Chapter 2

From QuickDraw to Quartz 2D

Before launching into an in-depth discussion of the features of Quartz 2D or a set of drawing techniques, this chapter begins by placing the library in context. The Quartz 2D library has a long legacy behind it and understanding that legacy may help you to recognize some of the idiosyncrasies of the library's imaging model. Moreover, Quartz 2D is a single part in a larger graphics architecture on Mac OS X. This architecture was designed from its inception to take advantage of advances in the graphics hardware of modern personal computers. This chapter begins with a historical perspective of the Mac OS X graphics system and the evolution of technology behind Quartz 2D. We then explore crucial aspects of the graphics technology that is shaping the graphics architecture of the Macintosh.

The Legacy of QuickDraw and the Rise of PDF

The QuickDraw graphics library stands to this day as testament to the original Macintosh computer and its creators. The library was the fundamental technology that made the original Macintosh graphical user interface possible. QuickDraw was brought to life by the ingenuity and skill of software designers such as Bill Atkinson and Andy Herzfeld. Apple created QuickDraw in the late 1970's and early 1980s, an era in which "real" computer graphics were the province of large, powerful mainframe computers, and personal computers were just making the transition from novelty to necessity.

QuickDraw had very modest beginnings. In various forms, the original code ran on computers like the Apple Lisa and the original Macintosh. The high resolution bitmapped displays of these computers was considered a revolution when compared to the character terminals of the previous computer generation. In spite of their sophistication, however, the computers could only display graphics in black and white.

QuickDraw was flexible enough to produce impressive graphics on both the screen and on printer. The library incorporated a number of revolutionary advances, features that were not found on personal computers prior to the Macintosh. Among these were the support for pixel regions, drawing operations that could be recorded into a meta-file (the infamous PICT file format), the ability to scale the drawings in a meta-file on playback, and drawing primitives for ovals, curves, and rounded rectangles.

Although QuickDraw and its immediate ancestors ran on computers that could only display black and white, the API actually supported several colors. Even though the computers could not display those colors on screen, QuickDraw could print in color to some printers.

The 9-inch black and white display of the original Macintosh quickly became a thing of the past. Computers evolved, as did the sophistication and requirements of applications and users. Apple evolved QuickDraw along with its computers. As displays became capable of reproducing millions of colors and two tone dot-matrix printers evolved into high-resolution, photo quality ink jet printers, QuickDraw both kept up the pace and pushed the envelope of graphics evolution. The era of QuickDraw ended, however, when Apple deprecated the technology in Mac OS X 10.4 Tiger. The QuickDraw library came of age as an important building block for vital graphics technologies like QuickTime and ColorSync. Along the way it not only served the graphics industry, but also took a hand in shaping it.

PostScript and Desktop Publishing

QuickDraw was not the only technology born in 1984 to have a profound impact both the graphics industry and the Mac OS X platform. In the same year, Chuck Geschke and John Warnock incorporated a new company, Adobe Systems Inc. Adobe released the first version of their PostScript graphics system that same year.

The PostScript graphics system grew out of research Adobe's founders performed while working at Xerox. That research centered around innovative ways to write control software for laser printers. The PostScript system combines a rich graphics model, a simple programming language, and a run-time environment. At the time of its introduction, the system's ability to repurpose graphics on a broad number of printers with very different capabilities was a clarion call for the graphics industry.

PostScript was unusual because of the device independence inherit in the system. A PostScript program could send the same drawing commands to two printers with very different capabilities, and both would reproduce the same graphic to the best of their abilities. One printer might draw the graphic with a low resolution and in black and white, while the same program on another printer might generate a high-resolution color image. This was in stark contrast to the fixed resolution, device dependent nature of QuickDraw and other graphics libraries.

The paths of QuickDraw and PostScript were destined to converge. Apple and Adobe brought these two technologies together when they developed the LaserWriter printer. In spite of the fact that QuickDraw had one drawing model, and PostScript a completely different one, application developers combined the on-screen drawing with QuickDraw and the printing might of PostScript. This synthesis led to the creation of creative applications such as Aldus PageMaker and Adobe Illustrator. Applications like these freed document editors from the proprietary typesetting systems that had dominated the industry. The Desktop Publishing revolution had arrived.

PostScript on the Screen

The rift between the graphics model of the screen and the graphics model of the printer continued for some time. In fact, this was not limited to the Macintosh platform. Microsoft Windows, for example, used the GDI graphics library when drawing on screen. On many UNIX systems, applications used the XLib library of the X11 windowing system to create graphics on the screen. Both GDI and XLib are graphics libraries very similar to QuickDraw. Each of these environments also supported PostScript as a tool for creating high-quality graphics on the printer.

In this environment, application developers applied their creativity and ingenuity to bridge the gap between libraries and create WYSIWYG applications. It wasn't long before enterprising developers realized the potential benefit of using PostScript for graphics on the screen as well. In fact, one of the earliest implementations of this idea came when Sun Microsystems created an entire windowing system based on PostScript! That windowing system was called NeWS, the Network extensible Windowing System.

NeWS was unusual because it used PostScript as more than just a graphics library. The windowing system itself was implemented on top of a customized PostScript interpreter. Developers could extend the system or write NeWS applications in PostScript. In spite of the novelty and innovation in NeWS, however, it never gained much of a foothold in the industry. Eventually it faded from view.

The lead engineer of NeWS was Jim Gosling. After working on NeWS, he turned his attention to Java where he designed the original language, compiler, and virtual machine.

The engineers at Sun were not the only group to bring PostScript to the screen. One of the most successful environments to integrate PostScript into its graphics architecture was the NeXT Computer operating system. Steve Jobs started NeXT Inc. shortly after leaving Apple. The computers that company created were revolutionary in a number of ways. The NeXT operating system included an optimized PostScript interpreter. Applications created their graphics in windows by calling routines invoked the PostScript interpreter.

NeXT sold a PostScript laser printer that didn't include a PostScript interpreter! The system relied on the computer to process the PostScript and then sent the resulting graphics to the printer over a high-speed communication line.

The success of the NeXT graphics library caused other industry leaders to take notice. Adobe collaborated with NeXT to create a standard called Display PostScript for PostScript graphics on the screen. Display PostScript incorporated a small set of extensions to the basic PostScript language. In time, Adobe licensed Display PostScript for use in other computing environments. The X11 applications on some UNIX computers could create windows that contained Display

PostScript graphics. Display PostScript, like the NeWS window system, never gained a broad acceptance in the industry. When Apple purchased NeXT Computer, in 1996, it acquired all of its technologies, including its implementation of Display PostScript.

PDF to Quartz 2D

Even though Display PostScript enjoyed limited popularity, the appeal of PostScript continues to this day. There is, however, one evolution that is particularly relevant to the Macintosh and Quartz 2D.

The PostScript system includes both an imaging model and a programming language. While the flexibility of the imaging model continues to this day, the programming language aspects of PostScript have proven to be problematic.

PostScript interpreters run PostScript programs. Like most programs, PostScript code must execute sequentially. This can lead to difficulty when printing documents. For example, if a press operator wants to print page 99 of a 100-page document, the PostScript interpreter must execute the code that draws the first 98 pages. The results of that drawing take time, and the resulting graphics are merely discarded. This is obviously a waste of time and resources.

Furthermore, because PostScript drawings are actually programs, they are susceptible to programming bugs. To continue the example just discussed, a logic error on page 97 might cause the printer to abandon the entire print job. As these problems came to light, Adobe added some features to PostScript. The new features helped to alleviate problems but did not eliminate them. In the end Adobe recognized the limitations of PostScript and took another tack on the problem.

The Portable Document Format (PDF) works around many of the limitations of PostScript. At its heart, PDF is a file format that includes the graphics features of PostScript (with a few minor changes). At the same time, the standard removes most of the programming language aspects of PostScript. A PDF document is not an executable program so much as it is a structured container for drawing commands and related metadata. The file format continues PostScript's advantages of device and resolution independence. In short, PDF retains the graphical power of PostScript but eliminates some of its less reliable aspects.

Apple evaluated the technologies they purchased for NeXT with a eye toward creating Mac OS X. They looked at the evolution of PostScript and the PDF strategy. Combining these elements with their own unique flair, Apple developed a new graphics library for Mac OS X that combined the richness of the Adobe Imaging Model with alpha channel support in the high-performance graphics library we know today as Quartz 2D.

Understanding this legacy can be an important part of understanding why the Quartz 2D imaging model behaves as it does. For example, Quartz 2D does not provide an easy mechanism for erasing graphics you have already drawn. The reason is that when you are drawing to a printer, the graphics may not be going to a frame buffer, and there is no way to erase graphics. Instead of erasing graphics, in Quartz 2D you create a mask, or clipping area, and draw the graphics you want the user to see relying on the mask to remove parts of the image you don't want to display. This is a simple example but very illustrative of how the drawing model of the library can affect how it is used.

Graphics Programming in the Modern Age

One advance in technology that contributed to the demise of QuickDraw was the trend in modern personal computers to move more of the graphics capabilities of the system to video cards. Indeed the advent of video cards with dedicated graphics processing units has ushered in a new age of computer graphics. Quartz 2D and Core Image take advantage of these recent developments to improve their functionality and performance. Core Image in particular is a direct benefactor of the power of modern graphics hardware. The evolution of the graphics system in personal computers is extending the reach of those machines to new and exciting fields of endeavor. The Mac OS X graphics system is at the forefront of this technologies wave. By using modern graphics APIs like Quartz 2D, your application can take advantage of the work Apple has done, and you can enjoy the benefits of the hardware while concentrating on a simple interface.

By way of an example, consider the impact that modern personal computers have had on video production. We live in an age where studios use computer graphics to create full-length, animated feature films. In the past, digital video production houses used expensive, dedicated workstations to produce their films.

Just as PostScript shifted print publishing from proprietary systems to the desktop, the development of software such as Final Cut Pro has professional quality video editing onto consumer computers. This transition works because of a combination of hardware improvements and the advancement of the graphics systems on personal computers.

In recent past, images that used 32 bits per pixel and alpha channels could only be manipulated by high-end applications. Today, however, these images are commonplace. The CCT chips in modern digital cameras can capture images using 12 bits per color channel or more. Storing these images in an 8 bit per channel image drops valuable color information. The high-end applications of today may choose to use a full 32-bit floating point value to represent just one color channel. Each pixel, therefore, requires 128 bits. An image with the same dimensions may require four times the storage just to hold the additional color information! Processing such an image requires the computer to sift through four times the data.

Shuffling around large volumes of pixel data is one difficulty. The color channels in these images are stored in floating point representations. Correspondingly, performing calculations on those pixels requires floating point math. Computing at this level requires significant processing horsepower and efficient use of graphics resources. Computer scientists have answered the demand for greater graphics processing power by adding dedicated computer graphics hardware to personal computers.

General Purpose Vector Processors for Graphics

A good example of the evolution of hardware with a corresponding impact on graphics is the addition of vector processing units to general purpose microprocessors. On the Macintosh platform, for example, the G4 and G5 PowerPC processors have a vector processor known as the Velocity Engine. Developers will recognize it by its geeky name, AltiVec. Intel-based processors include SIMD technologies like MMX or SSE.

AltiVec will serve as a good example of a general purpose vector processor. The registers of the AltiVec unit store quantities that are 128 bit wide. The processors instructions treat those bits as vectors. Depending on the instruction, the processor will interpret the 128 bits as a vector whose components have different

lengths. Figure 2.1 shows the different ways that AltiVec processors can interpret 128 bits.

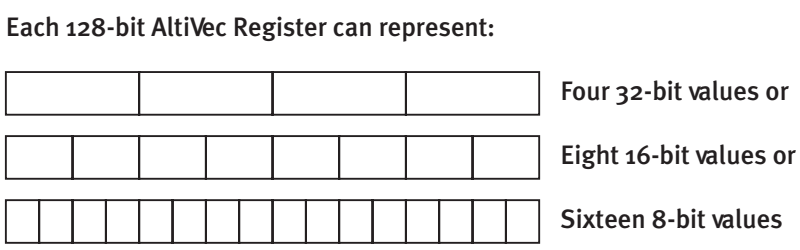


Figure 2.1 AltiVec Register Configurations

When interpreting a vector of four 32-bit values, the processor can treat those bits as either a 32-bit integer or a 32-bit floating point value. A graphics application might feed the AltiVec unit with 16 pixels of an 8-bit grayscale image all at once. The program could then lighten all 16 pixels at once using a single AltiVec command.

Using different instructions, a program could also load an AltiVec register with the four 32-bit floating point numbers that make up a single floating point ARGB pixel. The AltiVec processor could combine two floating point ARGB pixels in a single operation. From these two examples, it's easy to see how the processing muscle of a vector unit like AltiVec can improve graphics performance.

One shadow that complicates the use of vector processing units for graphics is the fact that the AltiVec unit is not dedicated to graphics alone. Computer games are popular clients of the graphics system. Many games contain computing engines that handle physics calculations. Physics involves working with vector-valued quantities like velocities and accelerations. These calculations are also a good fit for implementation with the vector processor. Scientific visualization applications also rely heavily on the graphics system and include algorithms that benefit from the vector processor. In many applications, the vector processor is shared between the graphics systems and other computation engines.

Computers with two or more microprocessors often have the added luxury of a second vector processing unit, and applications can employ that to alleviate some of the congestion. Unfortunately, there is a practical limit that prevents computers from scaling performance through the addition of processing units.

The complexity of a general-purpose vector processor means that they also take up quite a lot of space on silicon chips. Correspondingly, the amount of power they require and the amount of heat they generate increases with each additional unit. Issues like these make it very hard to scale the performance of a general vector processor by simply adding additional cores. As will be shown, however, if we reduce the complexity of the core, limit the operations it can perform, and focus it on a specific task, the idea of scaling vector processing power this way actually works quite well.

The Emergence of the GPU

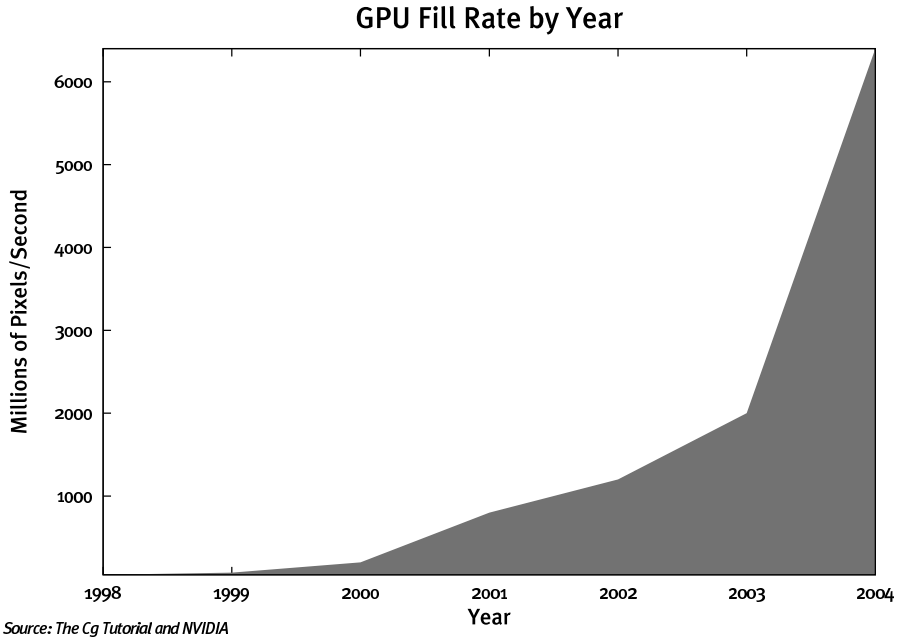
Another common technique to boost the graphics performance of a computer is to augment the CPU with an additional processor that is dedicated graphics. On the personal computer systems of today, that additional processor is usually found on the computer's video card.

Many of the earliest models of graphics coprocessors, particularly in the personal computer space, were simply tools to speed up some very specific parts of the 3D graphics pipeline. The cards had algorithms for applying lighting and shading models to simple geometric primitives like triangles. The algorithms were hard-wired into the video card and could not be changed. Communication with these cards flowed in one direction only, from the main computer to the video card. The cards were useful for rendering 3D graphics efficiently but could not be used in more general graphics applications.

As time progressed, the services provided by the video card's processor expanded to include more general purpose routines. The data path between the main CPU and the graphics card widened and became bidirectional. With those innovations, programs gained the ability to use the graphics processor to perform calculations and retrieve the results to main memory. This allowed the video card to behave as a graphics computation engine, not just a display mechanism.

Collectively, these more powerful processors have come to be known as Graphics Processing Units or GPUs. They play a significant role in boosting the graphics capabilities of modern computers. Along with the GPU, a typical video card will also contain a block of dedicated memory (called Video RAM or VRAM) and some kind of hardware that converts bits in the cards display buffer into video signals for a computer display. In many respects, the video card resembles a

self-contained graphics computer. Like other computers, graphics hardware continues to advance. To give you some idea of how rapidly video cards have evolved, consider the graph shown in Figure 2.2.



Source: The Cg Tutorial and NVIDIA

Figure 2.2 GPU fill rate by year

Figure 2.2 shows how the fill rate of graphics processors has grown over the years. A graphics card fill rate is roughly the number of pixels that the GPU can draw into video RAM in a single second. While the true processing power of the GPU varies for different tasks, this graph is a dramatic example of the advances that have been made in graphics processing power of GPUs.

The growth of graphics processing power actually exceeds the predictions of industry professionals. In 1965, Gordon Moore made a famous observation that the number of transistors on a single chip would double approximately every 18 months. This prediction has come to be known as Moore's Law. The computing industry has taken Moore's law to imply that the processing capabilities of silicon chips would grow at the same rate.

For many years the transistor counts and performance of CPUs has tracked this prediction with frightening accuracy. The graph in Figure 2.2 shows that the

performance of GPUs has been growing faster than Moore's Law would predict. In general terms, this means that the processing power of dedicated graphics processors is growing faster than that of general purpose CPUs like the PowerPC or the Intel x86 family. Applications that tap into that processing power enjoy dramatic performance improvements.

One of the reasons that graphics processors follow this performance curve is because the performance of the processor is easier to scale by throwing more silicon at the problem. Many of the algorithms that the GPU runs are what computer scientists call "embarrassingly parallel." An embarrassingly parallel problem is one in which a computer can easily work up a solution breaking it into smaller pieces and computing each piece along a parallel path.

Astute readers will recognize how a program might apply an SIMD vector processor, like the aforementioned Velocity Engine, to calculate a solution to an embarrassingly parallel problem. But graphics processors don't need to solve the same problems that general purpose vector units must solve. Because it can focus on solving graphics problems, the GPU requires fewer operations. For example, general purpose processors must deal with branches, loops, and error checking. In contrast, the GPU pushes its vectors through sequentially without branches and loops. Each of the parallel units in a GPU is much simpler than its counterpart in a more general vector processor. Hardware engineers can add more vector units, and therefore more parallel computation paths, in the same area. More computation paths mean more operations completed each cycle.

Because the vector units in the GPUs are dedicated to graphics, they don't suffer the resource contention issues that plague general purpose vector processors. There aren't as many parts of applications competing for processor time.

The Programmable Graphics Card

Computers have spent many years sending data to graphics cards, but the ability to send programs to the GPU is a relatively recent innovation. Graphics cards that accept GPU programs from the main computer are known as programmable graphics cards. This ability to program the graphics card is the feature that lends power to graphics systems like Quartz 2D and, in particular, Core Image.

Even though the parallel execution units of graphics cards can only perform a limited number of operations, those operations can be applied to solve problems that have nothing to do with graphics. Some computer scientists are investigating techniques for using the video hardware to run computations that are unrelated to Graphics. The field is known as "General Processing on GPUs," and a number of enthusiasts host a Web site dedicated to the topic at `http://www.gpgpu.org/`.

At it's heart, Core Image is a system for feeding GPU programs to programmable video cards. The programs it submits usually apply special effects to images. By using the power of the parallel processing paths on the hardware, the computer can calculate those effects much faster than the main CPU could. Another interesting aspect of Core Image is that it can run its effects even if no programmable graphics card is available on the system. This demonstrates another advantage of the Mac OS X graphics architecture. It helps your applications produce improved performance without undue complexity.

Managing Hardware Complexity

The challenge to today's applications is finding a way to conveniently take advantage of the power afforded by modern hardware. For example, application programmers who want to use the Velocity Engine must learn the AltiVec instruction set. They must also develop "vectorized" algorithms for solving the application's problems. The applications often must rearrange data structures so that the vector processor can access them efficiently. All of these issues require specialized knowledge and add complexity to the resulting application.

In a similar way, making direct use of the GPU requires an application to understand some of intimate details about the video card. Some cards accept longer graphics programs than others, and some card have special instructions that simplify GPU code. Writing code that is general enough to support the diverse range of graphics hardware from different vendors is quite difficult. If you're writing your application directly to the hardware, you will have to immerse yourself in the minutiae of every hardware combination, or you must limit your application to only that small set of hardware you are willing to support.

This same problem applies even in computers that don't have video cards. Even tasks that appear simple, like copying pixel buffers in main memory efficiently, require a detailed knowledge of the processors' cache behavior and the system's

virtual memory architecture. While these are all interesting topics, writing graphics programs this "close to the metal" can be complicated and error prone. Providing a rich, full-featured graphics system that allows applications to plug into the performance of the computer, while allowing programmers to focus on creating graphics and not hardware issues, is one of the toughest challenges the operating system vendor must face.

Quartz 2D and Core Image are both excellent technologies in this regard. They insulate applications from the complexity of the hardware but take advantage of that hardware in their own implementations. For example, Apple includes a compatibility mechanism inside of Core Image that allows the system to run GPU programs on any system. If a program uses Core Image on a computer without a programmable GPU, the program will continue to run correctly. The effects of applying a filter will take longer to achieve, but the results should be the same even without the dedicated hardware. The application programmer limits his attention to working with the interface of Core Image instead the details of the GPU. Similar arguments can be made with respect to the features of Quartz 2D. The graphics architecture of the system insulates applications from hardware details and is easier to use.

Mac OS X Graphics Architecture

Mac OS X, in general, is built as a layered software system. The lower layers are the ones closer to the hardware. These layers include software like hardware drivers and routines for accessing processor specific features such as AltiVec. The higher layers build upon the functionality beneath them and offer applications services that are easier to use. The challenge of the application designer is to decide at what level he needs to access the graphics system. It is a balancing act between application complexity and performance.

Figure 2.3 illustrates the layers of the Mac OS X graphics system.

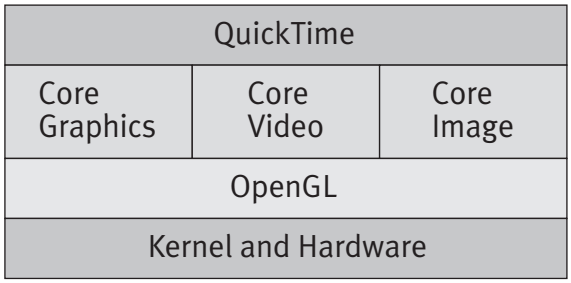


Figure 2.3 Mac OS X Graphics Architecture Overview

Figure 2.3 is meant to convey the layers of the graphics system and the ways that they are built on top of one another in the most general terms. Strictly speaking, the QuickTime software layer might interact more directly with the hardware, bypassing the layers beneath it, than this diagram would suggest. Nevertheless, the diagram is useful for describing the Mac OS X architecture. In the following sections, we will describe each layer in the Mac OS X graphics architecture and their roles in creating graphics.

Kernel and Hardware

The kernel and hardware layer represents the lowest levels of the operating system. The hardware includes both the components of the main computer and the chips on the video card. In terms of the main computer, the graphics system must often interact with the CPU, any vector processing units, and the memory system. This layer of the system handles the complexity of processor cache lines and complexities like processor-specific instructions. Hardware on the video card includes the presence of a programmable GPU, the amount of video ram, and the interconnection between the video card and main memory.

The kernel layer includes the video card drivers and other software that interacts directly with the hardware.

The software interfaces exported from the kernel and hardware layer encapsulate a tremendous amount of complexity. Applications that need the absolute highest level of performance may need to connect to the system at this level, but that is likely to be a very rare occurrence.

OpenGL

OpenGL was first released in the early 1990's and is widely known as an industry standard graphics library for creating 3D graphics images. Given its association with graphics accelerators, OpenGL is also a valuable tool for accessing the full capabilities of the video card. OpenGL is a cross-platform standard. A commission known as the OpenGL Architecture Review Board (ARB) oversees and steers the technology's development. The ARB's web site is `http://www.opengl.org`. It s a valuable repository for resources related to writing OpenGL code. The ARB site also contains innumerable links to other site, which makes it an excellent jumping off point for graphics programmers who want to know more about OpenGL.

Mac OS X includes a terrific implementation of OpenGL. Many 3D games and scientific visualization applications take advantage of the 3D graphics features of OpenGL. Interest in OpenGL in this volume, however, does not focus on its 3D graphics features. Nor will it concern itself with OpenGL's ability to serve as a first-rate 2D graphics library. Instead, the focus is on OpenGL as a rather direct interface to the kernel and hardware layers.

We already mentioned how early graphics cards were little more than shading engines and rasterizers for 2D and 3D graphics primitives. Applications would submit their primitives to the hardware through OpenGL. The library has evolved, in lock-step with the progress of video cards. As video card vendors add new capabilities to their hardware and drivers, the OpenGL community modifies their code to make those capabilities accessible to applications. Likewise, as OpenGL developers discover new and innovative graphics techniques, they first implement them in software. The most useful and popular may be bolstered with hardware implementation on future video cards.

When working with Quartz 2D, the system can use OpenGL to efficiently copy images onto the main display. OpenGL also submits GPU programs to the video card on behalf of Core Image. These are just two examples of how higher layers in the system can turn the features of OpenGL to their advantage. Applications that need to create 3D graphics will use OpenGL as a matter of course. 2D applications with very specific, high performance needs might also use OpenGL to communicate to the video hardware.

Core Graphics

As its name suggests, Core Graphics is one of the fundamental graphics systems on Mac OS X. Core Graphics is the proper name of the system, but it is also known by the marketing friendly term Quartz. Quartz has two primary subsystems—the window server and the Quartz 2D library.

The window server collects images of all the windows on the system, composites them together, and is responsible for the images displayed on all the computer screens. This system is also responsible for working with the computer hardware to collect user events from the mouse and keyboard and see that they find their way to the proper applications. For example, when you click the mouse, the window server determines which window the mouse is over and dispatches the event to the application that owns the window.

Quartz Extreme is a technology that pairs the hardware of the video card with the functionality of the Core Graphics layer. The Quartz Extreme initiative was originally applied to the window server. The Quartz Extreme compositor takes the window images generated by applications and maps them onto OpenGL textures on the video card. The window server draws upon the power of the GPU to combine the window images on to the display. This saves the main CPU from having to do the alpha blending calculations needed to combine the window images.

The second part of Core Graphics, the Quartz 2D library, will occupy most of this book. Quartz 2D is a high-performance, general purpose library for creating 2D graphics that uses an imaging model very similar to the one used by PostScript and PDF. The library supports a wide variety of output devices and takes advantage of hardware acceleration for improved performance.

Core Video

Core Video, which Apple created after many years of experience with QuickTime, helps applications that want to present motion graphics. In its current implementations, Core Video provides two main services. It handles buffer management and timing services. In presenting movies, a computer typically must decompress the frames and then present those images on screen. In the past, QuickTime would usually complete each of these steps at the same time by decoding the image directly into the display buffer. This limited performance because the computer couldn't decompress an image into the buffer until the

previous frame had been displayed. Quicktime's handling of buffers also made it difficult to support video encoding techniques which require the computer to decode several frames of the animation at once.

Core Video assists an application in managing several frames of animation at once. The library efficiently moves the buffers to the graphics hardware where the computer can display them on-screen. This decouples the decoding and display stages of the animation loop, allowing each to run as quickly as it can. This also allows the graphics hardware to handle the display of frames, freeing the CPU to decode subsequent frames at the same time.

Core Video also handles timing services. In the complex graphics environments available on the Macintosh, it can be difficult to get the timing of animation just right so that it looks as smooth as possible on the display. Core Video runs a high-priority thread on behalf of the application and uses a callback mechanism that allows the operating system to request animation frames from the application. By doing so, the computer can get the frames of the animation in such a way that it can present them on screen at the optimal time. Applications that present animations can use Core Video to their advantage in the performance benefits it offers.

Core Image

Core Image is a filter-based image processing API. The system allows applications to build chains of filters (actually a directed, acyclic graph), combine them, and apply them to an image all in a single step. The kinds of filters found in Core Image are also often found in popular image processing applications such as Adobe Photoshop or The GIMP (GNU Image Manipulation Program). Core Image includes dozens of image processing filters with the installation of Mac OS X. Application developers can provide their own image filters and can even package those filters so that other applications can use them.

As with Core Video, one of the attractive features of Core Image is its ability to take advantage of the graphics hardware. Many of the filters in Core Image are implemented as GPU programs, and the library can run those filters on a programmable graphics card if one is available. As has been said, however, Core Image does not require programmable graphics hardware to run; it can run its filter effects on the main CPU. The code will take advantage of other hardware features like AltiVec or SSE if they are available.

Applications can use Core Image to add a variety of effects and transitions to both user interfaces and application content.

QuickTime

QuickTime is a cross-platform architecture for working with a wide variety of media formats. It began as a system for presenting synchronized sound and video. Over the years, however, the extensible architecture of QuickTime has broadened its scope to include quite a lot more. At its heart, QuickTime is an excellent base for presenting any kind of time-based media. With the proper components, for example, QuickTime could even be used to present the experimental data captured from chemistry experiments.

QuickTime is a bit unusual in the way it touches on so many other aspects of the system. At the lowest level, QuickTime includes components for working with the video, audio, and data storage hardware on the system. At the highest level, QuickTime provides routines and components that present user interfaces and interact with the user—with lots of other functionality in between.

Of particular interest to 2D graphics developers is the fact that QuickTime contains components that make it easy to import images from popular file formats such as JPEG, GIF, and TIFF. QuickTime includes image processing filters and transitions that are somewhat similar to the ones found in Core Image, although the architecture is older and doesn't employ hardware technology as effectively. QuickTime also can transform 2D graphics and perform alpha channel compositing. In many respects QuickTime is a jack of all trades.

If QuickTime suffers from anything, it is a disconcerting dependence on QuickDraw. Apple is carefully freeing QuickTime from this anchor over time. Some of the features that have traditionally been the province of QuickTime are emerging as dedicated systems in Mac OS X. Core Image has already been described, which handles functionality analogous to the effects and transitions components from QuickTime. Core Video supplements QuickTime's video presentation abilities, and Quartz 2D can transform graphics and composite images with alpha channels. Image I/O is a new architecture for importing and exporting images to files. I/O is a dedicated system for importing and exporting images designed to take the place of QuickTime's image import and export components.

Application users that want to play movies and sounds or integrate a variety of different media should consider working with QuickTime. Some of the newer technologies, like Image I/O and Core Image are only available on newer versions of the Mac OS X. Applications that want to provide the same functionality on older systems can use the similar functionality in QuickTime.

Other Graphics Libraries

The libraries just discussed form the core functionality of the Mac OS X graphics system. Mac OS X also contains a number of additional libraries that are worth a brief mention here. For more information on these libraries you can consult Apple's developer documentation.

ATSUI and Cocoa Text

The Apple Type System for Unicode Imaging (ATSUI) and Cocoa Text systems play a vital role in the creation of graphics that contain text. Both ATSUI and the Cocoa Text system are graphics libraries for combining a block of characters, text style information related to those characters, and a region in which the text can be drawn to create a complete image of the text. These libraries are vital to the correct layout and rendering of Unicode text on the Macintosh system. While we will touch upon ATSUI and Cocoa Text in this exploration of Quartz 2D, the richness of these text layout engines would warrant an entire book in their own right.

QuickDraw

The same QuickDraw library that was vital to the continued success of the Macintosh is still available on Mac OS X. QuickDraw includes the same drawing capabilities today that it has in the past. While Apple plans to maintain binary compatibility with applications that use QuickDraw, the library is now deprecated technology. Apple is no longer changing the QuickDraw code, and the library may become unavailable in future versions of the operating system.

This fact has a profound implication for developers who use the Carbon API. Their applications are probably using QuickDraw as their primary graphics library. If these applications are to grow with the system, developers will have to

convert their drawing routines to use Quartz 2D or one of the other libraries in the Mac OS X graphics system.

In classic Macintosh operating systems, QuickDraw played an important role in the windowing system, not only as a graphics library, but also as a collection of routines for dealing with "graphical situations." For example, QuickDraw could tell you if a given point was to be found inside of a particular pixel region. QuickDraw also managed things like the set of displays on the system or the current system cursor. In general terms, QuickDraw provided several of services that were not related to drawing graphics.

In Mac OS X, these same services have migrated to other libraries that are in a better position to run them efficiently. Quartz 2D is, by-in-large, a drawing API and leaves non-drawing tasks to other systems. If you have code that makes use of any of QuickDraw's non-drawing related services, you may need to look to services other than Quartz 2D to find equivalent functionality on Mac OS X.

vImage

The vImage library is part of the Accelerate framework and is another image processing library on the system. While the focus of Core Image is on visual image effects and transitions, vImage contains routines related to more scientific image processing tasks. Here are some of the features found in vImage:

- Conversion of image data between several different pixel types including conversion between planar and interleaved pixel formats.
- General Convolution on image data.
- Apply Fourier Transforms (one and two dimensional, real and complex valued) to a block of data. (The Fourier Transform is a mathematical operation that occurs often in image and signal processing).
- Apply "morphological" operations on data. Morphological operations manipulate images so that items pictured in the image change their shapes. Usually the items in the image will expand, shrink, or take on other aspects of a bitmap called the morphological kernel.
- Perform geometric operations on an image (such as scaling and resampling, rotations, and reflections).

- Generate histograms for the samples in an image. A histogram tells you how many of the pixels in an image are of a given intensity. Histograms play an important role, for example, in some image enhancement code.
- Alpha channel compositing. (Combining translucent and partially translucent images together)

Looking at this list, you may see many operations that are found in other graphics libraries in Mac OS X. Both Quartz 2D and QuickTime, for example, can be used to rotate an image or perform alpha channel compositing. vImage is a low-level library upon which some of those higher-level libraries are built. Because of this, using the routines in the vImage requires a bit more care on the part of the programmer. vImage can take advantage of multiple processors if they are available and use the SIMD units in those processors as well if they are available.

Java 2D

Mac OS X is an excellent platform for Java development. The Java system available on Mac OS X includes an impressive collection of technologies including the Java graphics library, Java 2D. Java 2D shares a large number of features with Quartz 2D. Both include graphics services for line art, text, and images and include support for alpha channel compositing. The two libraries have some abstractions in common, and many of the drawing techniques that are effective in one library have close analogs in the other library. With so many similarities, it should not surprise you to learn that the Java 2D implementation on Mac OS X relies on Quartz 2D for much of its functionality. In spite of the similarities in their feature sets, Quartz 2D and the Java 2D API use very different programming interfaces. Because it is a completely separate API, one that happens to be implemented using Quartz 2D, but which does not directly expose the functionality of that library, Java 2D is not explored in this book. However, there are a number of excellent resources available on the web and in books to help Java programmers get the most out of Java 2D.

This chapter has covered a lot of ground. Understanding where Quartz 2D fits into the graphics system, both historically and architecturally can help you make effective use of the system. Some time has been spent discussing other graphics technologies that may be of interest, and we encourage you to explore them at your leisure. Writing good graphics code on Mac OS X is often a question of finding the services on the system that most closely match your needs. Hopefully this brief discussion will help you find the services that benefit you.