

## Chapter 2

# GRAPHICAL PROCESSING UNITS

### 2.1 Overview

Knowledge of the operations supported by GPUs and how data is processed in GPUs is necessary in order to understand how GPUs can be leveraged for cryptographic processing and protecting data. This chapter provides an overview of GPUs and their APIs. While GPUs allow for significant levels of parallel processing, the capabilities supported for graphics processing do not allow for general purpose computing within a GPU equivalent to that of a CPU. GPU capabilities continue to expand and the APIs are evolving to improve programmers' access to these capabilities, some of which can potentially assist in performing cryptographic operations. For the most recent GPU capabilities, the reader should refer to vendors' GPU specifications.

This chapter is organized as follows: Section 2.2 provides a summary of the general architecture and capabilities of GPUs. The steps a GPU performs on vertices and pixels when creating an image are described. Section 2.3 provides an overview of the types of operations supported by GPUs and the limitations of GPUs when used for general purpose programming. Section 2.4 lists the common APIs available for GPUs and explains why lower level APIs are more suitable for general purpose programming of a GPU compared to higher level languages that are more user friendly. Data can be processed in GPUs as either vertices or as pixels. For cryptographic applications discussed in this book, the data must be represented as pixels. Section 2.5 describes how pixels are processed in a GPU, focusing on the operations relevant to creating ciphers that can execute within a GPU. Section 2.6 discusses why vertex processing is not appropriate for existing cryptographic algorithms. The idea of using GPUs for cryptographic processing arose in part because of the processing power of GPUs and a growing number of other applications experimenting with using

GPUs in place of CPUs. A few examples of other non-graphics applications utilizing GPUs are listed in Section 2.7.

## 2.2 GPU Architecture

GPUs contain their own processors and memory. A GPU is connected to the system by either an AGP, PCI or PCI Express bus. The first programmable GPUs operated as a fixed pipeline. A program compiled on the CPU issued API commands to the GPU for execution. This allowed computationally expensive operations to be performed in the GPU to free up the CPU. When programming a GPU, operations are performed on either vertices or pixels. Vertices are specified as coordinates and are the most basic element for defining any line or object. A pixel is a string of bits interpreted according to a specific format to indicate which bits represent the red (R), green (G), blue (B) and alpha (A) components. A typical configuration uses 32-bit pixels with 8 bits for each of the components. In the past two years, the flexibility in programming GPUs has substantially increased with the addition of programmable vertex and pixel (fragment) units that allow for certain programs to execute on the GPU. The term "pixel processor" will be used throughout this chapter to refer to the pixel unit. Some GPU specifications, articles and graphics books use the term "fragment processor" exclusively while others use the term "pixel processor". Vertex and pixels programs are commonly referred to as vertex and pixel (or fragment) shaders. Graphics programming generally uses vertex processing; whereas, non-graphics applications using GPUs typically require the pixel processor [62]. The basic architecture of a GPU is shown in Figure 2.1. The number of vertex and pixel processors, and how the components handling the operations and memory outside of these processors will vary per graphics card. The main point to obtain from the figure is that the GPU contains a series of vertex processors and pixel processors working in parallel. The vertex data are received over the bus from the host processor, and are processed by the vertex shaders, which include a programmable unit. Some fixed steps are performed, including rasterization, of which the output is the fragments given to the pixel processors. Both programmable and fixed steps are performed during pixel processing. The vertex processing steps are not applicable when operating on pixels directly. Bytes stored in the system's memory can be written directly to the GPU's memory and then processed as pixels. "Fixed" steps refer to steps outside the programmable units. These steps are controlled to some extent by the programmer. For example, the programmer defines stencils and depth, and parameters for the viewing angle and perspective, among others.

Common components found within the vertex processor are a floating point unit, a floating point vector unit, a unit for fetching textures from the GPU's cache and a branch unit. One or more units for vertex assembly operations and viewport (mapping a 3D scene to the 2D viewing area) may be included.

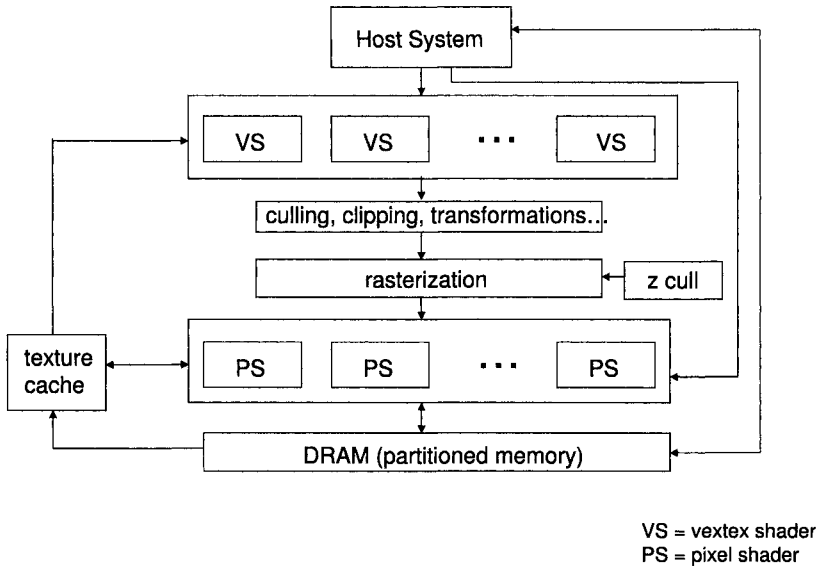


Figure 2.1. High Level View of GPU Hardware

Components found within the pixel processor include a texture processing unit that communicates with the cache, one or more floating point units, a branch unit and a fog arithmetic and logic operations unit (ALU).

The processing speeds of GPUs have been increasing at a rate faster than CPUs. In the last two to three years, GPUs have evolved to contain more transistors than typical desktop CPUs. Although processing speeds of GPUs now surpass those of CPUs, their capabilities are narrower in scope than those of CPUs and do not offer the general programmability provided by the latter. This is due to both API limitations and GPU capabilities. Newer GPUs process at rates exceeding 40 billion floating point operations per second (GFlops). For example, peak performance of a Nvidia GeForce 6800 ultra was listed as 40 GFlops in comparison to 6 GFlops on a Pentium 4 with a 3.2 Ghz processor [57]. The RAM in GPUs is of smaller capacity than that commonly found in systems today, with newer GPUs containing a maximum of 256 MB or 512 MB of RAM compared to the 1 GB to 4 GB of RAM available for typical desktop PCs. However, as the number of transistors per GPU (or CPU) increases, the power consumption and heat dissipation become of greater concern.

Until the year 2005, most GPUs supported 32-bit pixel formats and 32-bit floating point precision while others only supported 16-bit precision. Recently, support for 64-bit pixel formats and 64-bit floating point precision has become more common. Graphics cards with 128-bit floating point precision are becoming available.

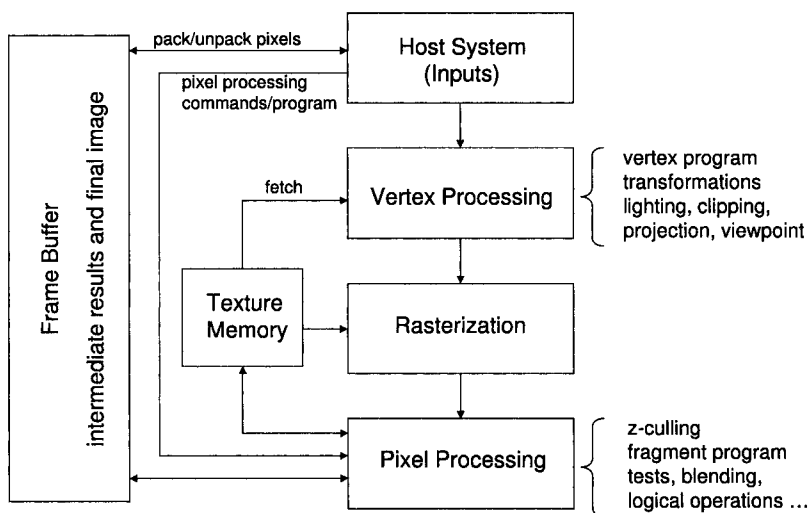


Figure 2.2. GPU's Main Processing Steps

The general flow for processing data in a GPU is shown in Figure 2.2. The general flow in OpenGL 2.0, a platform independent API for GPUs, is shown in Figure 2.3 from the OpenGL Version 2.0 Specification [75]. It is important to understand both the fixed pixel processing pipeline as well as the flow with programmable units because not all operations have been moved into the programmable units. For example, the rasterization and blending steps are outside the programmable units and most of the pixel processing in OpenGL still corresponds to the basic pipeline [39]. The implementation of the block cipher AES described in Chapter 4 uses the basic pixel processing of GPUs.

GPUs can be viewed as processing data in two formats. The first and most used in graphics applications is vertex processing. Vertices are specified as sets of coordinates. Any shape or object is formed by a set of connected vertices. Once objects are defined, transformations concerning properties such as the

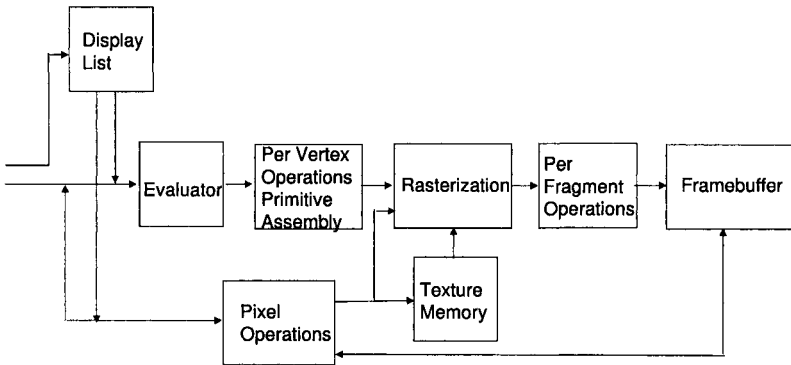


Figure 2.3. OpenGL Version 2.0 General Pipeline

angle and direction the scene is viewed from, lighting and intensity are applied. The resulting scene is converted into fragments (pixels) and undergoes pixel processing before being displayed. The coordinates and properties of vertices, including color and location, cannot be tracked and read back to system memory as data. As a result, vertices are not a suitable means for representing data to which cryptographic operations are applied with the intent of offloading work from the CPU then supplying the result to a process running on the operating system. Even when the intent is to decrypt data in the GPU and display it to the user, with no need to transfer the data back to the operating system, vertex processing cannot be used because of the floating point representation and rounding. The rounding in GPUs, even when considering the increasing precision of 64 and 128-bit floating point values, results in a lack of accuracy unacceptable for ciphers where changing one bit will produce an incorrect decryption.

Processing vertex data involves the following steps (some of these steps may be performed within the vertex program instead of the traditional pipeline):

- The various data needed to construct the image (vertices, including their coordinates and colors, properties, parameters and any textures) are defined. The data is passed into the GPU through API commands.
- Transformations are applied to set the vertices in the scene. The vertex coordinates (including depth) are multiplied by model and view transfor-

mations. The model transformation indicates any rotation, translation or scaling of the scene; for example, rotating about the X axis by 30 degrees and doubling the scale of an object. The view transformation is the angle (or camera position) the scene is viewed from.

- Lighting is applied. This sets the angle and intensity of the light.
- Clipping, projection and viewpoint are applied. Clipping removes areas outside the scene. Projection can be thought of as viewing the scene through a normal, telephoto or wide angle camera lens. It also determines if all objects appear to be of the same size or if objects that are further away are smaller than those at the front of the scene (as objects appear in real life). The viewpoint defines the shape and area of the screen where the objects will appear.
- Rasterization, the converting of vertices into fragments (pixels), is performed. Texture coordinates are interpolated from the texture coordinates of the vertices.
- The fragments resulting from rasterization are tested to determine which pixels to keep and which pixels to discard. The scissor test discards portions of the image outside of a defined region. The alpha test discards pixels based on their alpha values. The stencil test discards portions of the image outside of a defined stencil. The depth test discards pixels based on their depth. When a pixel program is applied, fragments that will fail the depth test are discarded before the pixel program is applied. The depth test and the other tests are applied after the pixel program.
- The pixels are combined with the current contents of the buffer. The default setting is for the new pixels to overwrite the current pixels in the buffer. The pixels may instead be combined in a few ways. The current and new values may be combined by blending. The resulting value of each color component is based on both the new and current values; for example, by multiplying both the old and new value by some factors then adding the result. By default, no blending is performed. Dithering may be applied. This averages pixels with neighboring pixels to eliminate abrupt color changes. By default, dithering occurs, but can be disabled by an API command. Logical operations can also be applied, such as XORing the new and current pixel values together. Logical operations are off by default.

A vertex program can replace the model and view transformations, and any per-vertex lighting. The vertex program may define textures and their coordinates. Vertex programs work on a single vertex at a time, with the output continuing through the remainder of the standard pipeline. Operations that require knowledge of multiple vertices and/or of topology are performed according to

the standard pipeline. A vertex program can read textures but cannot currently read from the framebuffer.

The second method for processing data in GPUs is pixel processing. Individual pixel values can be set and operated on, as opposed to drawing and manipulating objects. Pixels can be used to store and manipulate byte level data, as described later in this chapter. Pixel values can be transferred between the GPU's framebuffer and system memory (where they are stored as bytes) by executing commands from a program running on the CPU. Therefore, an application using a GPU to offload processing from the CPU can read the result from the GPU to use in a program running on the CPU. Pixel processing is described in more detail in Section 2.5. In Figure 2.2, the vertex processing steps are not applicable when dealing solely with pixels. The program executing on the system's CPU will write pixels to the framebuffer and possibly define textures, then the pixel processing steps will be performed.

## 2.3 GPUs and General Purpose Programming

The following is an overview of the types of operations supported by GPUs and the limitations of GPUs when used for general purpose programming. The types of applications best suited for GPUs are those that involve operations that take a single pixel's value, apply a function to it (with limitations on what the function can be) and output the result as a new pixel value. Parallel processing of data is performed by using multiple pixels and multiple color components of a pixel. Four streams of data can be operated on simultaneously in the GPU by using each of the four components of a pixel (red, green, blue and alpha — RGBA). As a general rule, data that is stored in an array when programming in the CPU should be represented as a texture in the GPU. Any loop running in the CPU should have the inside of the loop run as a kernel on the pixel processor. Complex functions that cannot be performed in the GPU can be computed in advance in the CPU and the results stored as tables (*e.g.*, as colormaps) or textures to be used by the GPU in some cases. In order to use table lookups in the GPU to represent a function, the function must only take a single input value and the input value must be able to be stored in a color component of a pixel. If the function takes multiple inputs, it may not be possible to represent it as a table lookup on pixel values. Applications that take multiple inputs and produce a single output; applications that require pixels be processed in a particular order; or applications that require using one pixel's value to determine which operation to perform on another pixel are not suitable for implementation in a GPU. Visiting pixels in a particular order in a single pass through the pixel processor is not possible because there is no way to control the order in which pixels are processed. It is also not possible to use the results from an already processed pixel when operating on a pixel that has yet to be processed.

Pixel processors can currently perform what is referred to as scatter. Scatter is the capability to output results to areas of the image other than those used as the input to the function. However, pixel processors cannot support memory accesses such as  $a[i] = x$  where  $i$  is a computed address. The operation  $x = a[i]$  is possible. If  $a$  is a texture and  $i$  is a computed value, then  $a[i]$  is a texture fetch instruction; whereas,  $a[i] = x$  requires a texture write instruction to a computed address,  $i$ . This is because in pixel processors the only writes allowed are to pre-computed fragment addresses that cannot be changed by a program running in the pixel processor. The GPU is also designed to only read texture data; whereas, a CPU is designed for read and write operations. This limits how data can be processed in a GPU compared to a CPU. A way around this is to write intermediate results to system memory then read the data back into the GPU. This increases the number of data transfers between the GPU and operating system, increasing the overall execution time. Furthermore, it makes intermediate results available to the operating system, which must be avoided for the applications addressed in this book. A second option is to use the vertex processor, which supports such indexing. This is unsuitable for applications whose data cannot be represented and processed as vertices, including cryptographic processing.

Branching is also not readily supported in pixel processors, although a workaround (described by Pharr [62]) can exist for certain applications. The GPU's pixel processor is still often a single instruction, multiple data (SIMD) design without support for branching or with minimal support where both paths of the branch must be taken, slowing processing. In contrast, vertex processors are now multiple instruction, multiple data (MIMD) processors and support branching. Some GPUs, such as Nvidia's Geforce 6 Series, supports different segments of the frame taking different paths. One segment is processed at a time, causing the other to wait. Processing after the branch does not begin until both segments have finished the branch. On Nvidia's FX Series, branching is supported by evaluating both possible paths then only writing the results of the path actually taken.

Another limitation is that GPUs treat all values as floating point values, including the values of pixel components. This must be carefully taken into account when the data being processed involves values that are to be interpreted as individual bits. The floating point representation results in limiting the range of integers supported and rounding error. The OpenGL version of AES in Appendix A illustrates the impact of rounding error by having to consider the impact when populating the tables used in the implementation. While the OpenGL shading language supports an integer data type, this is done for the benefit of the programmer. The values are actually stored and processed as floating point values in the hardware. In the OpenGL shading language, integers are currently limited to 16 bits plus a sign bit.



The time to read and write data to and from the GPU must be considered. The processing power of GPUs has increased faster than the data transfer rates between the system's memory and the GPU. Therefore, it is best to limit reads and writes to system memory. When the time to transfer data between system memory and the GPU is considered, functions that can be computed faster in the GPU may take longer than when computed in the CPU.

In general, functions that have a large ratio of arithmetic operations compared to memory accesses may perform better on a GPU (provided the arithmetic can be done on the GPU) than a CPU. Whereas, those that require a large number of memory accesses will likely be slower on the GPU. Note, neither symmetric key ciphers nor asymmetric key ciphers fall into the category of having a large ratio of arithmetic operations. Symmetric key ciphers have simple operations repeatedly applied to the data. Asymmetric key ciphers have a few arithmetic operations requiring large data structures; for example, exponentiation involving large integers.

## 2.4 APIs

The two most common APIs for GPUs are OpenGL and Direct3D. OpenGL is an open source, platform independent API. In contrast, Direct3D is specific to Microsoft Windows. These APIs are the lowest level, publicly available interfaces to GPUs. There are higher level languages built on top of OpenGL and Direct3D that provide a more user-friendly syntax and hide lower level details from the programmer, but such languages provide no additional capabilities in terms of what commands can be executed in the GPU since they rely on the OpenGL and Direct3D APIs. What can be executed within a GPU is restricted to the capabilities of the GPU, which are independent of the level or type of API used.

The higher level languages result in code that compiles to a combination of a program (usually *C* or *C++* code) that executes in the CPU and issues commands to the GPU. Such languages do not allow the developer control over which commands the code is translated into or even which commands are executed in the GPU. For example, code in a higher level language that XORs two bytes will likely be transformed into code executed in the operating system rather than converted into OpenGL commands that converts the bytes to pixels and XORs pixels. Using pixels to XOR bytes produces the desired result but is an inefficient way to XOR only two bytes when the operation can easily be performed in the CPU. Using the GPU to XOR two long sequences of bytes in a single step is useful. Cg [25], Brook GPU [8] and Vertigo [10] are some examples of higher level languages of which Cg is the oldest and the most well-known of the languages. Cg is a *C*-like syntax that compiles to either OpenGL or Direct3D code, depending on the platform. The Cg code must be included in a main program that compiles on the CPU, such as a *C* or *C++*

of the window system and providing a more user-friendly syntax for creating display windows than the APIs for the window systems. GLUT is closed source. Its executable is available from the OpenGL organization at: <http://www.opengl.org/resources/libraries/glut.html> There are several alternatives to GLUT, including open source versions such as Freeglut. A list of toolkits that provide wrappers for window systems' APIs along with links to their downloads are available at: <http://www.opengl.org/resources/libraries/windowtoolkits.html>. The experiments described within this book required using a low level API in order to issue commands directly to the GPU and required platform independence. Therefore, OpenGL was used in all experiments. GLUT was used to open the display windows. Further details regarding OpenGL pixel processing and vertex processing that are relevant to implementing ciphers within GPUs are provided in the next two sections.

At the time this was being written, ATI announced plans to provide support for general purpose GPU programming by publishing an API that is at a lower level than OpenGL and Direct3D in order to provide more direct access to GPU's capabilities [59].

## 2.5 OpenGL and Pixel Processing

The following is an overview of the OpenGL pixel processing pipeline and the OpenGL commands relevant to the experiments described in subsequent chapters. The implementations used in the experiments process data as 32-bit pixels treated as floating point values, with one byte of data stored in each pixel component. When using 32 bit pixels, 1 byte is typically dedicated to each of the RGBA components. Other formats, such as 10 bits for each of the red, green and blue components and 2 bits for the alpha component may also be supported. Since the time of the experiments, support for 64-bit pixels with 16 bits for each of the color components has become available. The following capabilities are not used in the experiments described in this book and therefore, are not described here: OpenGL's capabilities of processing pixels as color and stencil indices, and OpenGL's vertex processing (refer to [58] and [89] for a complete description). Figure 2.4 shows the components of the OpenGL pipeline that are relevant to pixel processing when pixels are treated as floating point values. While implementations are not required to adhere to the pipeline, it serves as a general guideline for how data is processed. The programmable pixel processor replaces part of the pipeline. Pixel shaders can access and apply textures, compute and set colors and depth, and apply fog.

As with vertex shaders, the various tests at the end of the pipeline (scissor, stencil, alpha, *etc.*) are performed according to the pipeline and are not programmed within the pixel processor. OpenGL requires support for at least a front buffer (image is visible) and a back buffer (image is not visible) but does not require support for the alpha pixel component in the back buffer. This limits

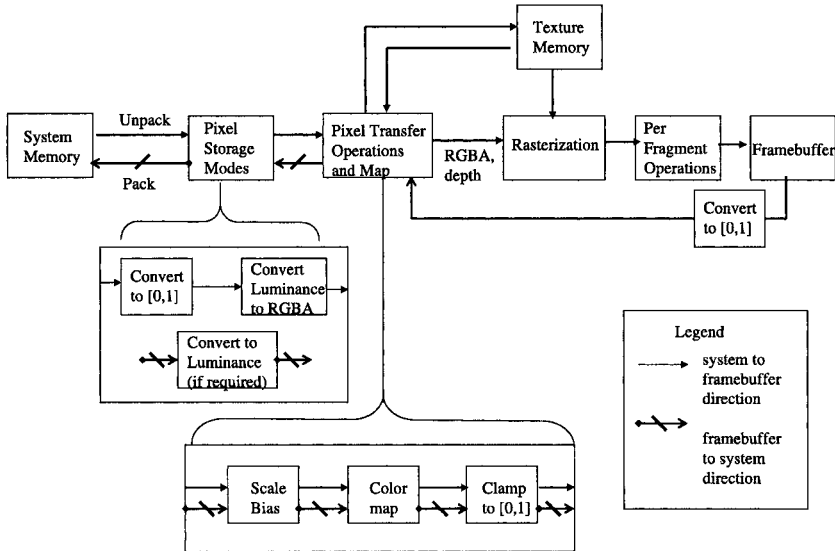


Figure 2.4. OpenGL Pipeline for Pixel Processing

the data representation to be three bytes per pixel (the red, green, blue components) when performing operations in the back buffer. It is worth mentioning that while a 32-bit pixel format is used in the implementations described in Chapters 4 and 5, the 32 bits cannot be operated on as a single 32-bit value, but rather is interpreted in terms of pixel components. For example, it is not possible to add or multiply two 32-bit integers by representing them as 32-bit pixels.

A data format indicating such items as number of bits per pixel and the ordering of color components specifies how the GPU interprets and packs/unpacks the bits when reading data to and from system memory. The data format may indicate that the pixels are to be treated as floating point numbers, color indices, or stencil indices. The following description concerns the floating point interpretation. When reading data from system memory, the data is unpacked and converted into floating point values in the range  $[0, 1]$ . Luminance, scaling and bias are applied per color component. The next step is to apply the colormap, which we describe later in more detail. The values of the color components are then clamped to be within the range  $[0, 1]$ .

Rasterization is the conversion of data into fragments, with each fragment corresponding to a pixel in the framebuffer. In work described here, this step

has no impact. The fragment operations relevant to pixel processing include dithering, threshold-based tests (such as discarding pixels based on the alpha value and on stencils), and blending and logical operations. These operations combine pixels being drawn into the frame buffer with those already in the destination area of the frame buffer. Dithering, which is enabled by default, must be turned off when storing data in pixels to prevent pixels from being averaged with their neighbors and their values changed as a result.

When reading data from the framebuffer to system memory, the pixel values are mapped to the range  $[0, 1]$ . Scaling, bias, and colormaps are applied to each of the RGBA components and the result clamped to the range  $[0, 1]$ . The components or luminance are then packed into system memory according to the format specified. When copying pixels between areas of the framebuffer, the processing occurs as if the pixels were being read back to system memory, except that the data is written to the new location in the framebuffer according to the format specified for reading pixels from system memory to the GPU.

Aside from reading the input from system memory and writing the result to system memory, the OpenGL commands in the implementations described in Chapters 4 and 5 consist of copying pixels between coordinates, with colormaps and a logical operation of XOR enabled or disabled as needed. Unfortunately, the copying of pixels and colormaps are two of the slowest operations to perform [89]. The logical operation of XOR produces a bitwise-XOR between the pixel being copied and the pixel currently in the destination of the copy, with the result being written to the destination of the copy.

A colormap is applied to a particular component of a pixel when the pixel is copied from one coordinate to another. A colormap can be enabled individually for each of the RGBA components. The colormap is a static table of floating point numbers between 0 and 1. Internal to the GPU, the value of the pixel component being mapped is converted to an integer value that is used as the index into the table and the pixel component is replaced with the value from the table.

For example, if the table consists of 256 entries, as in the AES implementation described in Chapter 4, and the map is being applied to the red component of a 32-bit pixel with 8 bits per color component, the 8 bits of the red value are treated as an integer between 0 and 255, and the red value updated with the corresponding entry from the table. In order to implement the tables used in the OpenGL version of AES as colormaps, the tables must be converted to tables of floating point numbers between 0 and 1, and hard-coded in the program as constants. The table entries, which would vary from 0 to 255 if the bytes were in integer format, are converted to floating point values by dividing by 255. Because pixels are stored as floating point numbers and the values are truncated when they are converted to integers to index into a colormap, 0.000001 is added to the result (except to 0 and 1) to prevent errors due to truncation.

The use of floating point numbers for pixels is one example of where GPUs do not readily provide a capability required for cryptographic processing, namely the need to maintain the accuracy of byte values with no change due to rounding or truncation.

There is no support for user specified conditional statements based on specific pixel values. For example, there is no way to say

```
pixel = pixel at x,y coordinates 80,90
if (pixel's blue value == 01010101) {
    turn on colormap 1;
}
else {
    turn on colormap 2;
}
```

and have this code executed entirely in the GPU. The pixel's value has to be read to the operating system's memory, the comparison performed using the CPU and system memory, then a command issued to the GPU to turn on the appropriate colormap.

## 2.6 Representing Data with Vertices

In contrast to pixels, vertices cannot be used to store data. Intuitively, if the data being operated on is used to define vertex coordinates and the operations performed on the data result in altering the coordinates of vertices, the resulting coordinates of the vertices represent the outcome of the computation. However, there is no support for tracking individual vertices and reading their coordinates back to system memory. A vertex is defined by its coordinates, thus these must be known. A vertex cannot be referenced by some other name and the coordinates obtained as if they were a property of a vertex object. Another obstacle is that there is no means by which to define conditional statements based on the coordinates of a vertex.

It may be thought that encryption of data can be performed by defining a vertex for a segment of data, such as a bit or byte, with the color of the vertex representing the data. Then a vertex program performs a series of transformations that alter each vertex's color and possibly its location, with the new color (which is just a pixel value) for each vertex representing the encrypted data. Decryption will require reversing the transformation or repeating the transformation in the case of a stream cipher. However, existing ciphers cannot be mapped to such a transformation. Rounding during the transformation (recall that all values are of floating point type) will result in a lack of precision. Just consider the vertex coordinates. The transformation must insure that calculations never result in a coordinate value that falls between two pixels to avoid indeterministic results.

For example, a vertex with an  $x$  coordinate computed to be 100.499 by one GPU and 100.500 by another will impact the pixel with coordinate 100 in the first display and 101 in the second display if the GPUs round to the nearest coordinate. The values 100.999 and 101.000 will map to  $x$  coordinates of 100 and 101, respectively if the GPUs truncate the floating point values when determining the coordinates.

In summary, a program cannot create an image defined by vertices at specific locations, perform transformations on the image that alters the locations of the vertices, then read properties of a particular vertex because the coordinates of that vertex are no longer known. Specifically, if a program creates a vertex  $v$  at coordinates  $(x_1, y_1, z_1)$ , there is no general way to track the movement of  $v$  and at the end of the program read  $v$ 's new coordinates,  $(x_2, y_2, z_2)$ , or color. (A vertex can be tracked when using only simple transformations, such as rotating a square 90 degrees.)

## 2.7 Non-Graphic Uses of GPUs

As the processing capabilities of GPUs increase, the idea of using GPUs for non-graphic applications is becoming common. The General Purpose Computation on GPUs (GPGPU) organization catalogs experiments that use GPUs for general purpose computing. The GPGPU website is located at <http://www.gpgpu.org/>. Nvidia's *GPUGems 2* [62] devoted a section to general purpose computing using GPUs. Applications range from natural extensions of graphics programming that use graphical simulations for physical processes, such as particle flows, and visualizations, such as 3D representations of protein structures, to scientific computing and basic mathematical algorithms. Scientific computing applications include parallelizable algorithms used in genetic research [92] and options pricing [62]. Standard mathematical problems include solving systems of linear equations [26], fast Fourier transforms, the Floyd-Warshall algorithm, and sorting [62].

When the purpose of using the GPU is to obtain faster processing as opposed to isolating data from programs running on the operating system, applications can split processing between the CPU, system memory and the GPU to work around GPU limitations. How the work is split between the system's CPU and memory, and the GPU must be considered when using a combination of the system's and GPU's resources. The number of data transfers between the GPU's and system's memory, and how often the GPU and/or CPU must wait for the other to complete an operation before proceeding can negate any benefit the GPU provides for the steps it performs. A loop that runs in the CPU and maintains a counter or performs conditional tests, but whose main body executes in the GPU is an example of how to split work when conditional statements are needed that cannot be performed in the GPU. However, the conditional tests must at most involve transferring a small number of data from

the GPU to the system's memory. An example of this is one of the sorting programs mentioned in [62]. In contrast, a program that performs most of a loop within the GPU but needs to transfer all data back to the system's memory in order to perform part of the loop may result in the data transfers decreasing or eliminating the performance gained from using the GPU. For cryptographic processing, it is acceptable if operations that do not involve any intermediate results are performed on the CPU. For example, when implementing a block cipher with  $r$  rounds, the loop maintaining the round counter can run on the CPU while the body of the loop, which is the round function, executes in the GPU.

CryptoGraphics

Exploiting Graphics Cards For Security

Cook, D.; Keromytis, A.D.

2006, XVI, 140 p. 20 illus., Hardcover

ISBN: 978-0-387-29015-7