

The GapiDraw Platform: High-Performance Cross-Platform Graphics on Mobile Devices

Johan Sanneblad and Lars Erik Holmquist
Future Applications Lab, Viktoria Institute
Horselgangen 4, SE-41756 Goteborg, SWEDEN
www.viktoria.se/fal
{johans, leh}@viktoria.se

ABSTRACT

The *GapiDraw* platform supports the creation of high-performance graphical applications across a variety of handheld hardware configurations, including Palm, Symbian and Windows Mobile devices. Handheld computers makes it possible to create applications and services not possible with stationary computers, thus there is a need for a high performance development platform for rapid prototyping on mobile devices. Unlike desktop computers there has not yet evolved a single standard for graphics on handheld devices. Typically, handheld computers only provide direct frame buffer access, and there are major differences in implementation details across different hardware configurations, making it difficult to use mobile devices for prototyping. Using *GapiDraw*, developers can re-use the same code across a variety of devices and do not have to focus on device-specific implementation details. *GapiDraw* is actively used as an enabler platform in numerous research labs, and has also been used in over one hundred commercial games. We give an overview of the platform, and highlight some new mobile application concepts made possible through the use of *GapiDraw*.

Keywords

Handheld computers, mobile phones, mobile games, graphics framework, graphics API, graphics middleware, prototyping

1. INTRODUCTION

Handheld computers are rapidly gaining in popularity – and the capabilities of such devices are increasing at an accelerated pace. From Apple's *Newton* and Palm's *Pilot*, with their monochrome screens and processor speed in the single-digit megahertz range, current handhelds outstrip the performance of a stationary computer only a few years old. A typical PocketPC such as the *Toshiba e805* has a screen with 65.535 colors and a resolution of 480 x 640 pixels, a processor running at 400 megahertz, and 128 megabytes of RAM. This evolution means that handheld computers will see a shift from low-resolution grayscale graphics and text-based interfaces, to interactive applications that take full advantage of these new possibilities – including high-resolution, full-color graphics and full-motion

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MUM 2004, October 27-29, 2004 College Park, Maryland, USA.
Copyright 2004 ACM 1-58113-981-0 /04/10... \$5.00



Figure 1: *FirePower - Onrush* by ManaiSoft is one notable game that uses the *GapiDraw* platform.

video.

Unfortunately, while for stationary PCs there has been a convergence in graphics hardware and software towards a common developer platform (the Microsoft DirectX platform [3]), this has not yet been the case for handheld computers. Because of the heterogeneous hardware specifications of mobile devices, developers who want to create high-performance graphic applications on one handheld computer have to create new optimized graphics routines for each new device they want to target. Important hardware variations include frame buffer orientations, cache memory sizes, memory access latencies and timer granularities. Very few handheld computers currently offer graphics acceleration in hardware, which means that the speed of the graphics routines is crucial in the performance of software applications.

We believe that this lack of a common development standard is seriously hindering the creative potential of mobile devices. As a solution, we introduce *GapiDraw*, a freely usable graphics platform for handheld computers. Originally developed specifically for games programming (hence the name, which stands for Game Application Programming Interface for Drawing), *GapiDraw* supports all applications that require high performance graphics. It borrows many features from DirectX, easing the transition for programmers already familiar with games programming on stationary PCs. *GapiDraw* supports development over a wide range of handheld devices, including Palm handheld computers, Symbian mobile phones and all Windows Mobile devices (Pocket PCs and Smartphones). For development and testing purposes it also runs natively on stationary PCs. *GapiDraw* was first used in education and is supported by a lively developer community and an active online forum. *GapiDraw* is used

extensively in commercial applications, making it something of a current de-facto standard for mobile application development. One of the more notable releases so far is *FirePower - Onrush* by ManaiSoft (see Figure 1).

2. HANDHELD COMPUTER GRAPHICS

The sheer number of different mobile devices and mobile operating systems has made it quite difficult to create mobile applications that run on more than one device. In fact, the uniqueness of the various operating systems has made it difficult to create applications for just one device, since mobile application development in many cases is very different from developing applications for stationary computers. Thus, there have been several software platforms released to assist with application development for mobile devices. We will now highlight some of the platforms for creating high-performance graphic applications on mobile devices, and then introduce our GapiDraw platform and relate it to these platforms.

2.1 Graphic Hardware APIs

With *Graphic Hardware APIs* we mean Application Programming Interfaces that expose the video hardware of a device, with much of the functionality available only if the appropriate video hardware is available. *OpenGL ES* is such an API and is an embedded subset of the industry-standard OpenGL graphics platform [11]. OpenGL ES has been designed for today's mobile microprocessors (e.g. replacing floating point arithmetic with fixed point), while still providing a feature set matching that of stationary PCs just a few years ago. The numerous differences to the stationary OpenGL API however may make it difficult to create cross-platform OpenGL / OpenGL ES applications using OpenGL ES (the lack of a GLUT [11] for OpenGL ES is one such thing), and it is today not clear what devices will ship with OpenGL ES compatible hardware in the near future. Devices that will not support OpenGL ES are Microsoft Windows Mobile devices, and Microsoft ships a product called *Direct3D mobile* [3] with their operating system Windows CE 5.0. Direct3D Mobile has a similar API to DirectX 8.0, but lacks most of the more advanced features used in many game titles for stationary PCs, and also has several API changes such as replacing floating point arithmetic with fixed point. When creating applications for mobile devices that take advantage of graphics hardware through APIs such as these, developers have to adapt their applications to the feature sets of each graphics library individually for each device they want to support.

2.2 Java-Based Graphic APIs

Java-enabled mobile phones are many, and so is the number of *Java-Based Graphic APIs*. Java on mobile devices (e.g. Java2 Micro Edition, J2ME) is in most cases a small subset of Java on stationary PCs, with performance most suitable for simple visualizations and puzzle games. To improve visual performance on Java for mobile devices, some mobile phone manufacturers ship native rendering libraries with Java APIs with their devices. Examples are *Mophun* by Synergenix, *Brew* by Qualcomm and *JBlend* by Aplix. Typically, Java-Based Graphic APIs such as Brew or JBlend only work on mobile phones, making it difficult to target other devices such as handheld computers using the same code base. Being designed for a small footprint, these graphic

libraries provide a minimum of functionality, making them difficult to use for prototyping.

2.3 Frame-Buffer APIs

Many handheld computers and Smartphones do not ship with a graphics API suitable for high-performance interactive applications such as games. Examples of such devices are Palm, Symbian and Windows Mobile devices. Thus there has evolved a market for *Frame-Buffer APIs*, which provide software-based rendering operations that draw graphics directly into the frame buffer of the device, using only the CPU as a graphics engine. Currently there are a few such platform-specific graphic libraries available, for example PocketFrog [4] on Windows Mobile, and Razor [12] on the Palm platform. One library worth mentioning is PocketHAL [4], which provides a unified way to access the frame buffer on Windows Mobile and Symbian devices, but does not by itself provide any drawing tools.

2.4 GapiDraw

The GapiDraw API is a mix between a Frame-Buffer API and a Graphic Hardware API. Most graphics operations provided by GapiDraw use graphics hardware if available (through DirectX or other APIs such as TwGfx on the TapWave Zodiac device), but every graphic operation is also implemented in software using techniques such as template meta programming and platform-specific assembler code. GapiDraw differs from the previously mentioned APIs in three ways. Firstly, GapiDraw supports mobile application prototyping through a high level framework that abstracts device-specific issues such as event handling, interfacing with the operating system, stylus control, image handling and file management. By using the GapiDraw framework, developers only have to consider the actual logic of their applications, without dealing with device-specific tasks. Secondly, GapiDraw was initially designed for handheld computers, and then later adapted to work on stationary PCs. There is no "downscaling" involved as with OpenGL ES or Direct3D Mobile. Thirdly, GapiDraw is currently the only graphic API that works across all mobile devices using the Palm, Symbian or the Windows Mobile operating systems.

3. DESIGNING A MOBILE GRAPHICS PLATFORM

GapiDraw was designed from the beginning with mobile devices as the prime target, and thus the implementation of GapiDraw is strongly influenced by current mobile hardware. During the design process, companies such as PalmOne, Microsoft and TapWave assisted with both device hardware and implementation details to aid with development. Below we highlight some of the design challenges that had to be considered when creating GapiDraw.

3.1 Frame Buffer Color Depths

Optimizing the performance of direct frame-buffer access has been a regularly visited research topic ever since the first raster-scan displays became publicly available almost three decades ago (e.g. the SUN display [1]). Different approaches for alternative frame buffer configurations (such as the 8 by 8 display [15]) were introduced for performance improvements, but the format that remained was the linear frame buffer format that was introduced with the first displays.

Typical mobile display depths are 12-bit (4096 color) or 16-bit (65535 color). To draw images to the display, developers have to manually convert RGB color values into the current native screen format and write them as bytes to the frame buffer of the device. If the display uses a 16-bit color depth, each pixel occupies 2 bytes. This representation is shown in Figure 2 (in the figure, the Palm OS 5 display is stored in Little Endian format where the rest of the operating system uses Big Endian, which explains why it might look odd). To convert an RGB value to native screen format, the processor has to perform up to ten operations for each pixel (four AND, three SHIFT and three OR for the Palm device). Since this is a costly operation, images should be pre-rendered to match the native format of the frame buffer before being used in any graphics operations. One difference between current mobile devices and stationary PCs with regards to pixel formats is that the color depth of the frame-buffer cannot be changed, where stationary PCs can re-initialize the frame-buffer to use any color depth in any resolution. Thus, supporting multiple pixel formats on mobile devices also means that several graphics routines have to be created, one for each display type, for optimal performance. One way to achieve this is to use *template meta programming*, which is described later.

3.2 Frame Buffer Orientations

Implementing high performance graphics on mobile computers, developers need to reconsider hardware aspects that were a common research topic almost three decades ago on stationary PCs. One such optimization is cache optimizations when reading and writing pixel data to the display frame buffer. For design reasons, displays are internally aligned differently on various mobile devices. Form factor is important, and the display and its connector are often rotated 90 or 180 degrees to decrease the physical size of the device. Currently there are mobile devices available with all of the four possible frame buffer orientations. Two such examples are seen in Figure 3. In the figure, the same four colors are displayed in the top left corner of the display. Depending on how the display is physically oriented in the device, the location in memory of the colors varies between devices.

The variation in frame buffer orientation is one of the more difficult issues to target when creating high-performance graphics for mobile devices. On stationary PCs, displays always present themselves to the developer in one format on all computers (where *xPitch*, the number of bytes to add to step to the next pixel, is always the size of one pixel, and *yPitch*, the number of bytes to add to step to the next row, is the 32-bit aligned width). If the display is internally stored in another format, video hardware will transform the display pixels to this format with a minimum of overhead (called a *swizzle*). The advantage of using a single display format is related to how memory is accessed using a *data cache*. Reading a single byte from any location in memory will automatically cause a computer to read an additional number of consecutive bytes (a *cache line*) and place that in the data cache. On a handheld computer using the ARM CPU, the size of the cache line is 32 bytes. The data cache stores a specific number of

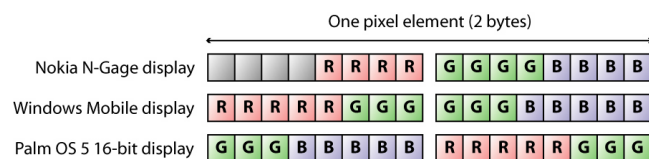


Figure 2: Three mobile display configurations.

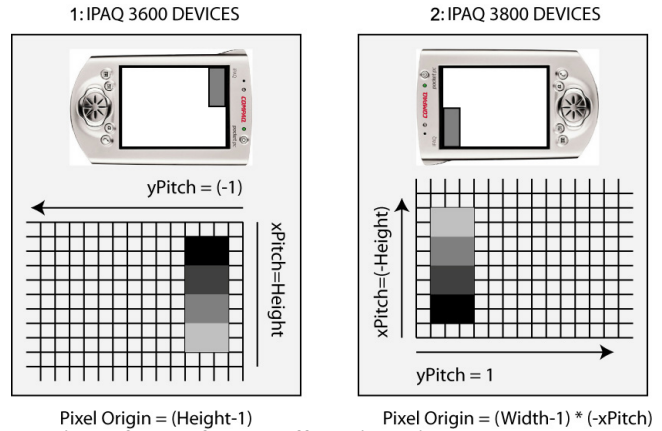


Figure 3: Two frame buffer orientations.

cache lines (the ARM CPU stores 256 cache lines, which equals 8kb of data cache), its contents depending on what data that was last requested. The advantage of using a data cache is that data retrieved from the cache is accessed significantly faster from it than data from the main system memory.

Using a display format such as the one used on stationary computers, data is read and written to the display in a format that matches the data cache. Reading the first pixel will cause the cache to automatically contain the second, third, and up to the 32nd pixel of that same row (depending on the pixel format). On displays that are rotated, such as those used on many handheld computers, the alignment of the display must be analyzed so that pixels are read in a way so that the cache memory is always optimally used. For example, in Figure 3, eight pixels should be copied three times (3-1), and three pixels should be copied eight times (3-2). While issues related to reading and writing non-cache-aligned data to the frame buffer is not new (e.g. the 8 by 8 display [15]), it was a long time ago developers had to consider aspects such as these when developing applications for stationary PCs.

3.3 Display Orientations

While the displays on mobile devices are stored internally in various orientations, they are typically presented to the developer in a portrait orientation (where screen width is less than screen height). Many applications however require a rotated landscape mode to operate correctly (where the screen height is less than screen width). Many video cards on stationary PCs support display rotations in hardware, and simply present the developer with a display with a different aspect ratio – the display is accessed using the same display format, and the video card does the final rotation transparently. Supporting display rotations on mobile devices however requires taking both the visual display rotation and the internal frame buffer orientation into consideration. This means that all image pre-rendering and all graphics operations must consider both of these aspects for optimum performance.

4. THE GAPIDRAW PLATFORM

GapiDraw was initially created for educational use. In the fall of 2001, the primary author was responsible for a course in mobile application development. The goal of the course was to teach

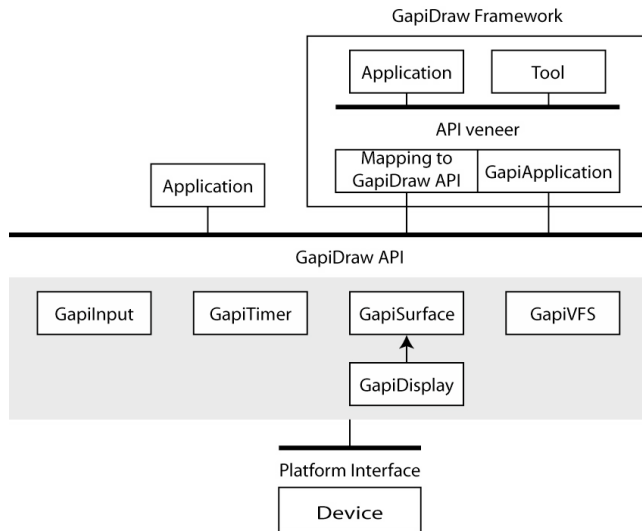


Figure 4: The GapiDraw platform.

students write networked interactive applications for handheld devices. To assist the students in their work two software platforms were created: the networking platform *OpenTrek* [13], and the graphics platform *GapiDraw*. It was not until March 2002 that *GapiDraw* was released on the Internet, and it was instantly picked up by a variety of software developers.

GapiDraw has two main components (as seen in Figure 4): a cross-platform Application Programming Interface (API) and a cross-platform Framework. The *GapiDraw API* comprises several classes for accessing the display, loading and manipulating images, and applying real-time effects such as opacity. The *GapiDraw API* was designed to be similar to the DirectX API in use, so that existing applications for stationary PCs could be transferred to mobile devices with a minimum of work, and vice versa. Some of the commercial games that have been ported from a stationary PC to a mobile device using *GapiDraw* are *Warlords II* and *Atlantis – Redux*. The *GapiDraw Framework* contains a single base class *GapiApplication*, which initiates full screen mode, enables application switching, captures messages from the operating system, and forwards button and stylus input to the application. Using *GapiApplication*, all device-specific logic is hidden beneath a cross-platform interface.

4.1 GapiDraw Components

The various components of *GapiDraw* are packaged as classes in a dynamically or statically linked library, depending on the target device. We will now describe the various components of *GapiDraw* and motivate the design choices made during the implementation process.

4.1.1 GapiApplication

GapiApplication is a cross-platform application shell that is used as an interface between the physical device and applications created with the *GapiDraw* platform. *GapiApplication* contains code for retrieving necessary information about the hardware configuration and pass this on to the necessary functions in the *GapiDraw API*. An application created with *GapiApplication* does not have to consider issues related to device hardware or operating systems, and *GapiApplication* can thus be used to enable cross-

platform prototyping on mobile devices. *GapiApplication* captures button, keyboard and stylus input and passes them on to the application in a cross-platform format. *GapiApplication* initializes full screen mode on the device and locks all hardware keys for exclusive access. *GapiApplication* also captures operating system events (such as those received when the device is switched on and off) and correctly notifies the operating system when the application requests to be minimized or otherwise wants to interact with the operating system.

The use of an application framework such as *GapiApplication* is commonly used by other graphic libraries for cross-platform development, such as GLUT [11] for Open GL development. *GapiApplication* however is the first application framework for creating cross-platform applications for mobile devices such as Palm, Symbian and Windows Mobile, not including interpreting language environments such as Java.

4.1.2 GapiInput

GapiInput captures all button events and forwards them to the application. Key codes are transformed to a common format across all devices, and they are also mapped to the current display rotation of the device. For example, if the display is rotated 90 degrees counter clockwise, if the user presses the “Up” key it will be reported to the application as the “Right” key.

Even though *GapiInput* uses a cross-platform API, there are still several cross-platform considerations that have to be done by the application developer. Many mobile devices do not have touch displays (e.g. most mobile phones), some devices do not have any buttons (e.g. the Sony Ericsson P900 only have a navigation stick in full screen mode), and some more advanced devices have it all – lots of buttons, touch displays, and even analog button devices (such as the Tapwave Zodiac). The variations in input options can in some cases make it impossible to create cross-platform versions of a mobile application, even though it is technically feasible.

4.1.3 GapiSurface

Images in *GapiDraw* are stored in *surfaces*. A surface in *GapiDraw* is an object, comprising a set of functions for initialization and image manipulation, and a memory area located in either system memory or video memory (depending on the hardware capabilities of the device). Images can be loaded from files (either physical files or files stored in the *GapiVFS* virtual file system), an image resource embedded in the executable file, or from an image located in system memory. To simplify development and to enable prototyping, the surface object in *GapiDraw* comprises a wide range of features, such as copying surfaces with variable opacity (alpha blend), drawing tools such as rectangles and lines, font tools, and real time rotation and scaling. If the proper video hardware is available to accelerate surface operations, *GapiDraw* will use this video hardware seamlessly. *GapiDraw* surfaces support clipping in all operations (see e.g. [7]), either to the entire surface or to a specified rectangle. *GapiDraw* stores all surfaces in a native format matching the frame buffer color depth, the frame buffer orientation and the display rotation of the device – all images are automatically color-converted and rotated as necessary when they are loaded.

One design feature of *GapiSurface* is that it is possible to subclass it to re-use its features. One such subclass is *GapiDisplay*, allowing all drawing tools available in *GapiSurface* to be used to draw graphics directly to the device frame buffer. Other examples of subclasses available in the *GapiDraw API* are

GapiMaskSurface, *GapiBitmapFont* and *GapiCursor* (not shown in figure 4). *GapiMaskSurface* is a surface specifically created for Z-buffer based collision masks. This type of collision masks are typically used in 2D adventure games. *GapiBitmapFont* is a bitmapped font class. Since *GapiBitmapFont* is a *GapiSurface* subclass, developers can change the font in real time using all of the drawing tools available. *GapiCursor* is an animated, alpha blended cursor class for applications that run on devices without a touch screen.

4.1.4 GapiDisplay

GapiDisplay is a subclass to *GapiSurface* and provides a cross-platform interface to the frame buffer of the device. Where on stationary PCs it is possible to configure the display mode such as color depth and double buffer mode, mobile devices all use different display modes and it is in most cases not possible to change the display configuration on a specific device programmatically. *GapiDisplay* thus supports all variations in display configurations and simply reports back to the application what modes are available (such as direct frame buffer access). *GapiDisplay* also manages surfaces stored in video memory, and provides support for many of the more proprietary device features, like the possibility to automatically resize the display from 320x320 to 320x480 on the Palm Tungsten T3 handheld computer.

4.1.5 GapiTimer

GapiTimer serves two purposes – first it can be used to synchronize the updates to the display to the vertical blanking period of the device, secondly it can be used to limit the number of frame updates each second for the application. Limiting the number of frame updates each second is necessary to improve battery life of the mobile device. If a mobile application continuously updates the display as many times as possible each second, the processor usage would run at a constant 100% and the batteries would be drained quickly. On devices having a timer with a resolution of 1 millisecond or better, *GapiDraw* provides the option to limit the maximum number of frame updates each second to save batteries. The *GapiTimer* implementation is similar to the timer included with the SDL graphic library [16], in that it analyzes the time all previous frames have taken to render and then calculates how long it should wait in milliseconds before rendering the next frame. If the last frame took longer to render than allowed, *GapiTimer* notifies the application, skips the next frame and resets the frame time history.

4.1.6 GapiVFS

Some mobile devices (such as devices manufactured by Palm and Tapwave) do not include a hierarchical file system. On other devices with a “real” file system, file management differs greatly between devices, making it difficult to distribute applications on multiple hardware platforms. *GapiVFS* is a cross-platform virtual file system that stores all images, sounds and other resources in one single database file. This database can then be included either as a resource (on Palm, the database is automatically split into multiple 32kb resources that are automatically merged by *GapiVFS*) or as an actual file, depending on the device. *GapiVFS* supports folders and subfolders, and files stored in the virtual file system can be individually compressed using zip compression to

```
// One loop for all flag combinations
<loop through all rows in surface>
{
  <loop through all pixels in current row>
  {
    <if predicate(pixel)>
      <pixeloperation(sourcecopy(pixel))>
  }
}
```

Figure 5: Syntax code for a pixel loop in *GapiDraw*.

save memory storage (*GapiVFS* automatically decompresses these files when they are loaded).

5. IMPLEMENTATION

The *GapiDraw* source code is written in C++ and assembler. The source code is split into two parts: one large common part for all devices, and one minor, customized part for each unique operating system. *GapiDraw* uses an optimization technique called *template meta programming*. Template meta programming is a C++ programming technique that allows optimizations such as loop-expansion and automatic means of creating temporary variables in optimized code. Template meta programming is a recently introduced technique [17], and has previously been used in class libraries such as Blitz++ [2]. *GapiDraw* uses the template meta programming technique for loop-expansion. In most graphics operations, *GapiDraw* accepts several flag parameters that changes how pixels are to be copied between surfaces. Examples are transparency, color mask and colorization. Implementing support for three flags, a developer can write either one loop that covers all flags, or eight individually optimized loops, where each single loop is optimized for one unique flag combination. Writing eight different loops to support three flags, means having to write another eight loops to support an additional frame buffer color depth. With *GapiDraw* accepting up to eight different flags as parameters to some operations, and supporting four different frame buffer color depths (12-bit, 15-bit, 16-bit and 16bit PalmOS5), writing individually optimized loops for each unique flag combination proved to be impossible. Writing one loop that supports all possible flag combinations by itself however imposes significant performance issues, in that all flag combinations have to be checked for every single pixel being copied.

GapiDraw's pixel loops are created with template meta code and comprise three virtual function calls that are inlined by the pre-compiler: *<predicate>*, *<pixel operation>*, and *<source copy>*. By adding a unique call to the same loop for each flag combination (and changing the predicate function, pixel operation function, and source copy function for each call), the pre-compiler expands the code for all unique flag combinations before passing them on to the compiler. The end result is that each single pixel loop only needs to be written once, and is then automatically expanded into $(2^n)*i$ unique loops, where n is the number of unique flag variations, and i is the number of frame buffer formats supported. A typical *GapiDraw* loop is shown in Figure 5. *Predicate* determines if the source pixel should be copied or not (for example, if pixels matching a certain color should be ignored). *Pixel operation* declares how the source pixel should be blended with the destination (for example using a pixel shader such as an alpha blend). *Source copy* defines what will be copied to the destination, such as the source pixel or a pre-defined color

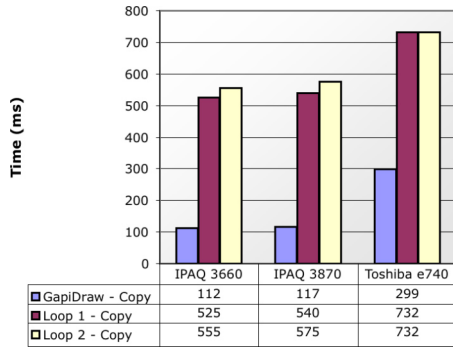


Figure 6: Time to copy 1000 images to the display.

value. If the loop in Figure 5 takes eight flags as an argument and supports two different color depths (12-bit and 16-bit), the pre-compiler expands it into 512 separately optimized loops at compile-time.

6. PERFORMANCE TEST

To test the graphics performance of GapiDraw we wrote a simple test application using code samples from the book Pocket PC Game Programming [8]. The samples in the book do not consider frame buffer rotations and do not use template meta programming. We wanted to test two aspects of drawing graphics to the display: First, what difference in performance can one expect from not considering the frame buffer orientation of a mobile device? Secondly, how much does the use of template meta programming affect performance?

The test application we implemented copies a bitmapped logo (128x40 pixels) to the display, once using GapiDraw and then using simple pixel-by-pixel loops. The application was tested on three different Pocket PC devices – all with different frame buffer orientations (an IPAQ 3630, an IPAQ 3870, and a Toshiba e740). The display of the Toshiba e730 is not rotated, the IPAQ 3630 display is rotated 90 degrees counter clockwise, and the display of the IPAQ 3870 is rotated 90 degrees clockwise.

The GapiDraw part of the test application was created by extending the GapiDraw Framework. The image logo was loaded into a surface, and then simply copied 1000 times to the display using the GapiDraw API. The test was run ten times on the three devices to get an average time to copy the images. To create the pixel-by-pixel loops the image was first loaded into a memory area and converted to the frame buffer color depth. Two separate loops were then implemented. The first loop copies the image to the display pixel-by-pixel. The second loop takes a Boolean value as a parameter. If the value is true, the image will be copied to the display, if the value is false, nothing will be done. The second loop will show the performance difference of using template meta programming for loop-unrolling, when comparing it in performance to the first loop. Pixels are copied on a row-by-row basis: for every row ($0 \leq n < \text{height}$), all pixels are copied ($0 \leq n < \text{width}$).

6.1 Test Result

The test result is seen in Figure 6 and shows the performance difference in ms of copying an image aligned to the frame buffer orientation, versus not considering the rotation of the frame buffer. Simply by pre-rotating all images to match the frame



Figure 7: Students playing the game *Pac-Man Must Die*, created in three weeks with GapiDraw.

buffer orientation, GapiDraw can copy up to two pixels simultaneously using 32-bit data reads and writes. The performance difference of copying an image aligned to the display using GapiDraw is up to 4.7 times faster than copying the image unaligned (using the first pixel-by-pixel loop on the IPAQ 3630 device). The option flag introduced in the second loop decreases performance up to 6.5% as compared to the first loop when the test application is run on the IPAQ 3870 device.

The performance decrease of adding an option flag and not using template meta programming in this test might not seem that significant (the difference was up to 6.5%). This performance decrease however scales exponentially with the number of flags added – meaning that four flags would slow down loop 1 up to 104%! By using template meta programming, the performance in GapiDraw is not affected by the number of flag combinations exposed, which in some operations can include up to eight variations.

7. GAPIDRAW AS AN ENABLER PLATFORM

GapiDraw was designed to enable cross-platform application prototyping on mobile devices. As such, it has been widely used in numerous university courses, research projects, and commercial game development projects.

7.1 Education

For two years we held a course in software development on mobile devices at a local university. Using our platforms GapiDraw and OpenTrek [13], the students were instructed to create games that required people to collaborate to succeed. The students were assigned only three weeks to implement the games. A total of 12 games were created each year, where the students worked in groups of 2-4 people on each game.

One of the games created by the students was *Pac-Man Must Die*. The game is played similar to the traditional game *Pac-Man*, where the player needs to collect dots in a labyrinth while avoiding enemies. However, some of the dots are located on the displays of other players' devices! The player can enter another person's handheld display by using "doors" at the edges of the map. When a player has entered the display of another computer she has to look at the other user's display to control her ghost (as seen in Figure 7). The game was recently tested in a use study [14], where players quickly found out that they could run away

with their displays, preventing other people from controlling their ghosts when they are at another person's display.

7.2 Research

GapiDraw is used by several researchers for application prototyping on mobile devices. Examples of such applications are *Tilt and Feel*, *PlaceMemo*, *Slide Scroller*, and *Total-Recall*. The Tilt and Feel project [10] explores the use of a mobile device augmented with a tilting sensor and a vibrotactile transducer. Using GapiDraw, the project group created several sample applications during the design phase of the project, including a tilt-driven maze game, an address book and a map application. PlaceMemo [5] runs on mobile devices and uses GPS positioning to allow road inspectors to connect voice notes to a geographical location. Using GapiDraw, the researchers implemented a navigational tool that allowed real-time zooming and manipulation of a map of the user's current location. The Slide Scroller prototype [6] uses a mobile device augmented with an optical mouse sensor. By "scrolling" the device, it is possible to navigate large documents such as web pages on the small display. In the project, researchers used GapiDraw to test several application concepts on the actual device in the design phase. Finally, in the Total Recall project [9], handheld computers are augmented with an ultra-sonic positioning system to introduce a new way to view captured whiteboard annotations – in place, where they were drawn. Using GapiDraw, the first Total Recall test prototype running on an actual mobile device was created in just one day.

7.3 Commercial games

GapiDraw has been used in more than 100 commercial games for handheld computers, including titles such as *EverQuest for the Pocket PC* by Sony Online Entertainment, *Warlords II* by Pocket PC Studios, and *Atlantis – Redux* by TetraEdge / DreamCatcher Europe. Due to the increased number of companies that used GapiDraw commercially, the GapiDraw project was moved to a separate company in May 2004 to manage sales and support.

8. CONCLUSION AND FUTURE WORK

We have presented the GapiDraw platform, which supports the creation of cross-platform applications with high-performance graphics on mobile devices. GapiDraw implements numerous optimization techniques for the heterogeneous hardware designs of mobile devices, while at the same time providing an easy to use cross-platform API and an extensible framework suitable for prototyping. Based on the extensive use of the platform in commercial applications, educational settings and research projects, we argue that GapiDraw can play an important role as an enabler platform to implement and evaluate new application concepts for mobile devices, on actual mobile devices. Future work related to the platform has already begun, with current focus on exploring new ways to make mobile 3D graphics more accessible for prototyping and educational use.

9. ACKNOWLEDGEMENTS

This research is funded by the Swedish Research Institute for Information Technology (SITI), and the Mobile Services project financed by the Foundation for Strategic Research (SSF). "Pac-Man" is a registered trademark of Namco, Inc.

10. GAPIDRAW DOWNLOAD

The GapiDraw platform and documentation can be freely downloaded from the web at: www.gapidraw.com

11. REFERENCES

1. Bechtolsheim, A. and Baskett, F.: High performance raster graphics for microcomputer systems. *Computer Graphics 14*, 3 (July 1980), pp 43-47.
2. Blitz++, <http://www.oonumerics.org/blitz/>
3. DirectX, <http://www.microsoft.com/directx/>
4. Droneship Software, <http://www.droneship.com/>
5. Esbjörnsson, M. and Juhlin, O. PlaceMemo - Supporting Mobile Articulation in a Vast Working Area through Position Based Information. In proceedings of the *European Conference on Information Systems*, 2002, Gdansk, Poland.
6. Fallman, D., Lund, A., and Wiberg, M.: Inside-Out Interaction: An Interaction Technique for Dealing with Large Interface Surfaces such as Web Pages on Small Screen Displays, to be presented at *SIGGRAPH 2004, Sketches program*, Los Angeles, USA.
7. Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J.: *Computer Graphics: Principles and Practice*, second edition, Addison-Wesley, 1990.
8. Harbour, J. S.: *Pocket PC Game Programming*, Muska & Lipman / Premier-Trade, 2001
9. Holmquist, L. E., Sanneblad, J., and Gaye, L.: Total Recall: In-Place Viewing of Captured Whiteboard Annotations. In *Extended Abstracts of CHI 2003*, Fort Lauderdale, Florida, United States.
10. Oakley, I., Angeseleva, J., Hughes, S., and O'Modhrain, S.: Tilt and Feel: Scrolling with Vibrotactile Display, in *Proceedings of EuroHaptics 2004*, Munich, Germany
11. Open GL, <http://www.opengl.org/>
12. Razor Graphics Engine, <http://www.tilo-christ.de/razor/>
13. Sanneblad, J. and Holmquist, L. E.: OpenTrek: A Platform for Developing Interactive Networked Games on Mobile Devices, in *Proceedings of Mobile HCI 2003*, Udine, Italy
14. Sanneblad, J. and Holmquist, L. E.: Why is everyone inside me?! Using Shared Displays in Mobile Computer Games, to be presented at the *3rd International Conference on Entertainment Computing, 2004*, Eindhoven, The Netherlands.
15. Sproull, R. F. and Sutherland, I. E.: The 8 by 8 Display. *ACM Transactions on Graphics*, Vol. 2, No. 1, January 1983, pp 32-56.
16. The Simple DirectMedia Layer, <http://www.libsdl.org/>
17. Veldhuizen, T. and M. E. Jernigan. Will C++ be faster than Fortran? In proceedings of the *International Scientific Computing in Object-Oriented Parallel Environments*, 1997, Marina del Rey, California, United States.