computer graphics

Computer graphics is a very large concept. Roughly speaking computer graphics could be divided into 2D (two dimensional) and 3D (three dimensional) graphics. Though all computer graphics operations produce two dimensional images with width and height, 3D graphics are used to create an illusion of depth in the image. Three dimensional graphics are widely used in the video game industry as well as in movies. In addition, the most powerful handheld devices today provide games and other content utilizing three dimensional graphics.

This study mainly focuses on 2D graphics which are still the most commonly used type of graphics. The two main areas of two dimensional graphics are vector and raster graphics which again can be divided into numerous categories.

Vector graphics are created with geometrical primitives such as points, lines, curves and shapes (2). One benefit of using vector graphics is that vector graphics can be zoomed in endlessly because the image is always redrawn depending on the required zoom level. Vector graphics consume less memory than raster graphics because only the drawing primitives are saved. Vector graphics are used in many applications such as the Google Maps – service (3).

Most graphical applications are based on raster graphics. Raster graphics, as in pixmaps, are a presentation of an image with pixels (4; 5). Bitmaps are a presentation of image data with zeros and ones and they are widely used in every aspect of computer graphics. A benefit of raster graphics compared to vector graphics is that the saved data can be presented in great detail. For example, all images taken with a normal digital camera are saved as bitmaps. Most of the S40 GUI utilizes bitmaps for creating menus, numbers, letters and other images that can be seen on the display of the mobile phone.

Bitmap consists of a width times height number of pixels. This multiplication is called the resolution of the image, for example 800 x 600 pixels. Pixel is a single point in a raster image and it is the smallest accessible unit in the image. The actual color components of the pixel are saved to the memory usually in RGB (red, green, and blue) color space. For example, the pixel selected in the figure (see figure 1) would be saved to the memory as a hexadecimal value of 0xF1B590 if the presented image was created using a 24-bit RGB color. The first eight bits from the left of the color value represent the amount of red in the pixel, the next eight bits the amount of green and the last eight bits the amount of blue. These values are seen in the color component chart, which is a part of the figure.

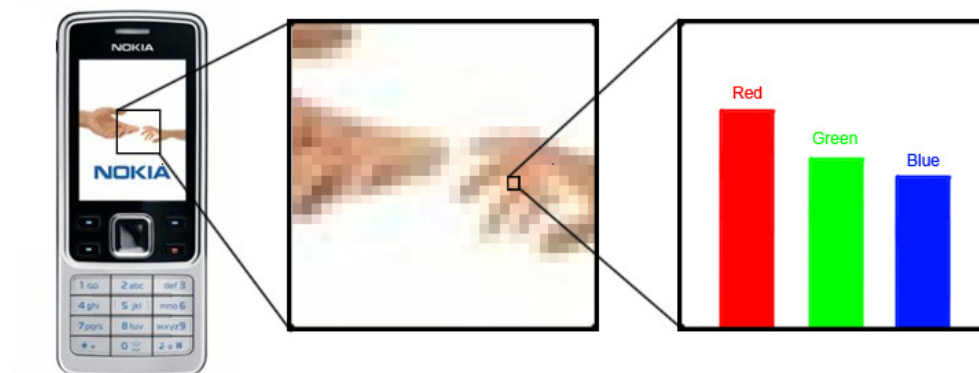


FIGURE 1. Pixels of Nokia logo presented on the display, zoomed in and in a color component chart

2.2 Graphics processing

Every graphics processing operation is either done with hardware or software. The hardware assisted graphics processing means that the device has a dedicated hardware that is only used for graphics processing. Most PCs (personal computers) today utilize hardware accelerated graphics and they usually include one or more chips for graphics processing. These chips are generally known as GPUs (graphics processing unit). In personal computers, the CPU and the GPU are commonly physically separated to different parts and they communicate through a bus. In mobile devices where hardware acceleration is used, the GPU is usually located in the same chip with the CPU.

Software graphics processing utilizes the CPU of the device and, by doing so, it also uses processor cycles from other CPU operations. If the CPU contains only one core, the impact of the graphics processing in the same core with other operations is even greater than in multi-core processors where one or more cores can be assigned to handle only graphics processing. Unfortunately, most of the mobile device processors still have only one core processor, though multi-core CPUs are emerging the mobile market as well.

2.3 Computer graphics in mobile devices

Computer graphics in mobile devices have existed since the first mobile device with a display was made. The first displays were small with small resolution and they could only show black and white images. Development has exploded from those times, and the displays of today can show increasingly high resolutions. Resolution in the display means the number of pixels the display can show. In mobile phones resolution is usually near QVGA (quarter video graphics array, 320 x 240 pixels), whereas most PCs show HD 1080 (high definition, 1920 x 1080 pixels) or even higher resolutions. When the resolution of the display increases, higher

performance is required from the graphics hardware or software graphics processing simply because there are more pixels to draw and manipulate.

Most of the graphical output of mobile devices is created using two dimensional graphics. This includes all bitmap and vector graphics operations as well as general drawing primitives such as drawing lines, circles, triangles and rectangles to bitmaps. Generally speaking, every GUI in mobile devices is currently made by using 2D graphics only.

Because of their slower CPUs compared to those of PCs and usually even the lack of hardware accelerated graphics, mobile devices need a highly optimized way to draw graphics to the display with software. There are some hardware graphics accelerators for mobile devices in the market but they are quite expensive compared to a very well implemented software graphics solution. For this reason, they are usually avoided in mass-produced products such as mobile phones.

Computer graphics in more powerful devices such as PCs and gaming consoles have already utilized three-dimensional graphics for several decades. For mobile phones this area is developing fast. 3D graphic APIs (application programming interface) such as Khronos Group's OpenGL ES (5.) are already implemented in the most powerful mobile devices and are mostly used by third-party developers. The main problem in using three-dimensional graphics in mobile devices is that a great number of calculations and high performance is needed to generate three dimensional images.

In the future, it will be possible to create completely three-dimensional GUIs for mobile devices. To achieve this, mobile devices will have to develop fast enough to render the considerable amounts of data needed by 3D graphics processing.

2.4 Graphics rendering chain

For the image to appear on the display it must go through a graphics rendering chain. The rendering chain includes various graphics operations before ending into the display as seen in the simplified figure below (figure 2).

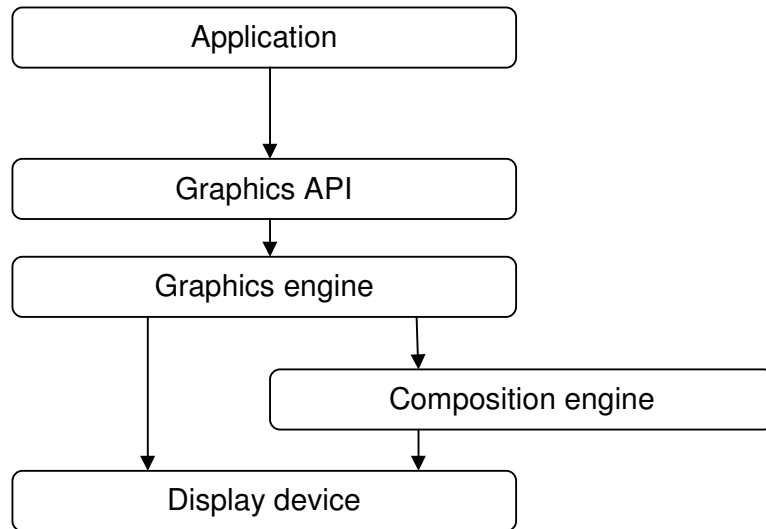


FIGURE 2. Simplified presentation of graphics rendering chain.

Firstly, an application is needed to create the required content by using graphics API. This content can be either 3D or 2D depending on what the application is designed to draw and what kinds of API and graphics engine are used.

2D graphics in PCs are usually drawn using the operating system's own API such as Microsoft GDI+ (7.) or DirectDraw (8.) in Microsoft Windows. 2D graphics in mobile devices also utilize their own internal graphics engine or a public graphics engine such as Qt (9.). For vector graphics, OpenVG (10.) is used both in mobile devices and in PCs.

The best known 3D -graphics APIs are OpenGL (11.) and Microsoft's DirectX (12.) for PCs and OpenGL ES for mobile devices. All graphic APIs include a

large list of functions prototypes, type definitions, defines and other functions for the application to create the required content. The mastering one of these APIs, for example OpenGL, can take several years from a typical programmer.

Secondly, after the application has created the required content, the graphics engine under the API creates a raster image of one or more source images or object data from the application. Output from the graphics engine is then optionally run through a 2D composition engine to create a full screen image. The composition engine may use a 2D blitter to create the required output. In addition, the graphics engine itself can be a simple 2D blitter. Blitters are discussed in more detail in the following chapter 2.5. After the composition is completed, the full screen image is transferred to the display using DMA (direct memory access).

The composition part can also be overridden if the application is running in the full screen mode. Full screen mode means that the output resolution of the graphics engine is equal to the resolution of the display and there is no other data drawn over the output. In the full screen mode, the output of the application is transferred directly to the display device. In some cases, this can speed up performance because no composition is needed and many graphics operations can be skipped.

2.5 Block Image Transfer

Bit blit (block image transfer) is an action in the software or hardware graphics processing where chunks of two dimensional image data are moved from one memory location to another (13). The component carrying out the bit blit is called a blitter. The history of the block image transfer dates back to 1975 when a component called BitBlit was developed for Xerox Alto computer by Dan Ingalls, Larry Tesler, Bob Sproull, and Diana Merry. Around the 1980's, computer manufacturers developed graphics coprocessors containing a blitter to decrease the workload of CPU (14). This marked the start of the hardware accelerated graphics as we know it today.

Later, the coprocessors called GPUs or even separate graphics hardware acceleration cards have been incorporated in blitters to boost the graphics performance of devices. Some of the modern blitters such as SDL (15) still use software for carrying out the operations (16).

The basic functionality of a blitter is to run through the destination surface pixel by pixel and write the modified or unmodified source pixel data to the target. This operation is also known as a raster scan. (17.) The pixel by pixel operation is performed in two loops. For example, 200 x 200 pixels' target area is run 200 times in the vertical loop (y-axis) and 40,000 times through the horizontal loop (x-axis). As the horizontal, or inner loop, is run a great number of times, it is the most potential place to search for possible performance problems at the blitter.

The most common use case for the blitter is to blit the source surface directly to the target surface as seen in the figure (figure 3).

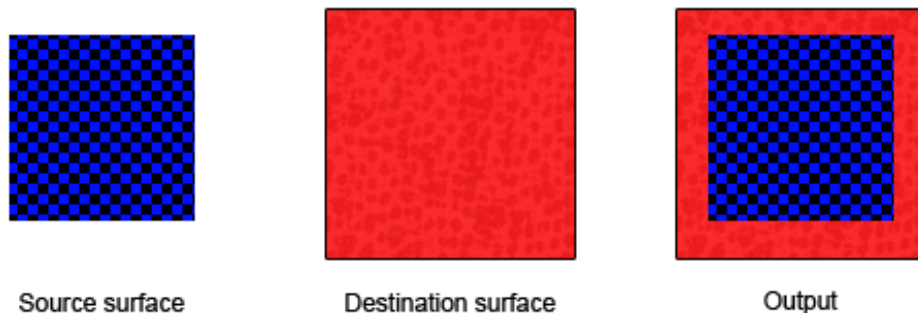


FIGURE 3. Source surface blit to destination

2.5.1 Color conversion

Most of the blitters can also handle many types of graphical surfaces and blits between surfaces in different pixel formats. The action of changing the pixel format between the source and the destination is called color conversion. To carry out the color conversion, the blitter has to know the pixel formats of both the source and the destination. All color conversion

operations must be done pixel by pixel and they usually take place in the early stage of the blit operation.

RGB is the most commonly used pixel format in computer graphics. Even though the colors are usually in the same order, RGB values can also be presented in different sizes in the memory. The most used RGB types are 24-bit RGB, 16-bit RGB and 12-bit RGB. In the 24-bit mode, each color and the possible alpha channel are represented in eight bit long chunks. If the alpha channel is included in the 24-bit RGB, the amount of the memory required is 32 bits per pixel. In the 12-bit color, RGB chunks are four bits long for each component. Equally, if an alpha channel is included, the amount of the memory increases up to 16 bits per pixel. The 16-bit RGB differs from these formats in the way that it does not support the alpha channel. The 16-bit RGB is also called 565 RGB, which stands for 5 bits of red, 6 bits of green and 5 bits of blue for each pixel. RGB color formats are usually named with the amount of bits per component, for example, 4444 ARGB stands for a 12-bit RGB with an alpha channel.

Another commonly used pixel format is Y'CbCr. This format is mainly used in video processing and image compression applications. The format is very often described as YUV, which stands for an analogue video signal system (18). When digital video or image processing are concerned, YUV actually refers to the Y'CbCr color model.

Similar to RGB color modes, YUV is also saved to the memory. There is quite a few different types of YUV formats, which all represent the same data but are saved to the memory in slightly different ways. The best known YUV formats are YUV420 and YUV422. YUV or Y'CbCr presents information not in colors but in luma and color differences. Luma (Y) defines the light intensity at pixel, whereas the color difference as in chroma (Cb and Cr) defines the shade of the pixel. With YUV packing it is possible to save in memory consumption by subsampling the chroma data.

There are also other kinds of pixel formats that are used in computer graphics. For example, the 1-bit, 4-bit and 8-bit alpha modes are used for creating one color content. These color modes only contain alpha channel information and the actual color must be obtained from some other source. These color modes are useful, for instance, when drawing one color fonts because they fit in much smaller memory blocks than, for example, a full 32-bit ARGB font would.

Usually, color conversion is done from the source pixel format to the destination pixel format. By doing this, other operations, such as alpha blending, need only one color conversion instead of first converting the target pixel format to the source pixel format, carrying out the alpha blending and then converting the result back to the pixel format of the destination.

When converting from a source format that does not have enough bits per color component for the destination format, the blitter needs to fill extra bits with the MSB (most significant bit) or MSBs of the source in order to perform correct conversion. This is necessary for example when converting from the 4444 ARGB format to the 565 RGB mode (see figure 4).

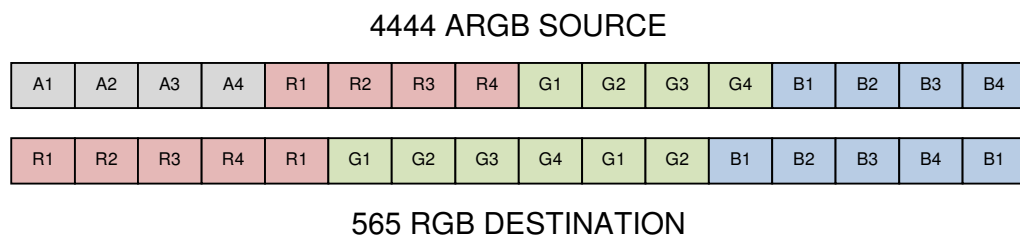


FIGURE 4. An example of color conversion from the 4444 ARGB format to the 565 RGB format presented in bit level.

2.5.2 Transformations

One of the most essential features of a good blitter is its ability to support transformations. Transformations stand for modifying the source surface before the actual blit. Usually the blitters support various transformations

such as rotation, clipping, scaling and mirroring. A common case is that transformations are performed by using matrix algebra, except in low-level blitters where using matrices is really slow and transformations are carried out by using more simple mathematics.

Rotation is used in many cases, for example, when a handheld device is turned to the landscape mode. In this case, all graphic surfaces must be rotated and possibly scaled to fit the screen and show the correct output for the user. Rotation can be done in various ways. Usually blitters support only 90, 180 and 270 degree rotations but also free rotation with the custom amount of degrees is supported in more advanced blitters. In general, when the rotation is concerned, degrees are presented in the clockwise (CW) direction.

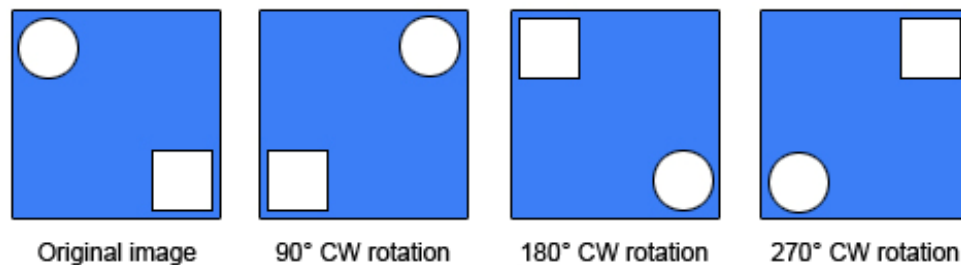


FIGURE 5. Clockwise rotation of the original image with fixed degrees of 90, 180 and 270.

Rotation in mathematics is performed using transformation matrices. A general 2D transformation matrix for the clockwise rotation is presented in the following formula (formula 1).

FORMULA 1. 2D clockwise rotation transformation matrix. (19.)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The matrix can be used to calculate the actual position of a given pixel after rotating it with θ degrees in the clockwise direction. The rotation is rather simple in particular when only fixed degree rotations are used because \sin and \cos equations with a degree dividable by 90 are always either 1 or 0 and only simple multiplications have to be done.

In mirroring, the source surface is mirrored to the opposite direction and then blitted to the target surface. This operation is also known as flipping in some graphics applications. Mirroring can be used for example to create graphical effects in GUI. Mirroring can be done in both vertical and horizontal axes. An example of mirroring a surface is presented in figure (figure 6).

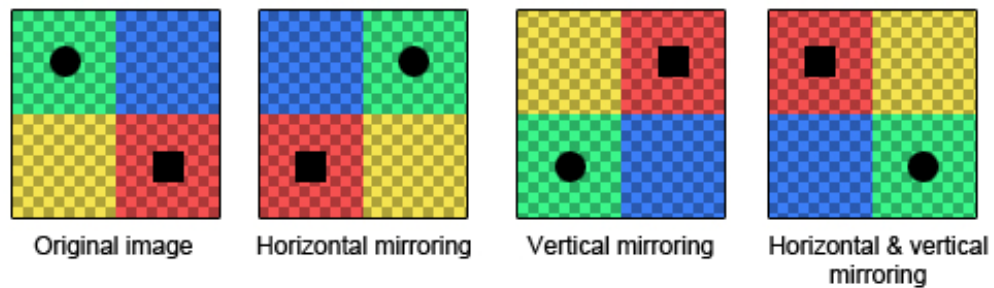


FIGURE 6. Horizontal and vertical mirroring of the original image

A frequently used feature in blitters is clipping. Clipping can be used to set source and destination rectangles to define where to and where from the blit is done. (20; 21) This saves time because there is no need to draw the source surface completely to the target and therefore less pixels must be handled in the rendering process. In the figure (figure 7), clipping is carried out from the area of the source surface marked with a red rectangle to the red rectangle area in the destination surface. If these rectangles are of different sizes, the blitter should also carry out the scaling for the correct output.

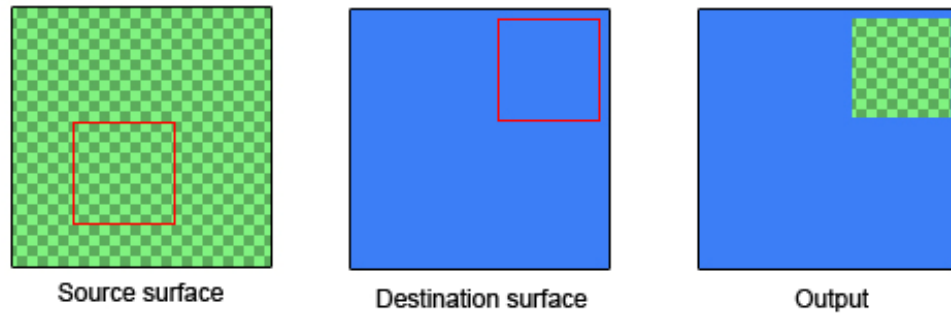


FIGURE 7. Clipping from the source to the destination

2.5.3 Scaling

Scaling is one of the most important features of a good blitter. It is also a part of the transformations that can be applied to a blit. Scaling means that the source surface or part of it is blitted on the target surface in different dimensions than it originally was. For example, when the source surface with the size of 200x200 pixels is blitted on a target surface in size of 300x300 pixels, the blitter has to perform scaling. There are various scaling algorithms for carrying out the operation, and each of them produces a different quality of scaling (figure 8).

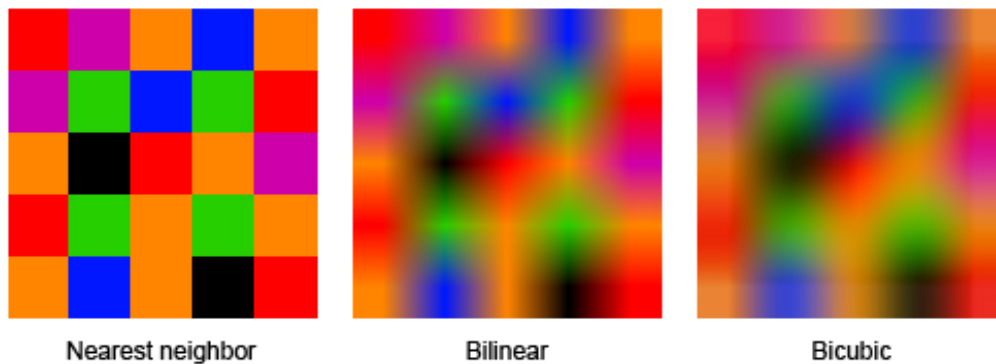


FIGURE 8. A 5x5 pixel area scaled to 70x70 pixels with the nearest neighbor, bilinear and bicubic scaling algorithms

As seen from the figure (image 8), the differences between scaling algorithms can be considerable. All scaling algorithms calculate the source pixel coordinates to match with the destination dimensions. The following formula (formula 2) is used to determine the values of source x and y coordinates of the pixel used for blitting.

FORMULA 2. Calculating source coordinates in scaling

$$x = \frac{source_{width} - 1}{destination_{width} - 1} * destination_x$$

$$y = \frac{source_{height} - 1}{destination_{height} - 1} * destination_y$$

The coordinates (x,y) calculated with this formula are non-integer values and must not to be considered as actual pixel values. In addition, in case of destination surfaces with the width or the height being either one or zero, the y or x values must be forced to be zero in order to prevent division by zero errors.

The nearest neighbor scaling uses the nearest pixel from the formula (formula 2). For example, if the coordinates in the formula are (3.125, 6.304), the source pixel used for blitting is rounded to (3,6). The nearest neighbor is the fastest scaling method but it also produces the worst quality, in particular when scaling up.

Bilinear scaling computes an interpolated value of colors from the four nearest pixels (2x2 pixel area) and uses the computed value for blitting with the target surface. The basic idea of bilinear interpolation is to first linearly interpolate in one direction and then to the other. (22.)

Each color component must be calculated separately when using bilinear scaling. Because interpolation is calculated between values ranging from

zero to one, each color component must be divided with its maximum value before applying the interpolation formula and then again multiplied with its maximum value to get the actual color value. For example, the maximum value of a 4-bit color component is 15, and it should be used as divider and multiplier when working with the 12-bit RGB pixel format.

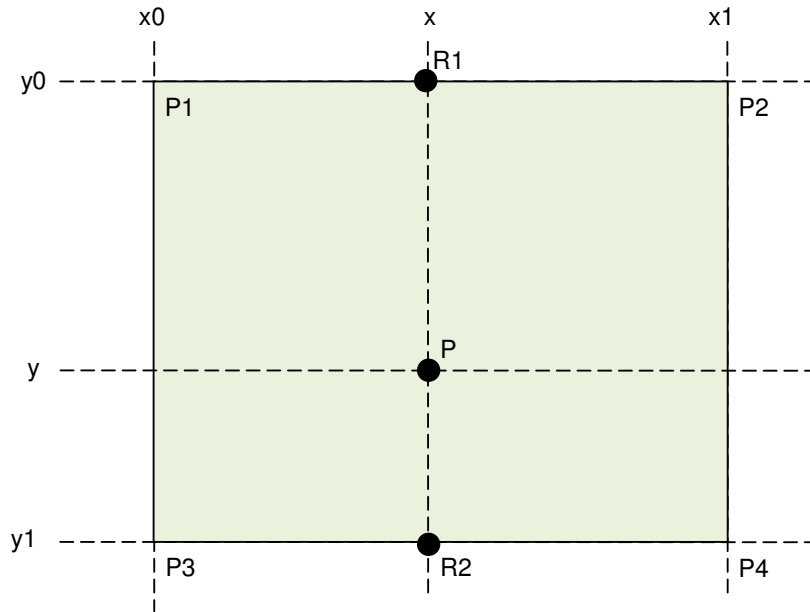


FIGURE 9. An example of bilinear interpolation.

By using the basic linear interpolation formula we can create the following formula (formula 3) for calculating the color value of P.

FORMULA 3. Linear interpolation of point P

$$C(R1) = \frac{x1 - x}{x1 - x0} * C(P1) + \frac{x - x0}{x1 - x0} * C(P2)$$

$$C(R2) = \frac{x1 - x}{x1 - x0} * C(P3) + \frac{x - x0}{x1 - x0} * C(P4)$$

$$C(P) = \frac{y_1 - y}{y_1 - y_0} * C(R1) + \frac{y - y_0}{y_1 - y_0} * C(R2)$$

Point P is the point of x- and y-coordinates calculated from the destination pixel point by using the source coordinate formula (see formula 3).

Bicubic scaling is the most expensive and also the best scaling algorithm. An example of bicubic scaling is shown in the figure (figure 8). Bicubic scaling uses the color values of 16 surrounding pixels to calculate the final pixel color value. In mathematics, this operation is called bicubic interpolation. (23.) Bicubic scaling is not exhaustively discussed in this study because it is not included in the empirical part of the study due to the limitations in the processor architecture.

There are also other scaling algorithms, which all work slightly differently, using, however, the same basic idea, for example the Bresenham scaling algorithm. It is difficult to select the best algorithm for each use case and therefore a good blitter should support many different algorithms.

2.5.4 Alpha compositing

Sometimes pixels may contain an alpha component, in addition to the color components. The alpha component stands for the transparency value of the pixel in the raster image. It was first introduced by A.R. Smith in the late 1970's (24). In 1984, Tom Duff and Thomas Porter created the theory of alpha compositing, which is widely used in today's computer graphics (25). Usually, the alpha component is described from zero to one where zero stands for fully a transparent and one for an opaque. The figure (figure 10) presents the basic operations of the Porter-Duff theory.

In the 32-bit ARGB mode, the alpha value is in the first eight bits of the pixel value and it therefore uses values from zero to 255. In the 16-bit ARGB mode, the alpha component is only four bits long and it can have values ranging from zero to 15. Because alpha values are not presented in a

common zero to one scale, the blitter must treat all alpha values in the scale of the pixel format.

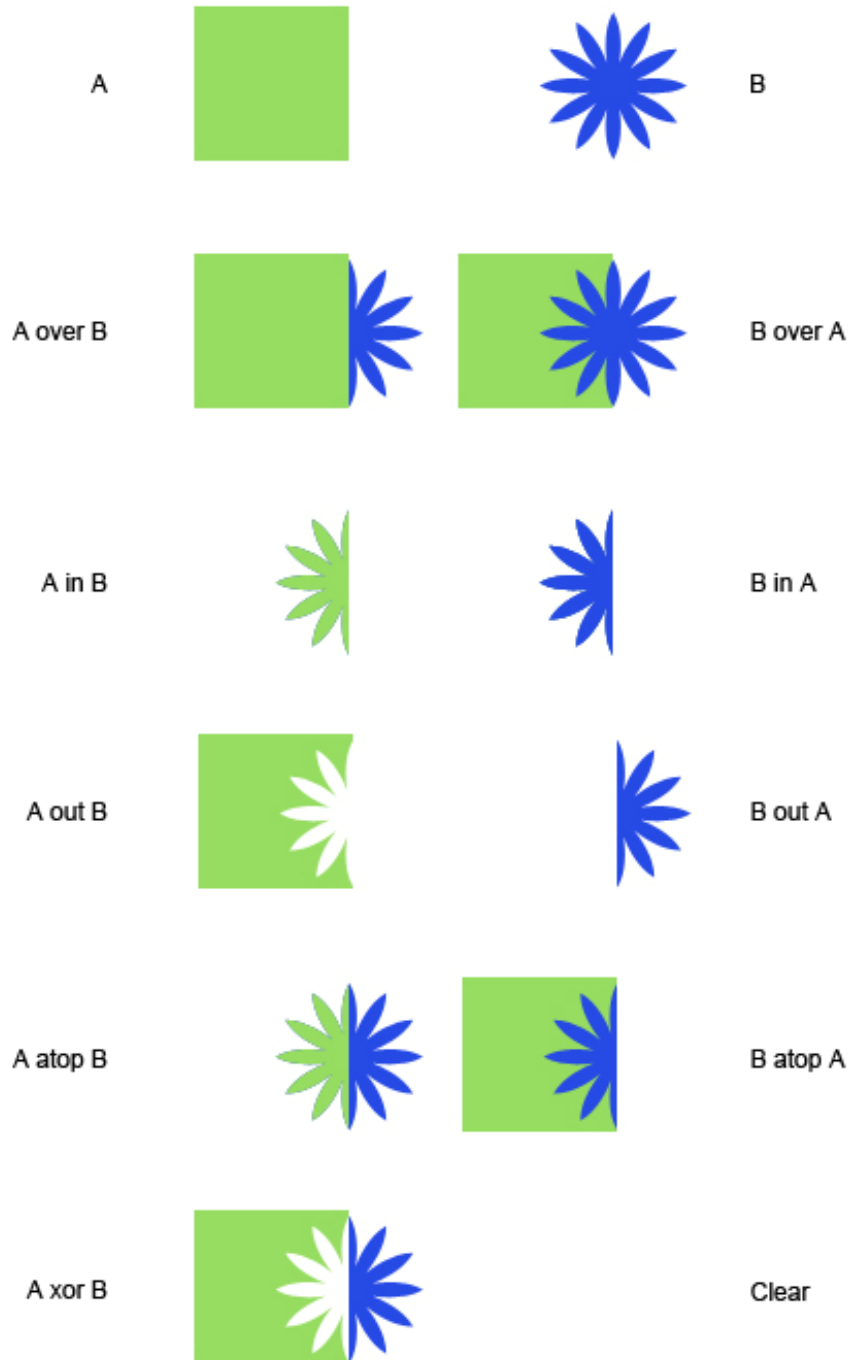


FIGURE 10. Porter-Duff alpha composition operations (26).

Porter-Duff operations can be achieved by changing the source and the destination factors in an alpha blending algorithm (see formula 4).

FORMULA 4. Alpha blending algorithm

$$C_0 = \frac{\alpha_a \cdot C_a \cdot f_a + \alpha_b \cdot C_b \cdot f_b}{\alpha_0}$$

The final alpha (α_0) of the pixel is calculated with the following algorithm (see formula 5). This formula is also used in the alpha blending algorithm to divide the component color with the final value when working with not pre-multiplied alpha components.

FORMULA 5. Final alpha algorithm

$$\alpha_0 = \alpha_a \cdot f_a + \alpha_b \cdot f_b$$

The alpha blending algorithm is used to calculate each color component of the pixel. The definitions of symbols in the formula can be found in the following table (table 1).

TABLE 1. Symbols in the alpha blending formula.

C_0	Color component to be calculated
α_a	Alpha value of the source pixel
C_a	Color component value in the source pixel
f_a	Blend factor for the source
α_b	Alpha value of the destination pixel
C_b	Color component value in the destination pixel
f_b	Blend factor for the destination

The most used source over destination alpha blend would use one (1) as a source factor and one minus source alpha ($1 - \alpha_a$) as a destination factor. In Porter-Duff alpha composition operation (figure 10) this would mean the A over B operation. Further information about blending factors can be found in chapter 5.4.1 of this study.

Pre-multiplied alpha is used in computer graphics for saving time in blending calculations. Pre-multiplied alpha means that the color components of a pixel are already multiplied with the alpha value. This simplifies the alpha blending formula and saves computer cycles by removing the heavy divide operation (see formula 6). The same formula can be used to calculate the final alpha of the pixel as well.

FORMULA 6. Pre-multiplied alpha blending algorithm

$$C_0 = C_a \cdot f_a + C_b \cdot f_a$$

Global alpha is a commonly used concept. Global alpha means that the result of the blit uses defined alpha value for multiplications instead of using source or destination alpha values. With the global alpha, it is possible to define the opacity of the whole blit with one value. The global alpha is usually defined in blit parameters by the user of the blitter.

Alpha composition can also be done without the alpha channel with either color keying or masking. Color keying is a technique where only some color of the target surface is replaced with the color of the source surface or vice versa. This technique is used, for example, when blitting a video over some other surface. This technique can be used to mask, for instance, text, menus or other content on the top of the video.

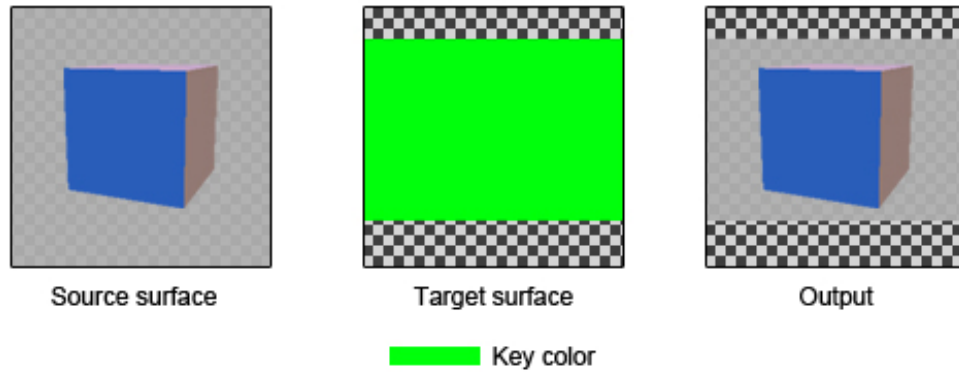


FIGURE 11. An example of destination color keying.

Masking is basically the same operation as color keying but the actual area is selected with an extra surface, which contains information on what pixels are selected to the output. To save memory, the mask is usually presented in the 1 BPP (one bit per pixel) format. An example of masking can be seen in the following figure (figure 12). In the example, the mask black color represents 1 and the white color represents 0 in the 1-bit per pixel format.

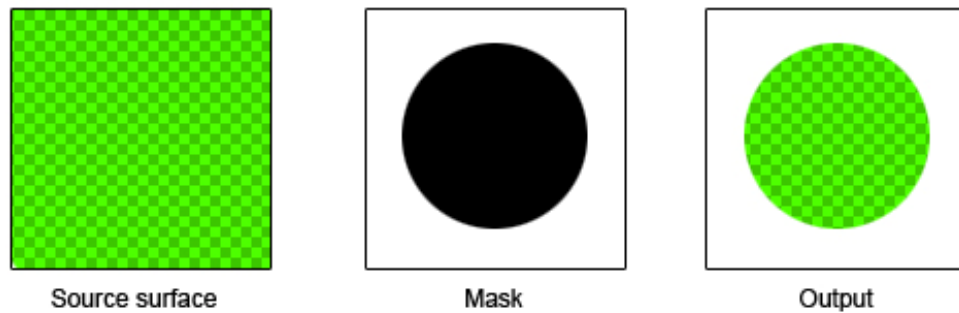


FIGURE 12. An example of masking

3 ARM 11 PROCESSOR FAMILY

The ARM11 core was licensed in 2002 by ARM Ltd. (27.). The ARM11 processor family architecture contains 32-bit RISC (reduced instruction set computing) microprocessors. These processors include numerous features, such as SIMD (single instruction, multiple data) operations and multiprocessor support. When using the SIMD operations in particular it is possible to speed up many use cases such as blitters. A total of four different processor types belong to the ARM11 processor family and they all utilize the ARMv6 processor architecture. These types are ARM1136, ARM1156, ARM1176 and ARM11 MPCore. The ARM11 family processors work from clock rate of the 350 MHz up to 1 GHz. All ARM11 processors utilize the ARMv6 instruction architecture (28; 29).

The most used ARM11 types are ARM1176 and ARM1136. They are used by many consumer electronics companies such as Nokia, Apple, Samsung, Nvidia etc. In this study the focus is on the ARM1176 processor because it is the hardware with which the actual work of this study will be tested and benchmarked.

ARM1176 supports an eight-stage pipeline and up to four simultaneously run instructions. An eight staged pipeline means that each processor instruction has to go through eight stages for the result. Not all of these stages are gone through for every instruction. The more complex the instruction, the more processor clock cycles there are and the more pipeline stages and time are needed for the process. By overlapping multiple stages, the processor can achieve the maximum clock rate to execute the instructions. Processor stages and stage overlapping are presented in the following table (table 2).

TABLE 2. The ARM 11 eight-staged pipeline

Time	1	2	3	4	5	6	7	8	9	10	11	12
Instr 1	Fetch	Fetch	Decode	Read	Shift	ALU	Saturate	Write				
Instr 2			Fetch	Fetch	Decode	Read	Shift	ALU	Saturate	Write		
Instr 3					Fetch	Fetch	Decode	Read	Shift	ALU	Saturate	Write

When an instruction is initiated in the processor, the first two stages are for fetching the instruction information from the memory and for carrying out branch prediction. Branch prediction means that the processor is trying to guess where the program is going to branch or jump next (30). Branch prediction can be used for achieving high performance boost if the prediction of the processor is correct. The performance comes from preloading the next instructions from the predicted address and fetching them to the processor pipeline. In case of misprediction, the processor must clear the pipeline of instructions and start processing from the actual branching address which can cause heavy penalty to the performance (31).

After fetching the instruction, the processor starts the decoding stage. This means that the processor is decoding the binary data including the instruction and preparing to issue it. The last part of the common pipeline is reading the register and issuing the instruction. The rest of the pipeline stages shown in the table (table 2) can change depending on the instruction. The three instructions presented in the table are typical ALU (arithmetic logic unit) instructions, such as bit shifting, adding or subtracting a register value. The processor uses different pipelines for multiplying and memory loading and storing instructions.

The ARM1176 processor has 13 general registers (registers 0-12), program counter register (PC), stack pointer register (SP), link register (LR) and one or two status registers when working in the ARM state. The registers are 32 bit long containers in the processor and they are used to load, modify and save data. The program counter register keeps track of the instruction running. The counter is always four instructions ahead of the actual processing because of the simultaneous running. The program counter can

be used for example in branch prediction. The stack pointer register holds the address in the memory where routines can save temporary data during the runtime. The link register or subroutine link register is used for saving the return address when the branch and link routine is used. Otherwise LR can be used as one of the general registers. Last but not least, the ARM1176 has one or two status registers in the ARM state. Status registers hold up flags of the instructions that want to update these status registers. These flags can be, for instance, results of comparison between two register values or processor state flags.

4 DYNAMIC ASSEMBLY

Dynamic assembly is a self-modifying code which means that the program is modifying or creating its own instruction stream while running. A self-modifying code is used to prevent reverse-engineering, and to optimize both memory usage and runtime of the program (32).

Assembly and software routines are usually written in the order of execution. In dynamic assembly, all routines are generated during the run-time depending on what kinds of operations the program should do. Though dynamic assembly is far more complicated than the traditional assembly, it offers a number of benefits for the system. Thanks to the dynamic code generation system, it is possible to use and compile only the code that is needed, which saves memory. In addition, recycling the code in such a way that a piece of code that has been generated once does not have to be generated again makes dynamic assembly useful in frequently repeated operations.

Blitters written in the dynamic assembly, as in binary code, are often much faster than normal blitters written in C- or C++-language. The difference comes mainly from the not optimized functions during the compilation and branches of if- and switch-case-sentences, which significantly affect the performance. The dynamic assembly or normal assembly can also be used to tweak the register usages and branch prediction when in C-language, for example, this is done by the compiler.

The dynamic assembly also allows using C variables for offsets and dynamic memory loading. In ARMv6, all ARM assembly routines are described as 32-bit binary codes, and all supported functions and binary codes can be found in the ARMv6 Reference Manual (29). It is also possible to use the THUMB or THUMB-2 encoding in the ARM11 processors that use 16-bit routines. Routines are generated on run-time, saved to the cache memory and run from there whenever the blitter is needed. An example of a dynamically generated assembly function is shown below (example 1).

EXAMPLE 1. Example of a generated assembly pipeline

```
#typedef uint32 unsigned long
#define REG_1 0x01
#define ADD_OP(pipeline,dst,value) pipeline = ((uint32)0xE1800000 | (dst << 16) |
(value << 8);
int main(void)
{
    uint32 i, pipeline[2];
    for(i = 0; i < 2; i++)
    {
        ADD_OP(pipeline[i],REG_1,i+1);
    }
    (void)*pipeline();
}
```

The example (example 1) of a dynamically generated pipeline in the ARM assembly language would be:

```
ADD r1,#1
ADD r1,#2
ADD r1,#3
```

The code would increase register 1 value with six. In an actual case, the memory for the pipeline would be dynamically allocated in order to increase the size flexibly when more routines are included in the pipeline. The component doing the dynamic allocation and creating the actual pipeline is called stitcher. Though the C-part of this code snippet is much longer in this simple use case, the actual implementation can be done using a much smaller number of code lines compared to writing the entire code in the traditional assembly language.

Dynamic assembly is not an answer to all problems. If the generation time is long compared to the actual execution time, the actual benefit of using the

dynamic assembly becomes meaningless. This means that very simple operations such as color conversion over target is not giving as good results as more complicated operations such as alpha blending.

5 SPLATTER

The project name for the dynamically generated assembly blitter is Splatter. The name comes from an English word which means "to splash and scatter upon impact" (33). The objective of the first phase when this study was commenced was to get the code generation environment running. The next phase was to create a test environment for Splatter. In the final part, the actual pixel pipeline generation was implemented and tests were run on the hardware. Moreover, a significant amount of performance tweaking was carried out for the blitter after the first implementation. Each of these phases are explained in more detail in the following chapters.

With two different implementations, one for the actual blitter and the other for the code generation, it is possible to re-use the code generation component for other frequently used operations in S40 architecture.

Before starting this project, other possibilities and previous implementations had to be studied. There are other blitters using the dynamic assembly, such as DrawElements' Blitrix (34). In addition, open source blitters utilizing a dynamically generated assembly exist, the currently discontinued The Core Pocket Media Player (TCPMP) project serving as an example (35). Neither of these products could be used in this study because open source products cannot be utilized for commercial use, in addition to which Blitrix only supports processors based on the ARM NEON architecture.

Furthermore, a reference blitter was written for Splatter. The reference blitter was written in C and it was used to check the output of the actual dynamic assembly blitter in the test environment. The reference blitter is also used in a phone emulation environment run on a PC where ARM assembly instructions do not work because of a different processor architecture (x86) and because possible performance loss is not so crucial.

5.1 Stitcher and code generation component

A new component was created for code generation and stitching. The Component was named ARM-Ray, which stands for fast ARM assembly generation. ARM-Ray was developed using the ANSI C language. The pipeline in this context should not be confused with the actual processor instruction pipeline and it should be understood as a generated binary pipeline.

Because ARM 11 processors only have 13 normal registers, pipelines are allocated with extra 13 soft registers. Soft registers are 4-byte long memory allocations where the user can save information for easy loading to the actual hardware registers. These soft registers are allocated just before the actual runnable code part of the pipeline and they are always replaced with new data if the pipeline is re-used later.

ARM-Ray uses a linked list to keep track of the generated pipelines. Pipelines are identified with a 32-bit hash, which is generated for Splatter depending on the source and destination formats, scaling quality, source and destination blend modes and the rotation of the blit. This hash is always created by the client of ARM-Ray. The hash value can be used to find pipelines that have already been generated and re-use them. This saves time if the same pipelines are used frequently.

The maximum internal cache size, the first and the last element of the linked list as well as the number of generated pipelines are stored in the pipeline holder structure. This structure has to be created by the client before any pipelines can be created. The structure is used for keeping track of the memory usage and the number of pipelines, in addition to which it also ensures that the pipelines of different clients using ARM-Ray are not mixed. At the moment, the only client for ARM-Ray is Splatter but it can be used in various other purposes in the future.

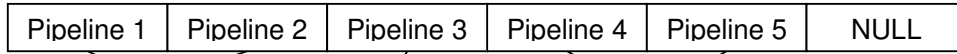
After the pipeline has been generated to the heap memory, it is moved to the Splatter's internal cache memory for running and the cache is invalidated. Invalidation ensures that the data in the internal cache memory is runnable and it is not corrupted.

Each pipeline structure contains a pointer indicating the start of the generated binary code of the pipeline in the cache. The structure also contains pointers to next and previous pipelines which are used when ordering pipelines in the pipeline container. When a pipeline is run, it is moved to be the head of the list because it is most likely to be used again in the near future. Consequently the least recently used pipeline is in the last place in the list.

ARM-Ray limits the memory size and the number of pipelines that can be saved to the pipeline holder. When either of these limits is reached, ARM-Ray deletes the least frequently used pipeline, i.e. the last pipeline in the list, from the memory and invalidates the cache again. Figure (figure 13) presents the pipeline operations in the stack and cache memory.

1. Starting point with five pipelines generated to the memory

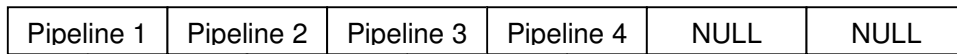
Pipeline structures in the heap memory



The generated binary code in the internal cache memory

2. Deleting the least frequently used pipeline (Pipeline 5)

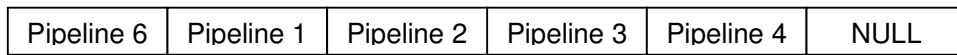
Pipeline structures in the heap memory



The generated binary code in the internal cache memory

3. Creating a new pipeline (Pipeline 6)

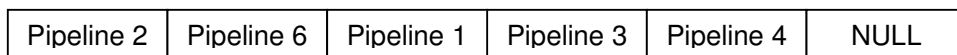
Pipeline structures in the heap memory



The generated binary code in the internal cache memory

4. Re-using and running a pipeline (Pipeline 2)

Pipeline structures in the heap memory



The generated binary code in the internal cache memory

FIGURE 13. ARM-Ray pipeline operations in memory

When a generated pipeline is re-used, soft registers must be transferred at the beginning of the pipeline in the internal cache in the way that the memory addresses and other data needed for the blit are correct.

5.2 Test environment

The test environment was created after the basic functionalities of Splatter and ARM-Ray were finished. RealView Debugger v4.0 was used for debugging. The test environment was written in Perl, and it separately compiles all written test cases and runs them through the debugger on the command line. The test environment does not give indications of the real speed of the blitter since it is run on a computer with a Linux operating system. However, it is possible to verify correct blit results using the output images of the test environment.

Getting an output image of the target surface at the end of the operation was essential for the test environment. This was done by creating a breakpoint in the part of the code where the blit had been made. When the test program hits the breakpoint, the script copies the target surface from the memory and saves it to a binary file. The breakpoint and the script to run test cases were created with the RealView debugger CLI scripting language.

The binary file had to be converted to the actual image data for previewing the results. This was done with the internal tool of the S40 UI Engine, and it saved a significant amount of work during the actual implementation of the blitter.

A total of 50 different test cases were written to Splatter. These test cases included the most commonly used blits in S40 GUI. The runtime of the test cases was relatively long since all test cases had to be run in the debugging environment separately.

5.3 Pixel pipeline generation

The actual pixel pipeline was generated in Splatter, which was also written in ANSI C. Since there are many supported canvas types (table 3.), rotation and mirroring, scaling, clipping and actual blitting, the pipeline generation was very difficult to accomplish in parts of the implementation.

To support so many features, the blitter must get the blitting information from the client. Splatter, in general, requires five different pieces of information from the client to carry out the blitting. This information is given to the blitter as function parameters. The information includes the structure of source and destination surfaces, source and target rectangles and blitting parameters. The surface structures contain data such as surface type, width, height and pointer to the memory allocation of the surface. Rectangles are used for clipping, and they contain coordinates of the source and destination clipping rectangles. The blitting parameter structure contains the alpha mode, blend types for source and destination, flagging to set the global alpha and mirroring, scaling quality, rotation degrees and rotation origo coordinates. Having this information, it is possible to create or re-use a previously generated pixel pipeline required by the client. When a new pipeline is generated, Splatter operates in accordance with the following pattern (see figure 14) to produce the binary output.

All of the steps in the figure are not needed in all blits. For example if the blit does not need to do the alpha blending, it does not need to read the destination pixel or do the actual alpha blending calculations. Similarly, if the source and destination surfaces are in the same pixel format, no color conversion is needed. The only mandatory steps in the pipeline are source pixel reading and writing to the destination.

It is possible to speed up the blitting by eliminating most of the steps from the pipeline. As explained earlier, the re-use of the pipeline is a crucial benefit for dynamically generated blitters.

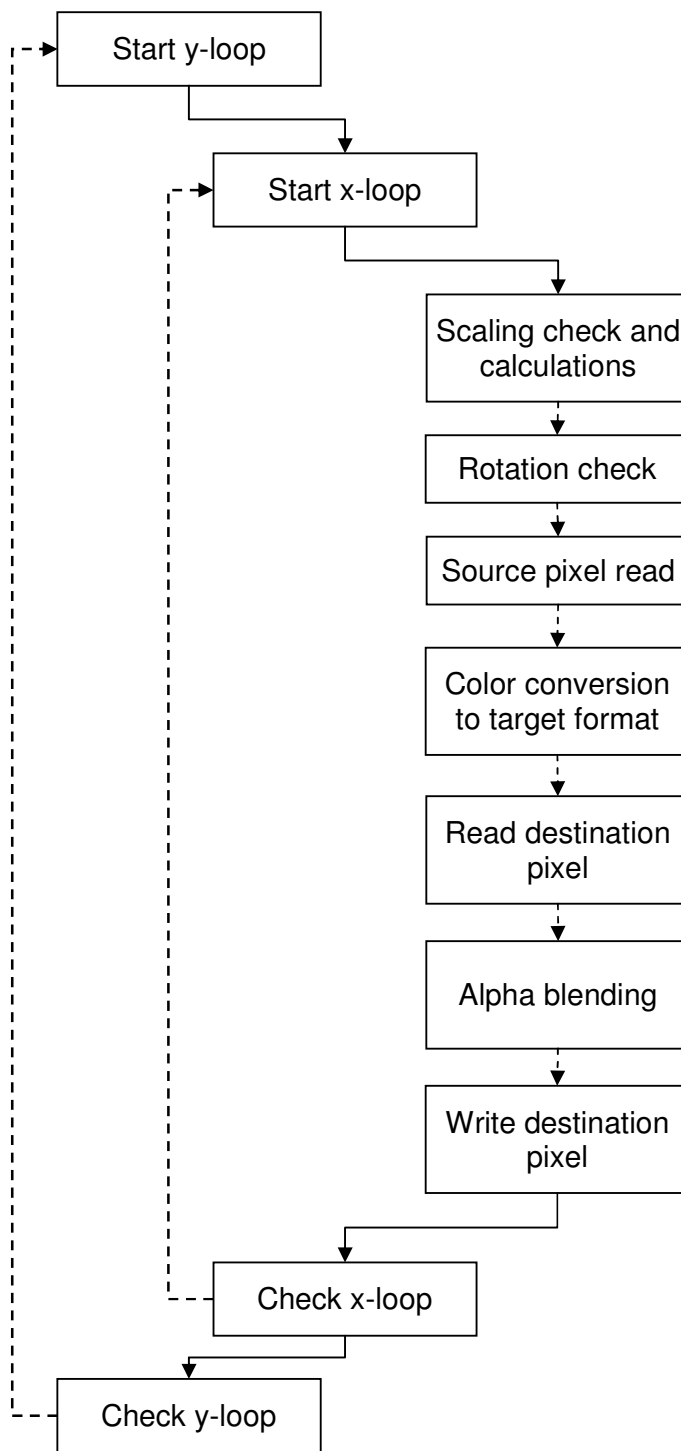


FIGURE 14. An example of a pixel pipeline.

5.4 Supported formats and blit types

Splatter supports all surface types that are the most frequently used in S40 GUI. Each of the surface types can be used as a source and as a destination surface except the alpha formats (4-bit alpha and 8-bit alpha), which can only be used as a source format. Splatter generates the pipeline to convert the source pixel format to the pixel format of the target if needed. The supported types and their memory alignment in a big-endian system are presented in the table below (table 3).

TABLE 3. The supported surface types and memory alignment

32-bit XRGB	0x0000					0x0004				
Dword	??	RED	GREEN	BLUE	??	RED	GREEN	BLUE		
Bits	8	8	8	8	8	8	8	8	8	
32-bit ARGB	0x0000					0x0004				
Dword	ALPHA	RED	GREEN	BLUE	ALPHA	RED	GREEN	BLUE		
Bits	8	8	8	8	8	8	8	8	8	
32-bit ARGB PRE	0x0000					0x0004				
Dword	ALPHA	RED	GREEN	BLUE	ALPHA	RED	GREEN	BLUE		
Bits	8	8	8	8	8	8	8	8	8	
16-bit XRGB	0x0000					0x0002				
Word	??	RED	GREEN	BLUE	??	RED	GREEN	BLUE		
Bits	4	4	4	4	4	4	4	4	4	
16-bit ARGB	0x0000					0x0002				
Word	ALPHA	RED	GREEN	BLUE	ALPHA	RED	GREEN	BLUE		
Bits	4	4	4	4	4	4	4	4	4	

16-bit ARGB PRE	0x0000			0x0002				
Word	RED	GREEN	BLUE	RED	GREEN	BLUE		
Bits	5	6	5	5	6	5		
4-bit alpha	0x0000				0x0002			
Word	ALPHA	ALPHA	ALPHA	ALPHA	ALPHA	ALPHA	ALPHA	
Bits	4	4	4	4	4	4	4	
8-bit alpha	0x0000				0x0004			
Dword	ALPHA	ALPHA	ALPHA	ALPHA	ALPHA	ALPHA	ALPHA	
Bits	8	8	8	8	8	8	8	
YUV 4:2:0	0x0000				0x0004			
Dword	Y0	Y1	Y2	Y3	U0	U1	V0	V1
Bits	8	8	8	8	8	8	8	8
YUV 4:2:2	0x0000				0x0004			
Dword	Y0	U0	Y1	V0	Y2	U1	Y3	V1
Bits	8	8	8	8	8	8	8	8

XRGB types (both 16-bit and 32-bit) and the 16-bit 565 are fully opaque surface types. ARGB surfaces include an alpha channel and are therefore run through an alpha blending algorithm. PRE types (both 32-bit and 16-bit ARGB) present pre-multiplied alpha formats. Alpha types (4-bit and 8-bit alpha) are surfaces that only include an alpha component and the actual color data is given in the blitting parameter structure.

5.4.1 Blit types

Blit types define how the source surface is being transferred to the target. For alpha blending, Splatter uses the commonly known alpha blending formula (see formula 1). The component supports the same source and

destination factors as, for example, OpenGL. Splatter supports a total of eight different alpha blending factors and values that are presented in the table (table 4.).

TABLE 4. The supported blending factors for alpha blending

Factor	Value
SPLATTER_ZERO	0
SPLATTER_ONE	1
SPLATTER_SRC_ALPHA	α_a
SPLATTER_ONE_MINUS_SRC_ALPHA	$1 - \alpha_a$
SPLATTER_DST_ALPHA	α_b
SPLATTER_ONE_MINUS_DST_ALPHA	$1 - \alpha_b$
SPLATTER_SRC_COLOR	C_a
SPLATTER_ONE_MINUS_SRC_COLOR	$1 - C_a$
SPLATTER_DST_COLOR	C_b
SPLATTER_ONE_MINUS_DST_COLOR	$1 - C_b$

The type defines the blending factor for both the source (F_a) and the destination (F_b), which is used to multiply the color components in the alpha blending (see formula 1). Splatter also supports three types of alpha modes. In the ALPHA_DISABLE mode, the source alpha is filled with a full alpha if applicable and then written to the destination. In the ALPHA_DIRECT mode, the source is written directly to the destination with no blending. The third

alpha mode sets the blitter to use the defined blend modes (see table 4) Changing these values significantly affects the resulting image.

5.4.2 Transformations

Splatter implementation supports two types of scaling. Scaling quality is defined in blit parameters sent to the blit function. The quality can be low, medium or high, and it is used for selecting the scaling algorithm for the blit as seen in the table (table 5).

TABLE 5. Scaling quality factors

Factor	Algorithm
SCALE_QUAL_NONE	No scaling
SCALE_QUAL_LOW	Nearest neighbor
SCALE_QUAL_MEDIUM	Nearest neighbor or bilinear
SCALE_QUAL_HIGH	Bilinear

Bicubic scaling was not included in the first implementation of the blitter. Bicubic scaling requires a great number of registers to work with and is relatively complicated to implement with ARM 11, which only supports 13 general registers. In addition, the performance of an bicubic scaling that uses a numerous memory reads and writes would be as poor as in the blitter written in plain C.

The most challenging part of the scaling was to get the calculations to function properly without using divisions. Since most of the low-end ARM11 processors do not have a separate floating point unit, fixed-point calculations were used to make the fraction numbers to function.

Splatter supports the clockwise fixed degree rotation. The Supported rotation values are 90, 180 and 270 degrees. Moreover, the rotation origo must be

taken into account when carrying out the rotations. The rotation origo is the point around which the source surface is rotated. The origo is defaulted to the upper left corner of the source surface but it can also be set by the client. The rotation is maybe the least challenging of the implementation process because it can be carried out simply by reading the source in the other direction.

The implemented blitter also supports the destination color keying and the global alpha modes. Masking was not carried out in this study because it has very few use cases in the current S40 UI.

5.5 Performance tweaking

After all the features of the blitter were implemented, a great amount of performance tweaking was carried out. Performance tweaking allows gaining even better results and improving the speed of the blitter. Most of the performance tweaking was carried out by trying to find a way to simultaneously run as many operations as possible. This means that some operations, such as multiplying two registers, take multiple cycles to output the result. After fetching the instruction, it is possible to start running another operation such as adding two other registers together while the multiplying operation is still ongoing in the pipeline of the processor.

Thanks to ARM11 parallel instructions, it was possible to perform multiple operations with very few processor cycles. For example, the SMUAD instruction, which multiplies the low and high bits of two registers and adds them together, takes only one cycle to run whereas normal multiplying takes three cycles. It was possible to easily outperform blitters written in C-language in speed by using these operations.

Furthermore, the branch prediction tweaking slightly allowed the speeding up the operations. Especially in the inner loop, it was essential to check the end of the line in the pipeline as soon as possible for the processor to be able to predict the following instructions.

6 RESULTS AND CONCLUSIONS

The results of the hard work were benchmarked using the internal benchmarking code of S40 UI Engine. The benchmarking code runs through all source formats and destination formats and executes different types of blits between them. The actual results of the benchmarking are then presented as Mpix/s and as time values. Mpix/s indicates how many megapixels (millions of pixels) the blitter can process in one second and the time value indicates how long it took from the processor to run the required blit. In this benchmarking process, plain cases (table 6) were run with no alpha blending, and the copy cases used the source over the destination alpha blending. All plain cases were run 240 times and the alpha blending cases were run 120 times. All operations were carried out from a full screen source surface to a full screen target surface. All of the following results were run several times because the results can slightly differ between the benchmarks.

All benchmarks were run against a blitter written in plain C currently used in S40 mobile phones. The benchmarking hardware was a mobile phone prototype with an ARM11 processor running at 900 MHz clock speed.

TABLE 6. Benchmarking results

CASE	C blitter Mpix/s	SPLATTER Mpix/s	Percentage
H32_A -> H32APRE PLAIN	192.00	196.09	102.13%
H32_A -> H32APRE COPY	10.04	9.66	96.22%
H32_A -> H32_A PLAIN	192.00	196.09	102.13%
H32_A -> H32_A COPY	4.09	9.72	237.65%
H32_A -> H16_X PLAIN	49.02	51.49	105.04%
H32_A -> H16_X COPY	19.78	19.28	97.47%
H32_A -> H32_X PLAIN	92.16	196.09	212.77%
H32_A -> H32_X COPY	9.60	14.13	147.19%
H32_A -> H16_A PLAIN	19.65	47.02	239.29%
H32_A -> H16_A COPY	5.50	11.58	210.55%
H16_X -> H32APRE PLAIN	24.19	49.02	202.65%
H16_X -> H32APRE COPY	24.38	48.51	198.97%
H16_X -> H32_A PLAIN	22.59	49.02	217.00%
H16_X -> H32_A COPY	22.59	49.55	219.34%

H16_X -> H16_X PLAIN	214.33	368.64	172.00%
H16_X -> H32_X PLAIN	52.36	52.07	99.45%
H16_X -> H16_A PLAIN	44.96	88.62	197.11%
H16_X -> H16_A COPY	45.18	88.62	196.15%
H32_X -> H32APRE PLAIN	91.25	90.35	99.01%
H32_X -> H32APRE COPY	94.04	88.62	94.24%
H32_X -> H32_A PLAIN	92.16	89.48	97.09%
H32_X -> H32_A COPY	92.16	85.33	92.59%
H32_X -> H16_X PLAIN	52.97	57.24	108.06%
H32_X -> H16_A PLAIN	52.97	53.89	101.74%
H32_X -> H16_A COPY	53.38	54.86	102.77%
H16_565 -> H32APRE PLAIN	40.07	42.67	106.49%
H16_565 -> H32APRE COPY	40.07	43.47	108.49%
H16_565 -> H32_A PLAIN	39.90	42.67	106.94%
H16_565 -> H32_A COPY	39.72	43.07	108.43%
H16_565-> H16_X PLAIN	58.70	60.63	103.29%
H16_565 -> H32_X PLAIN	39.90	44.52	111.58%
H16_565 -> H16_A PLAIN	45.40	61.03	134.43%
H16_565 -> H16_A COPY	45.18	61.44	135.99%
H16_A -> H32APRE PLAIN	26.18	40.96	156.46%
H16_A -> H32APRE COPY	14.13	10.84	76.72%
H16_A -> H32_A PLAIN	24.25	40.78	168.16%
H16_A -> H32_A COPY	4,82	10.82	224.48%
H16_A -> H16_X PLAIN	98.04	368.64	376.01%
H16_A -> H16_X COPY	20.39	23.27	114.12%
H16_A -> H32_X PLAIN	50.09	42.47	84.79%
H16_A -> H32_X COPY	12.84	16.70	130.06%
H16_A -> H16_A PLAIN	368.64	368.64	100.00%
H16_A -> H16_A COPY	5.82	12.91	221.82%
H16_A -> H16_565 PLAIN	47.75	40.78	85.40%
H16_A -> H16_565 COPY	18.58	19.44	104.63%
H32_A -> H16_565 PLAIN	55.19	46.31	83.91%
H32_A -> H16_565 COPY	20.76	19.69	94.85%
H32_X -> H16_565 PLAIN	52.97	51.78	97.75%
H16_X -> H16_565 PLAIN	47.75	44.52	93.24%
H16_565 -> H16_565 PLAIN	368.64	384.00	104.17%
H4_A -> H32_A PLAIN	5.11	54.21	1060.86%
H4_A -> H32_A COPY	5.09	11.29	221.81%

The results show that in most cases the process was much faster when using Splatter. The average percentage of these values is 161.10%, which is rather good compared to the optimization level of the code. The benchmarking indicates that most problems still lie in the pre-multiplied alpha surfaces (H32APRE) and in the 16-bit 565 RGB surfaces (H16_565). These

cases still need some optimization. The most used blits in S40 GUI, such as the 32-bit ARGB source over 32-bit ARGB destination surface alpha blend operation, are almost 2.5 times faster than the C-implemented blitter.

At the moment, no transformations are used in the benchmarking code. Accordingly, all of the supported surface types are not used either, but it nevertheless gives an idea of Splatter's speed and the speed of dynamically generated blitters in general. These problems will be taken into account in the future work of Splatter.

In conclusion, the final product was a success, and it seems to offer a promising new way of handling two dimensional graphics with software in S40. At the moment, no component in S40 architecture utilizes the final product of this study. However, the final product will be first implemented in the common S40 graphics drawing and later in the composition engine. In addition, the interface of Splatter is not currently open to all S40 clients.

7 DISCUSSION

The objective of this study was to design, implement and test a fast dynamically generated assembly blitter for S40 mobile phones with ARM11 processors. The names of the final products were ARM-Ray and Splatter, which are two separate software components. The high number of various transformations, surface types and pixel pipeline management caused problems related to the size of the software as well as its maintenance and development. All objectives set for this study were reached.

This study was highly challenging, which was known from the start. However, the design and implementation proceeded really well and the results of the work were better than first expected. Thanks to successful implementation and performance tweaking, it was possible to significantly speed up almost all blit operations. Problems still exist in the dynamic blitter, in particular with small blits. The high number of parameter checks and memory operations prior to the actual run of the pixel pipeline slow down the operations carried out to small surfaces.

Four different programming languages were used in the making of this study, namely ANSI C, ARM Assembly, Perl and Realview debugger scripting language. As the author of this study had no previous experience of most of these languages made the work interesting and challenging.

Further development of this work is already being planned. ARM Cortex and ARM9 supports will be implemented later. Bicubic scaling will also be done on ARM Cortex processors because they support vertex operations. Masking will also be implemented later if necessary. Moreover, a great amount of performance tweaking will be done especially for the cases needed most, in addition to which benchmarking must to be carried out against other blitters than those used in S40 development. The benchmarking code itself will be further developed to fully support all surface types and transformations.

LIST OF REFERENCES

1. Computer graphics. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Computer_graphics. Date of data acquisition: 3 January 2011.
2. Vector graphics. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Vector_graphics Date of data acquisition: 20 October 2010.
3. Google Maps. Available: <http://maps.google.com>. Date of data acquisition: 20 October 2010.
4. Foley, James D. - van Dam, Andries - Feiner, Steven K. - Hughes, John F. 1997. Computer Graphics, Principles and practice. Addison-Wesley. Pages 1-2.
5. Raster graphics. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Raster_graphics. Date of data acquisition: 20 October 2010.
6. Khronos Group homepage, OpenGL ES overview. Available: <http://www.khronos.org/opengles/>. Date of data acquisition: 19 October 2010.
7. Microsoft Developer Network, GDI+, Available: <http://msdn.microsoft.com/en-us/library/ms533798%28v=vs.85%29.aspx>. Date of data acquisition: 25 October 2010.
8. Microsoft Developer Network, DirectDraw, Available: <http://msdn.microsoft.com/en-us/library/ms879875.aspx>. Date of data acquisition: 25 October 2010.

9. Qt homepage, a cross-platform application and UI framework. Available: <http://qt.nokia.com/>. Date of data acquisition: 25 October 2010.
10. Khronos Group homepage, OpenVG overview. Available: <http://www.khronos.org/opencv/>. Date of data acquisition: 25 October 2010.
11. OpenGL homepage. Available: <http://www.opengl.org/>. Date of data acquisition: 25 October 2010.
12. Games for Windows marketplace, learn about DirectX. Available: <http://www.gamesforwindows.com/en-US/directx/>. Date of data acquisition: 25 October 2010.
13. Blitter. The free encyclopedia. Available: <http://en.wikipedia.org/wiki/Blitter>. Date of data acquisition: 2 November 2010.
14. Graphics processing unit. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit. Date of data acquisition: 21 September 2010.
15. Simple DirectMedia Layer homepage. Available: <http://www.libsdl.org/>. Date of data acquisition: 2 November 2010.
16. BitBLT. The free encyclopedia. Available: <http://en.wikipedia.org/wiki/Bitblt>. Date of data acquisition: 2 November 2010.
17. Foley, James D. - van Dam, Andries - Feiner, Steven K. - Hughes, John F. 1997. Computer Graphics, Principles and practice. Addison-Wesley. Pages 12-13.
18. YUV. The free encyclopedia. Available: <http://en.wikipedia.org/wiki/YUV>. Date of data acquisition: 3 November 2010.

19. Foley, James D. - van Dam, Andries - Feiner, Steven K. - Hughes, John F. 1997. Computer Graphics, Principles and practice. Addison-Wesley. Page 206.
20. Foley, James D. - van Dam, Andries - Feiner, Steven K. - Hughes, John F. 1997. Computer Graphics, Principles and practice. Addison-Wesley. Pages 924-944.
21. Foley, James D. - van Dam, Andries - Feiner, Steven K. - Hughes, John F. 1997. Computer Graphics, Principles and practice. Addison-Wesley. Page 55.
22. Bilinear interpolation. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Bilinear_interpolation. Date of data acquisition: 3 December 2010.
23. Bicubic interpolation. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Bicubic_interpolation. Date of data acquisition: 3 December 2010.
24. Alpha compositing. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Alpha_compositing. Date of data acquisition: 13 October 2010.
25. Porter, Thomas - Duff Tom. 1984. Compositing Digital Images. Proceedings of SIGGRAPH'84, Computer Graphics, Vol. 18, No.3, pages 253-259.
26. Foley, James D. - van Dam, Andries - Feiner, Steven K. - Hughes, John F. 1997. Computer Graphics, Principles and practice. Addison-Wesley. Page 838.
27. ARM architecture. The free encyclopedia. Available: http://en.wikipedia.org/wiki/ARM_architecture. Date of data acquisition: 12 December 2010.

28. ARM Ltd. homepage. ARM 11 processor family. Available: <http://www.arm.com/products/processors/classic/arm11/index.php>. Date of data acquisition: 20 December 2010.
29. ARM Infocenter. ARM11 processors. Available: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.arm11/index.html>. Date of data acquisition: 19 December 2010.
30. Branch predictor. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Branch_predictor. Date of data acquisition: 3 December 2010.
31. Hennesy, John L. - Patterson, David A. 2003. Computer Architecture, A Quantitative Approach, 3rd edition. Pages 196-199.
32. Self-modifying code. The free encyclopedia. Available: http://en.wikipedia.org/wiki/Self-modifying_code. Date of data acquisition: 10 December 2010.
33. Dictionary.com. Online dictionary. Available: <http://dictionary.reference.com/browse/splatter>. Date of data acquisition: 8 September 2010.
34. Draw Elements Ltd. homepage. Blitrix. Available: <http://www.drawelements.com/blitrix.html>. Date of data acquisition: 20 September 2010.
35. The Core Pocket Media Player. Available: <http://picard.exceed.hu/tcpmp/>. Date of data acquisition: 11 October 2010.

APPENDICES

App 1. Cover page

App 2. Title page

App 3. Abstract