A benchmark of graphic APIs for use in GIS rendering

Abstract

In this paper we will look at the rendering performance of several Graphics Application Programming Interfaces (API's) with a focus on geographic map rendering. A mixture of 2D and 3D engines will be benchmarked. We will focus on two aspects: Rendering speed of 2D geographic based data and the ease of implementing a rendering system based on each of the API's. From this data we discern which API is best suited for building a c# open source Geographic Information System (GIS) rendering engine. The results of the implemented benchmarks are discussed in further detail and the best API for implementing a GIS rendering system is identified.

Keywords - Header Times 12pt bold

Architectures and frameworks for open source software and data, Benchmarking, Graphics, GDI, GDI+, OpenGL, DirectX, Direct2D

1. INTRODUCTION

What graphics API should be employed in order to build a rendering engine for a GIS? How does one go about finding the best API? Renhart (2009) states that a requirement for creating a fast good quality mapping application starts at the selection of the graphics rendering API. This is largely due to the fact that performance tuning a mapping application is dependent on the drawing speed which is directly related to the graphics API used. It is thus important to select an API with good performance.

Computer graphics are used everywhere today and as a result there are a number of API's available. Each API has its own set of advantages and disadvantages. Some are easier to use at the cost of speed. Others provide a lower level of abstraction with a higher speed but with the added overhead of larger implementation costs. The APIs can be broken down into two categories. High level and low level. High level APIs allow for faster development times due to their higher abstraction. They are easier to use as most of the lower level functionality has already been abstracted into higher level functions. They often provide tools for scene management which has the benefit of not having to be

built from scratch. High level libraries may not provide the flexibility required to do specialized rendering. Low level API's provide a lot more flexibility and customizability. They provide little or no additional tools besides the rendering engine. The large overhead associated with building and application on top of a low level API makes them a less attractive option.

Rendering speed is important for a GIS. Gahegan (1999) asserts this importance from the highly interactive nature of exploring geographical information and datasets. Users require a mechanism to move around and through the data in an immersive, virtual and dynamic way. It is therefore important to choose an API that will deliver the required performance. This will result in a better understanding and interpretation of the data sets due to higher interaction and fluidity within the system and provide a better overall user experience. There are two criteria of measurement that are important. They directly relate to the speed or performance of the API coupled with the relative quality of the rendering. In other words, the objective is to generate the best image in the fastest time.

The easiest way to discern the most appropriate API is to use them to perform a series of benchmark tests. The tests should focus on the functionality that will most likely be implemented when creating a GIS rendering system. At the very least a GIS should be able to adequately render a point, line and polygon layer. Various rendering libraries can be tested to perform the rendering. The performance of each of the rendering functions utilized can then be compared and evaluated. One must however be careful when interpreting the results of benchmarks. It is easy to get a skewed view of what the results actually mean. The process of benchmarking the various APIs will be discussed in further detail later in this paper.

The paper is structured as follows: Section 2 discusses related work. Section 3 discusses the process of computer benchmarking. Section 4 gives an overview of the various API's that were benchmarked. Section 5 discusses the methodology in performing the benchmarks. Section 6 gives a short discussion on the results of each API. In section 7 we draw some conclusions.

2. BACKGROUND AND RELATED WORK

The International Organization for Standardization (ISO 19101:2002, 4.16), define a GIS as information concerning phenomena implicitly or explicitly associated with a location relative to the earth.

GIS data is linked to or represents real world spatial objects. The visual aspect of GIS is a powerful tool. The human mind coupled with a computer generated

"picture" of data is what is referred to as visualization. Tory & Moller (2004) define visualization as "a graphical representation of data or concepts which is either an internal construct of the mind or an external artifact supporting decision making. In other words visualizations assist humans with data analysis by representing data visually."

The visual aspect of GIS data means that large amounts of data can be quickly and easily interpreted by a person. The adage "A picture is worth a thousand words" - Fred R. Barnard, is well substantiated when applied to the realm of GIS data.

There are, however, several problems associated with the rendering of the large amounts of data comprising most GIS systems today. Gahegan (1999) explores several barriers that need to be overcome in order to successfully and adequately render geographic information. The first of these barriers are graphic in nature. They are the speed at which a scene can be rendered coupled with the combination effects that can be employed in order to discern relationships and trends between different datasets in a visual manner. The fact that most GIS systems comprise such large amounts of data mean that quick rendering is essential. The combinational compounding effects of the amount of data coupled with the complexity of the effects utilized to render the data will influence the total speed at which a map can be drawn. In an effort to speed up the rendering of data, generalizations may be applied to the geographic features which are representations of real world objects. Basaraner (2002) defines the process of generalization in a GIS environment as deriving purpose oriented lower detailed datasets at smaller scales or lower resolution from detailed data sources or a dataset at larger scale or higher resolution. GIS models inherently already represent spatial features in a generalized manner. Further generalization allows for the volumes of data to be reduced at the cost of data accuracy. One of the outstanding research questions to be answered is: What is the most effective visualization platform to be used when creating a GIS rendering system, Gahegan (1999).

The goals of this paper are to write some basic graphic rendering functions that utilize the various API's in order to determine the relative performance of the rendering. Other factors such as ease of use of an API will also be noted as it plays an important role for the development and maintenance of a software product.

In terms of benchmarking, Renhart (2009) has conducted research on various mobile graphics API's with the goal of deciding which of the available ones will be best suited for implementing a GIS mapping system. It is important to note however that the criteria for a mobile phone application and the criteria for a desktop application are quite different. Mobile devices have a lot more

restrictions imposed on them in terms of memory, performance, screen size and storage. Renhart (2009) accomplished the research by measuring the performance times of graphic operations and comparing them to other libraries. The metrics are simple. The API with the fastest overall time across different rendering functions will be the best API to use for building a GIS rendering system.

3. MORE ON COMPUTER BENCHMARKS

Zhang (2001) defines a benchmark as a set of programs that are run on different systems to give a measure of their performance. A benchmark is useful for measuring the relative performance of a system or aspects thereof which can then be compared to other existing systems. Care must be taken when designing a benchmark to ensure that one is measuring what is actually of value. Focusing on a single dimension like computational performance may not yield an accurate depiction of how the system will perform in a real world environment. During the implementation phase of benchmarks, special care needs to be given to ensure that the graphics API being benchmarked receives adequate volumes of data. The bottleneck is almost always getting the data from disk. We will later discuss a simple way to enable the fast delivery of data to the graphics API.

Benchmarks can be broadly subdivided into two categories. Zhang (2001) summarizes these as micro benchmarks and macro benchmarks. A micro benchmark tests the performance of a function on the lowest level. An example of this is the time it takes to draw a simple primitive on the screen. The advantages of this type of benchmark are that one gets a very good idea of the fundamental cost of a function. On the downside, it may be difficult to translate the actual measurements into values that will be equivalent to the cumulative result of the system in its entirety.

A macro benchmark consists of a larger inclusive set of functionality and more accurately measures the performance of a system as a whole. It is a much more accurate and practical representation of the actual performance that will be achievable by an application. The downside to this approach is the cost and time associated with the implementation of such a test suite. Care will also have to be taken to ensure the implementation does not include unintentional bottlenecks.
Benchmarks may measure various values like memory utilization and processing speed. Depending on where the focus is, some criteria may be given a higher importance. The focus in this paper is rendering speed.

4 GRAPHICS API'S

A graphics API in this context is the library of code that sits between the application and the graphics hardware performing the rendering. Not all API's utilize hardware acceleration via the Graphics Processing Unit (GPU). These libraries are executed on the Central Processing Unit (CPU) and are called software rendered API's. Software rendering is generally orders of magnitude slower than their GPU counterparts.

4.1 GDI API

Walbourn (2009) notes that the primary graphics API since early days has been that of Graphics Device Interface (GDI). This holds true even for many of the latest GIS mapping applications today. It is still employed as the primary API for doing graphics in Windows. This is a trend that will likely continue for some time still.

GDI was developed to keep the application programmer agnostic of the underlying details associated with a particular display device, Richard (2002). It acts as middleware between the programmer and hardware that facilitates the final rendering. Four types of primitives are supported by GDI: lines, curves, filled areas, bitmaps and text.

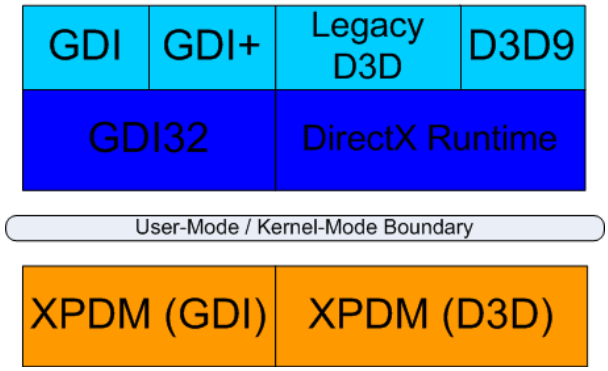


Figure 1: Walbourn (2009). Graphic Outlay of WindowsXP

The above figure serves to illustrate the graphics API's layout for the Windows XP operating system. The Windows XP Display Driver Model (XPDM) is divided into two sections. One that runs the GDI implementation which is not hardware accelerated, i.e. GDI performs all rendering via the CPU. The other section is the Direct3D section which utilizes hardware rendering. GDI's lack of

hardware rendering under Windows XP was a big disadvantage. Most computers today have powerful graphic hardware on board which, properly utilized, would bring major speed advantages.

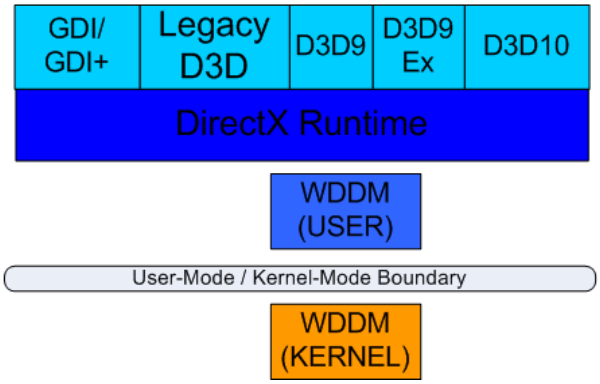


Figure 2: Walbourn (2009). Graphic Outlay of Windows Vista and Windows 7

The above figure shows how the API's were reshuffled in Windows Vista and up. A new driver model, the Windows Vista Display Driver Model (WDDM), brings GPU and Direct3D to the forefront. This allows for some of the previous GDI/GDI+ calls that used software rendering to be hardware accelerated should a graphics card be available and present.

4.2 GDI+ API

GDI+ is the revised version of GDI and its successor. It expands on and provides new capabilities to GDI adding additional flexibility to the programming model. It is not built on top of GDI but exists side by side on the same level (See Figure 1 and Figure 2 above). This library provides functionality for imaging, two-dimensional vector graphics and typography. GDI+ can be used in conjunction with GDI if so desired.

There are several open sources mapping API's available today that utilize GDI+ as their rendering engine. Examples are SharpMap (http://sharpmap.codeplex.com/), MapWindow (http://www.mapwindow.org/index.php) and DotSpatial (http://dotspatial.codeplex.com). This is by no means an exhaustive list but merely serves as proof of the widespread use of GDI/GDI+ for GIS systems.

4.3 DIRECTX API

DirectX is Windows's premier game programming API, Jones (2004). It consists of two layers. The first is the API layer and the second the hardware abstraction layer (HAL). The HAL links the API functions with the underlying hardware and is usually implemented by the graphic hardware manufacturer. The DirectX API sends commands to the graphic card via the HAL. The API itself is based on the component object model (COM). Jones (2004), states that the DirectX COM objects consist of a collection of interfaces exposing methods which are usable by developers to access the graphics API. The COM objects themselves usually consist of DLL files that have been registered with the system.

4.4 OPENGL

The Khronos Group (2012) claim OpenGL to be the premier environment for developing portable, interactive 2D and 3D graphic applications. The OpenGL platform is designed to allow vendors to easily implement their own extensions and so allow for their own spin on implementing high end graphic functions. OpenGL incorporates a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. One big advantage to using OpenGL is that it is supported on a wide range of operating systems and software systems making it very portable. The industry tends to prefer OpenGL for doing application type graphics such as CAD applications whereas DirectX is preferred for creating games (Luten, 2007). DirectX and OpenGL are two directly competing API's. The full implementation specification for OpenGL is available on its website (www.opengl.org) should it be required.

4.5 DIRECT 2D API

Microsoft (2012) has introduced Direct2D as a new API for Windows 7. It is a hardware accelerated, immediate mode 2D graphics API that provides high performance and high quality rendering. Immediate mode means that the API does not cache any of the objects sent to it for rendering. For each frame that needs to be rendered the API has to be resent all the data. This API has been primarily designed for developers to give them a viable replacement to GDI/GDI+.

5. METHODOLOGY

In order to determine the fastest API a simple rendering system was implemented in C# utilizing each of the listed API's. A real world point, line and polygon layer was rendered and the performance times of each of the feature types were logged. The point layer consisted of a collection of points of interest covering most of South Africa. The line layer contained spatial features for a large part of the South African road network. The polygon layer consisted of polygons denoting property stands across South Africa. Table 1 gives a short overview of the composition of the test data utilized.

	Point Layer	**Line Layer**	**Polygon Layer**
Feature count	249313	900000	900000
Total Points	249313	10158849	9679727
Size of Database	1.04 GB		

Table 1: Spatial Data Statistics

Each API is required to render points, lines, polygons and text from the supplied data. The spatial reference system of the test data is WGS84. The data was not re-projected for display on screen. Due to this fact a bit of distortion occurs when rendering the data. Figure 3, which denotes the output of the points of interest layer, shows how the above-mentioned distortion manifests itself. The image seems stretched in the horizontal axis. The distortion is cause by the fact that WGS84 is a geographic coordinate system. It uses a three-dimensional spherical surface to define locations on earth. A computer screen is inherently a 2D object so re-projection is required in order to correctly display the data. In order to project the data a mathematical equation is applied to transform each point. Unless explicitly performed in a GPU shader program the computation is performed on the CPU. A shader is a small piece of code written specifically for execution on a GPU. As the focus is on the graphic rendering speed, coordinate re-projection was ignored. Each API is still required to render and process each point so the projected state of the data will not influence the rendering speed of the API. Of the benchmarked API's only OpenGL and DirectX support the use of a custom shader program. The other API's will have to use a CPU based function to perform the re-projection.

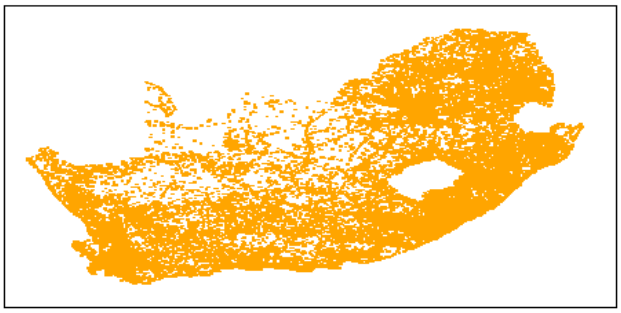


Figure 3: Distortion due to displaying geographic coordinates un-projected on a 2D surface

It is important to be able to serve up data faster than what the graphic engine can utilize it as we have previously touched upon. While working with the datasets it was immediately evident that the first bottleneck would be disk input and output (IO). A number of experiments were conducted in order to determine the fastest way to serve up the data from the storage medium.

The first experiment read directly from a shape file, which is Esri's geospatial vector format for storing data[1]. The binary reader proved to be the bottleneck in this case and the performance was not adequate.

The second experiment involved the loading of the data into a SpatiaLite[2] database. .NET's ActiveX Data Objects (ADO) data provider was used to load the data into the application. Performance was a lot better out of the database but was still not sufficient.

A third experiment involved removing disk IO from the equation by performing the first and second experiments again but with a single difference. A RAM disk was created and used as the storage medium. A RAM disk allows a partition of memory to be mounted and then accessed and utilized like a normal hard disk partition. The speed increase is significant. The results of the IO RAM disk benchmark vs. the hard disk can be seen below here in Figure 4. RAM is so much faster than a HDD that it is barely visible on the graph. The bottleneck was found to be the shape file and database driver so alternative methods were explored.

[1] Esri Geoportal Server is a free open source product that enables discovery and use of geospatial resources.

[2] SpatiaLite is a spatial extension to the SQLite relational database management system. It provides vector geodatabase functionality.

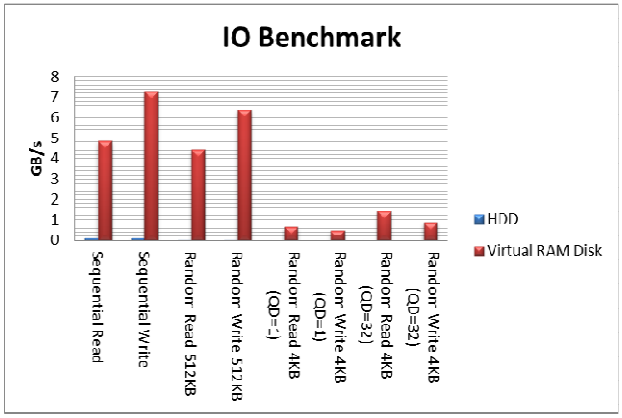


Figure 4: IO Benchmark

In the end the best method proved to be the caching of the data directly in main memory in the form of a dictionary object. This provides the fastest access to the data. The total time averaged across three test machines to loop through the data and convert it to floats:

- Point Layer : 20.3ms
- Line Layer :1690.7ms
- Polygon Layer :881.8ms

The above-mentioned values represent the fastest theoretical rendering time if the graphics API could render instantaneously. It is simply a measurement of the time it takes to loop through each of the features contained in the test datasets. The benchmarking of each API was done by feeding it the point, line and polygon data and drawing the appropriate primitive on screen. Additionally the point layer was used as a location to repetitively draw the same piece of text. The time for the rendering of each of the mentioned primitives was then logged. Each test was run ten times and an average was calculated and then displayed on graphs. The test application was run on three machines. The specifications of each of the machines are noted in table 2 below.

	PC1	Laptop1	Laptop2
CPU	Inter® Core™ i7-2600 CPU @ 3.4 GHz 3.4 GHz	Intel Core 2 Duo T7250 2.00 GHz	Intel® Core™ i7-2860Qm CPU @ 2.5 GHz 2.5Ghz
RAM	8.00 GB	4.00 GB	16 GB
GPU	NVIDIA GeForce GTX 560 Ti	Intel Display with Mobile Intel 965	NVIDIA Quadro 1000M

		Express Chipset	
OS	Windows 7 64-bit	Windows 7 64-bit	Windows 7 64-bit

Table 2: Hardware specifications of benchmark PC's

Two machines have relatively decent graphic cards available with the other having a standard Intel display card.
Ants Performance Profiler was run on each of the implanted rendering functions to determine the function in code where most of the processing time was spent.
The reason for this was to determine if the graphics API was being used to its full potential.

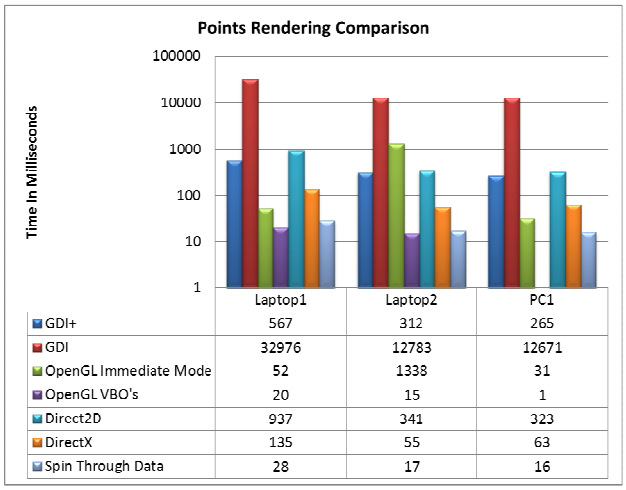
6. RESULTS

Each of the above mentioned graphics API's were tested. We will now discuss each of the API's in more detail. We will take a look at the method used to implement the drawing of each of the primitives on a per API basis. We will also mention where the bulk of the processing time was spent.

In terms of the test computers in order of diminishing performance we have PC1, Laptop2 and lastly Laptop1. The benchmarks focus on immediate mode rendering only. This was to try to eliminate differences between API's. Not all of the libraries allow for more advanced drawing methods. Immediate mode rendering is the common denominator across the benchmarked API's.

A vertex buffer object benchmark was performed on OpenGL merely to highlight what hardware optimizations could bring to the table. It serves to give an idea of what is possible to achieve in terms of rendering performance. This will later be discussed in more detail. Below follows the results of the API benchmarks. Take note however that the OpenGL vertex buffer benchmark does not include the spin times. This can be added to the first run time if a comparable value is required. It was omitted due to the fact that the vertex buffer is only setup once during initialization and then remains in the video card's memory. This initialization was done on application startup so there was no perceived performance penalty.

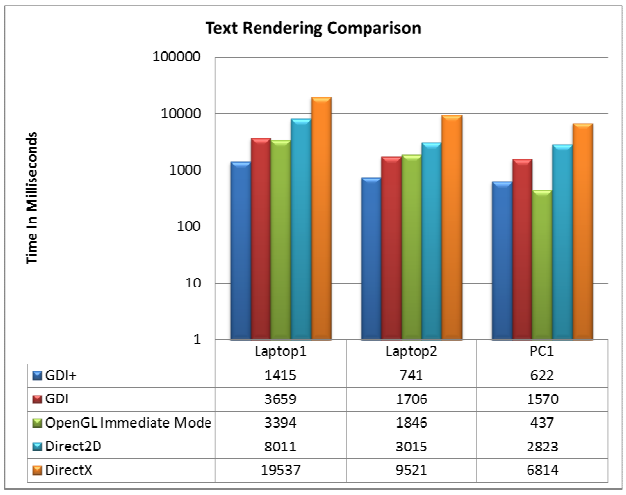
Figures 5 to 8 show the average rendering time across ten runs for each API. Also visible in each graph is the results of each API grouped by the computer it was executed on. This was in order to discern if having a more powerful GPU would yield faster rendering time when compared to the non-hardware accelerated libraries.



	Laptop1	Laptop2	PC1
GDI+	567	312	265
GDI	32976	12783	12671
OpenGL Immediate Mode	52	1338	31
OpenGL VBO's	20	15	1
Dircct2D	937	341	323
DirectX	135	55	63
Spin Through Data	28	17	16

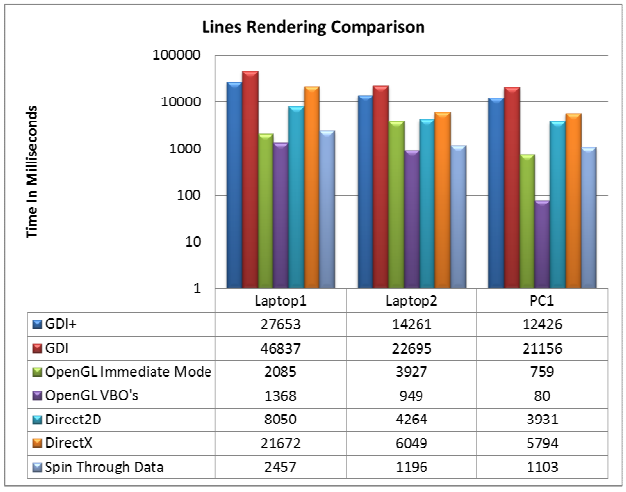
Figure 5. Points Rendering Comparison

We will now discuss in more detail the benchmark of each of the previously mentioned API's.



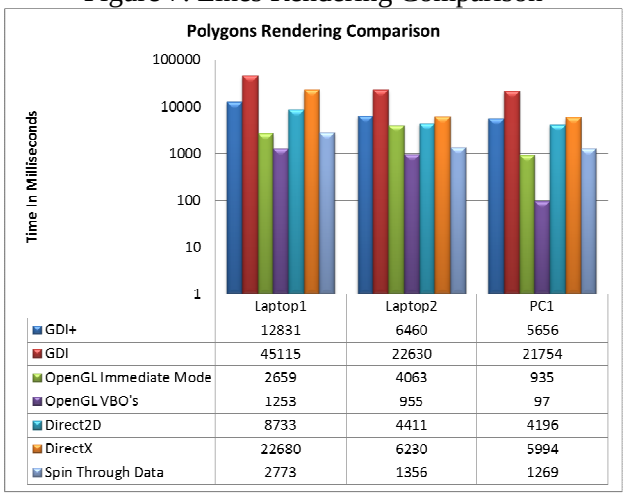
	Laptop1	Laptop2	PC1
GDI+	1415	741	622
GDI	3659	1706	1570
OpenGL Immediate Mode	3394	1846	437
Direct2D	8011	3015	2823
DirectX	19537	9521	6814

Figure 6. Text Rendering Comparison



	Laptop1	Laptop2	PC1
GDI+	27653	14261	12426
GDI	46837	22695	21156
OpenGL Immediate Mode	2085	3927	759
OpenGL VBO's	1368	949	80
Dircct2D	8050	4264	3931
DirectX	21672	6049	5794
Spin Through Data	2457	1196	1103

Figure 7. Lines Rendering Comparison



	Laptop1	Laptop2	PC1
GDI+	12831	6460	5656
GDI	45115	22630	21754
OpenGL Immediate Mode	2659	4063	935
OpenGL VBO's	1253	955	97
Direct2D	8733	4411	4196
DirectX	22680	6230	5994
Spin Through Data	2773	1356	1269

Figure 8. Polygons Rendering Comparison

6.2 GDI API

GDI is the old drawing API utilized by Windows. It has been replaced by GDI+ but as it is still in active use on the Windows operating system it is still applicable. It is not directly available for use in C#. Platform Invoke (P/Invoke) calls were utilized in order to make use of the GDI32.dll library drawing functions. P/Invoke is a feature of the Microsoft Common Language infrastructure implementation allowing managed code to call native code. It is not

an intuitive way to utilize a library as the method signatures are not always that well documented. It does however work very well. Using GDI itself is not difficult and works very similarly to GDI+. The only caveat is that one needs to make sure one correctly disposes of variables or memory leaks will result.
Drawing of the point, line, polygon and text was accomplished using the following methods:
Points : FillRect() – 89% of all rendering time.
Lines : PolyDraw() – 10% of all rendering time. The rest of the time was spent on coordinate transformations.
Polygon: PolyDraw() – 10% of all drawing time. The rest of the time was spent on coordinate transformations.
Text : TextOut() – 40% of all rendering time. The rest of the time was spent on coordinate transformations.
Coordinate transformations were manually handled as the library does not provide built in matrix functions to help with the coordinate transformations. GDI expects all coordinates to be specified in terms of screen coordinates. Most of the rendering time was spent translating the points to the correct locations on the screen. GDI rendering scored the lowest out of all the benchmarks performed. The only exception to this was text rendering which performed in the mid ranges compared to the other libraries. The time to translate the coordinate system to screen coordinates has been included in all the benchmarks as it is a vital and necessary function that will have to be performed by a graphic rendering system.
To clarify translation should not be confused with re-projection. Translation here means the conversion of the arbitrarily defined world coordinate system to the screen coordinate system.

6.2 GDI+ API

GDI+ is contained in the System.Drawing library which is one of the libraries available to .Net. The GDI+ API proved easy to use. The library uses a graphics object which encapsulates a drawing surface. It contains methods for drawing lines, rectangles, paths and other primitives. The library does not have a point primitive. The recommended way to draw a point is via the fill rectangle function.

Drawing of the point, line, polygon and text was accomplished using the following methods:
Points : Graphics.FillRectangle() – 75% of all rendering time.
Lines : Graphics.DrawLines() – 98% of all rendering time.
Polygon: Graphics.DrawPolygon() – 92% of all drawing time.
Text : Graphics.DrawString() – 91% of all rendering time.

Coordinate transformations were accomplished via a single matrix. The Graphics class has a property to allow the setting of a translation matrix which is then applied to all points sent to the API. Overall the performance was not bad. On the two laptops GDI+ had the fastest text rendering times of all the API's. The PC having a good graphics card managed to outperform GDI+ slightly via OpenGL. GDI+ outperformed Direct2D and GDI in terms of points rendering. In terms of rendering lines and polygon GDI+ ended up second last.

6.3 DIRECTX API

DirectX was utilized through the Microsoft.DirectX and Microsoft.DirectX.Direct3D libraries. In order to reference these libraries it is necessary to install the DirectX software development kit. The June 2010 version of this library was used. This library was rather difficult to use. Implementing the benchmark on this library took significantly longer than the other libraries. The library has methods to draw points, lines and triangles. Other primitives need to be constructed using these basic primitives.
Drawing of the point, line, polygon and text was accomplished using the following methods:
Points : Device.DrawUserPrimitives(PrimitiveType.PointList) – 4% of all rendering time. The rest of the time was spent building the arrays of structs which contain the required data points to be passed to DirectX for rendering.
Lines : Device.DrawUserPrimitives(PrimitiveType.LineList) - 51% of all rendering time. The rest of the time was spent building the arrays of structs which contain the required data points to be passed to DirectX for rendering.
Polygon : Device.DrawUserPrimitives(PrimitiveType.LineStrip) – 49% of all rendering time. The rest of the time was spent building the arrays of structs which contain the required data points to be passed to DirectX for rendering.
Text : Direct3d.Font.DrawText() – 98% of all rendering time.
Coordinate transformations were once again accomplished via a translation matrix which can be passed to the device context.
Performance results were mixed. The actual rendering times are really good if the time it takes to morph the data into a format that DirectX can utilize is ignored. On the rendering of points the DirectX API was only bested by OpenGL. The rendering performance in terms of lines and polygons was midway between the other libraries. DirectX text rendering was the slowest of all the libraries.

6.4 OPENGL

OpenGL was utilized via a 3rd party wrapper called OpenTK. It is a lightweight wrapper that more or less directly wraps the native OpenGL function calls. The

library does make use of the advantages associated with a managed language like generics and strongly typed enumerations. OpenGL is comparatively very easy to use and there is a lot of help available. It is a very powerful API which can accomplish the rendering of very high quality graphics at high speeds. OpenGL supports the required primitives for rendering points, lines and polygons. OpenGL does not have support for drawing text and an OpenTK extension was utilized in order to facilitate this functionality. The extension library utilized is called QuickFont. In order for OpenGL to render text, it is converted to a bitmap which is then sent to the graphics card in the form a texture. The texture is then displayed showing the text.
Drawing of the point, line, polygon and text was accomplished using the following methods:
Points : GL.Begin(BeginMode.Points) and Vertex2 – 6% of all rendering time. The rest of the time was spent setting up the OpenGL context.
Lines : GL.Begin(BeginMode.Lines) and Vertex2 – 58% of all rendering time. The rest of the time was spent setting up the OpenGL context and looping through the data.
Polygon: GL.Begin(BeginMode.Polygon) and Vertex2 – 53% of all rendering time. The rest of the time was spent setting up the OpenGL context and looping through the data.
Text: QFont.Print() – 91% of all rendering time.
Coordinate transformation was accomplished yet again using a translation matrix. An orthographic projection was utilized during the setup of the OpenGL context. OpenGL has really good performance. The only rendering function that had slightly slower performance was that of text rendering. On the two laptops where the graphics card was not as good as the PC's the text rendering was faster in GDI+.

6.5 DIRECT 2D API

In order to use Direct2D a 3rd party lightweight wrapper called SharpDX was used. SharpDX is a fully featured managed DirectX API that wraps the COM libraries. Direct2D is also quite easy to use and is similar to GDI/GDI+. The library also does not have a point feature so a fill rectangle structure was used to render points. It supports lines and path geometries which can be utilized to build up more complex objects.
Drawing the point, line, polygon and text was accomplished using the following methods:
Points : Direct2DRenderer.FillRect() – 85% of all rendering time.
Lines : Direct2DRenderer.FillGeometry() and LineString– 20% of all rendering

time. The rest of the time was spent getting the data into the geometry path structure.
Polygon: Direct2DRenderer.FillGeometry() and Polygon– 21% of all rendering time. The rest of the time was spent getting the data into the geometry path structure.
Text: Direct2DRenderer.DrawText() – 94% of all rendering time.
Coordinate transformation was done using a matrix which can be passed to the Direct2D context much like GDI+.

7. DISCUSSION/CONCLUSION

The conclusion of the benchmark has yielded some surprising results. OpenGL and DirectX were assumed to have the best performance in terms of rendering speed and quality. Although this fact is true for OpenGL, DirectX did not perform as well as expected. As previously mentioned most of the time was spent creating structs which could be consumed by DirectX. There are optimizations available for DirectX to increase its performance but further research in this regard will have to be conducted in order to determine the magnitude of the benefit. The research conducted here was to determine the fastest library to use in order to implement a GIS rendering system keeping in mind ease of use. In terms of utilizing a library for software development, documentation is of utmost importance. In this regard there is a lot available for OpenGL. There was also no shortage available for Direct2D and GDI+. DirectX examples were scarce for C# and in terms of managed languages the documentation it is non-existent. There is however a number of usage examples as part of the SDK. The examples however are for C++ and although they are somewhat helpful, it was tricky to get the library to render the graphics using C#. This was due to the fact that not all the functions map directly to their managed counterparts. Of all the libraries GDI+ and OpenGL were found to be easiest and most intuitive to use. OpenGL yielded the best results with the overall best ease of use to performance ratio. Direct2D was also not difficult to get working although it is only slightly faster than GDI+.
The performance of GDI was poor, due to the fact that the coordinate transformations had to be performed via CPU. The API itself expects all coordinates to be in device coordinates which translate directly to the screen. In the case of the other libraries dedicated functions were available for translation making the process easier and more efficient.

DirectX and OpenGL perform the vertex translations via the graphics hardware if it is available. This makes the process quick and efficient. The newer versions of OpenGL and DirectX allow for the use of shader programs. This gives a lot of flexibility and control over how a scene is rendered. A major speed

increase could be achievable by preforming re-projection in a shader program. This makes OpenGL and DirectX a very good candidate for the implementation of a rendering engine. If more time is spent on optimizing the current benchmark implementation of GDI the rendering speed will be comparable to GDI+.

One other benchmark that was performed and which deserves mention was the use of a vertex buffer for rendering the points lines and polygons in OpenGL. The results here show truly what the benefits of using hardware acceleration can yield. Every frame after initial data load was magnitudes faster than the other methods. In terms of the graphics card available to the PC the re-rendering of the frames was instantaneous (less than 1 millisecond). The down side here is that video memory on the graphics card is limited and storing an entire GIS data set in video memory is not going to be feasible. The challenge in building a high performance GIS renderer will be to efficiently make use of the limited on-board video memory. Streaming data to and from the graphic card is also comparably slower than the speed at which the GPU can process the data so to get good performance this will have to be carefully managed and optimized.

Although using OpenGL is slightly more complex than utilizing a pure 2D API like GDI+ these results show that there are some benefits. Hardware acceleration can greatly benefit 2D drawing for use in a GIS renderer especially where advanced drawing methods are concerned. An additional added benefit to utilizing this library is the support for 3D data structures which will allow the expansion of the renderer to accommodate 3D scenes.

After implementing the rendering via the different API's, the best conclusion that can be drawn is that OpenGL is a viable solution to the GIS rendering problem. Correctly utilizing this library will allow for fast high quality rendering to be performed which will benefit a GIS greatly.

8 REFERENCES

Gahegan, M. (1999). Four barriers to the development of effective exploratory visualization tools for the geosciences. *INT. J. Geographical Information Science*, *13*(4), 289-309.

Jones, W. (2004). Beginning DirectX 9. Boston: Stacy L. Hiquet.

Microsoft. (2012, March 7*)*. About Direct2D. Retrieved April 19, 2012, from Windows Desktop Development: http://msdn.microsoft.com/en-us/library/windows/desktop/dd370987%28v=vs.85%29.aspx

Renhart, Y. (2009). Fast Map Rendering for Mobile Devices. Master Thesis, University of Gothenburg, Department of Applied Information Technology, Gothenburg.

Richard, N. G. (2002, November 15). Microsoft Windows' Graphics Device Interface (GDI*)*. Retrieved April 16, 2012, from http://classes.engr.oregonstate.edu/eecs/spring2003/ece44x/groups/g1/WhitePaperRichard.pdf

The Khronos Group. (2012). OpenGL Overview. Retrieved April 19, 2012, from OpenGL: http://www.opengl.org/about/

Tory, M., & Moller, T. (2004). Research, Human Factors In Visualization. *IEEE Transactions on Visualization and Computer Graphics, 10*(1).

Walbourn, C. (2009, August). Graphics APIs in Windows. Retrieved April 16, 2012, from MSDN Dev Center: http://msdn.microsoft.com/en-us/library/windows/desktop/ee417756%28v=vs.85%29.aspx

Wang, Y.-M., & Chung, P.-Y. E. (n.d.). Retrieved April 19, 2012, from Exploring Customization of Distributed Systems using COM: http://research.microsoft.com/en-us/um/people/ymwang/papers/HTML/COMEssay/S.htm

International Organization for Standardization, ISO 19101, Geographic Information, 2002

Luten, Eddy. OpenGLBook.com. 19 12 2007. http://openglbook.com/the-book/preface-what-is-opengl/ (accessed May 23, 2012).