Briefing:
The Relevance of Videogame Graphics

A review of the evolution of videogames graphics processing, the importance of triangles, the efficiency of geometry, its employment within the rendering pipeline, its emergence as a driving force behind GPU development, the new world of GPU Computing and the convergence of gaming technology in the world of BigData & Analytics.

December, 2013

CONTENTS

ABSTRACT

The purpose of this paper is to attempt to highlight the criticality of videogames as a component of the "convergence" of technologies (Cloud, Gaming/MMOG, Gamification and BigData) that is clear to those inside the IT world. This paper is written in as much a chronological manner as could be achieved to try to take business, non-IT, non-programming literate readers on a journey to reveal why the humble graphics processing capabilities have become part of the bedrock for our future exploitation of computer processing as a whole.

In doing so it is hoped this paper will answer some seemingly simple questions during the journey, namely:

i. Why GPU's were developed;

ii. Why triangles are so important to graphics processing;

iii. Why high degrees of parallelism are becoming increasingly important;

iv. How GPU's are being utilized to deliver significant gains in industries and market sectors far beyond the original design criteria for the GPU; and

v. Why GPU's cannot wholly replace CPU's and that the future is most likely a symbiosis of the two capabilities leveraging each for their inherent strengths.

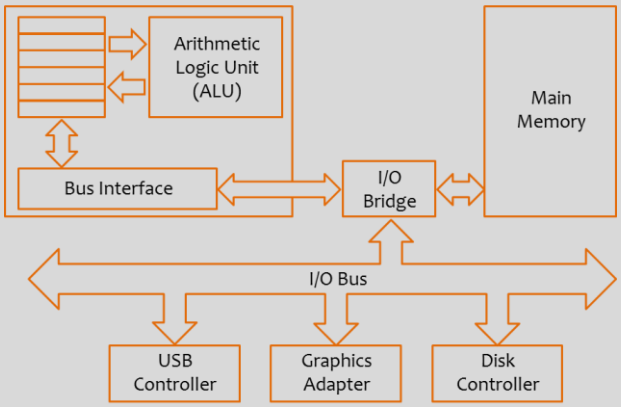
1. INTRODUCTION TO VIDEOGAMES

The previous briefing paper "An Introduction to Gaming & MMOG" finished with the observation that computer graphics, engine architecture and networking & communications are fundamental components of modern video and online games. This section seeks to expand upon the developmental history of computer graphics, to provide an outline of its current (2013) status, and to explain why it is now becoming the bedrock upon which BigData and analytics will stand. This paper is not an exhaustive introspective on videogaming as an industry rather it is a briefing paper that outlines the key developments over the past 60 years in order to highlight the exceptional advances that have been made and the collateral benefits we are about to harness today for areas far beyond the original motivations.

1.1 Number Crunching, Batch Runs & Spooled Reports

Today we are all very familiar with having computers permeate almost every aspect of our lives, especially with the advent of tablets and smartphones. This recent ubiquity in computing masks many of the millennial generation from the time before the Internet, Google, Apple, Samsung and Facebook. That was a time when computing really was a "heavy duty" job. Indeed even today those big jobs continue in data centers the world over. Those tasks are batch computing jobs running overnight (although often now intra-day also) on mainframes and mid-range computers. This type of computing began with LEO, the Lyons Electronic Office in 1951 which crunched the numbers on bakeries evaluations. The pioneers of business computers were companies such as International Business Machines (IBM), Digital Equipment Corporation (DEC), Wang, Xerox, and Data General to name but a few. Batch jobs continue to be critical today across all industries for bank account updates, logistic manifests, customer billing and transaction updates. The outputs from batch processes used to be spooled or printed reports which internal administration and accounting functions would pour over during the day to check stock levels, orders, customer credit status and much more.

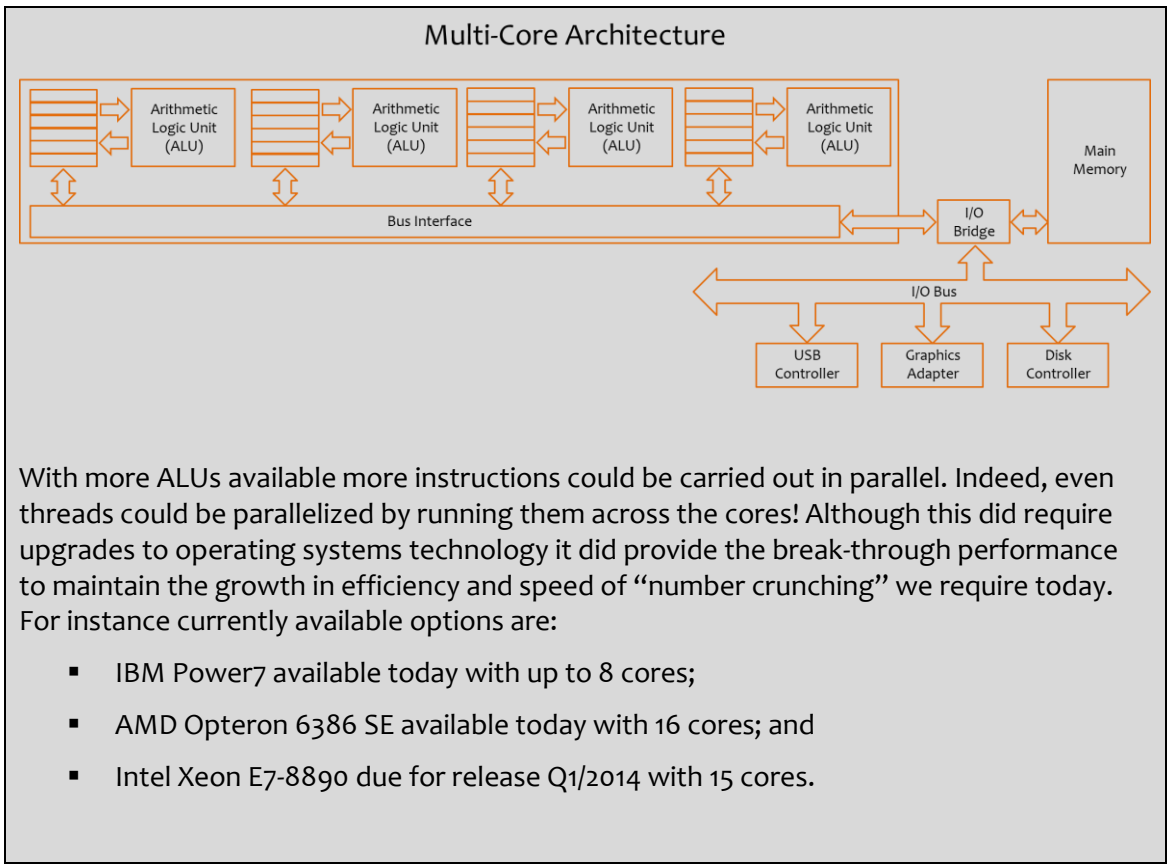
> ***Quick Tangent on Central Processing Unit (CPU) Architecture***
>
> Without getting too technical it is worth noting at this juncture the effect the need for "number crunching" had on the development of computer processor design. This will link later in section 2.0 to graphics processing. The design for computers has followed an originating architecture outlined in 1945 by John von Neumann of Princeton University. Essentially a central processing unit can be viewed logically as being comprised of three components surrounded by supporting input and output mechanisms, visually:



While acknowledging that this was an incredibly efficient design, processor designers knew that inherent difficulties would meet a ceiling and sure enough we reached it just before the turn of the century. The expected limitation is the amount of empty instruction cycles or null cycles when the ALU is waiting for instructions to be fetched from memory and loaded into the registers in order for it to compute. The decades from the forties to the nineties saw developments focusing on alleviating this issue with (some examples):

- Cache Memory – A small amount of memory located right next to the processor (denoted L1 cache) and another cache located slightly further away (denoted L2 cache). Common or frequent instructions can be stored in these caches to reduce their fetch time;
- Pre-Fetch – Because of the general sequential nature of computer programs it is possible to anticipate the next required instruction and to fetch in advance those instructions to reduce the wait time;
- Pipelining – Think of the processor being set up like an assembly line in a factory where performance of many operations can happen in parallel. So if a processor uses five stages (such as RISC processors: FETCH->DECODE->EXECUTE->MEMORY->WRITE) it means the fetch for the next instruction happens in parallel with the decode of the current greatly speeding up processing instructions; and
- Threading – Threads are many small sequences of instructions within a computer program which can be executed independently of each other. Essentially the main program can be split into threads and each thread can run independently being cached, pre-fetched and executed once the processor is available.

All this development improved processing speeds and made it possible for computing to deliver "number crunching" at greater and greater efficiencies. There was still a limit though and from the late nineteen-nineties the semi-conductor giants knew a new concept was required and that was multi-core processing. This can be visualized below:



With more ALUs available more instructions could be carried out in parallel. Indeed, even threads could be parallelized by running them across the cores! Although this did require upgrades to operating systems technology it did provide the break-through performance to maintain the growth in efficiency and speed of "number crunching" we require today. For instance currently available options are:

- IBM Power7 available today with up to 8 cores;
- AMD Opteron 6386 SE available today with 16 cores; and
- Intel Xeon E7-8890 due for release Q1/2014 with 15 cores.

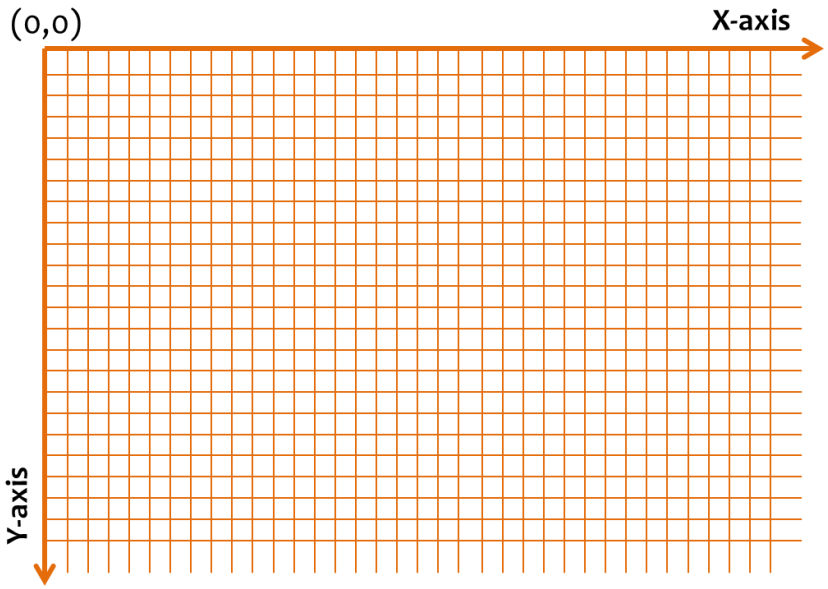
While clearly a huge benefit, reading reams of reports and papers was a laborious task and even then localized calculations were required to be carried out by hand during the 1970s for submission at the close of day or close of week by local offices. This created a need for a general purpose local computing which was filled during the 1980s with the explosion of the personal computer (PC). Introduced in 1981 by IBM and quickly cloned the PC continues to be the workhorse of the office computing environment. Home computing also took off during the 1980s with Apple, Acorn, Atari, Sinlcair and Commodore leading the way.

1.2 The Need for Visualization & Interactivity

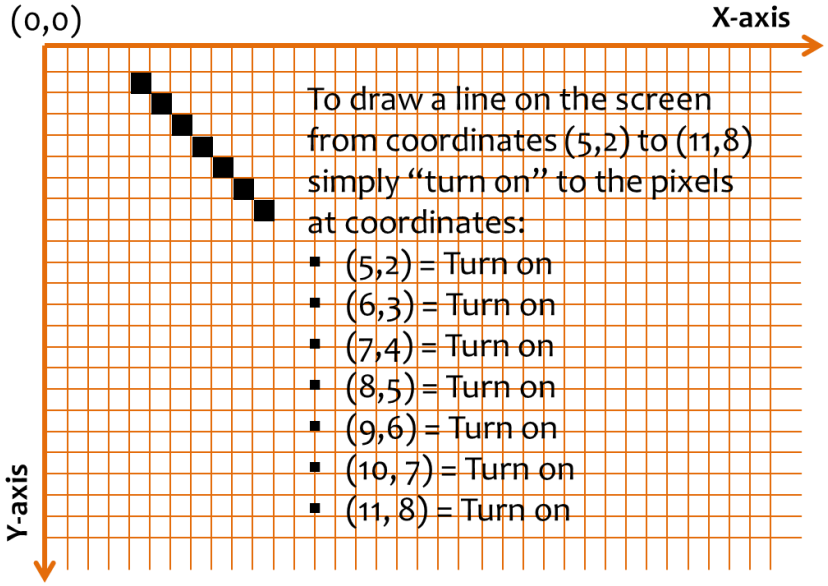
Of course the PC and home computing revolutions required affordable microprocessors but given 50% of our brains neurons are associated with vision we needed something even more important – displays & graphics – we needed to be able to display our outputs. Neither business nor home users would have invested as heavily as they did if we couldn't "see" into our devices, if we couldn't input characters and visualize the results. This race had been going on for a couple of decades beforehand with businesses and scientists also needing to view inputs and outputs. The breakthrough came in the seminal work of Ivan Sutherland for his Ph.D. thesis which introduced the world to "Sketchpad" and interactive computer graphics.

We need to briefly return to the late 1800s to understand computer display graphics and the works of the famous artists Georges Seurat and Paul Signac and other pointillists of the period. The approach of pointillist artists was to form a picture from single points or dots of colour in such a manner that viewed at a distance the dots form a cohesive image. Irrespective of whether or not computer graphics were ever deterministically construed as a form of pointillism the parallels are obvious. An image or visualization can be presented to the human eye formed from

many very small points of colour. This approach is now the cornerstone for all computer graphics with pixels - single indivisible points of illumination projected onto a display to form characters, lines and ultimately images. For computer processing it means that we can think of the display as being made up of a grid of single points and manipulate the existence and colour of those single points. For instance, (Now I know I should have listened better to Mrs. McKenna's Cartesian coordinate classes! Please also be aware that a computer screen is not precisely a Cartesian system and that mapping is required to convert to a display format but for the purposes of description hopefully this simplified view should suffice) if we first divide up a display unit from the top left hand sides highest corner to the lowest right hand side we can create a grid thus:

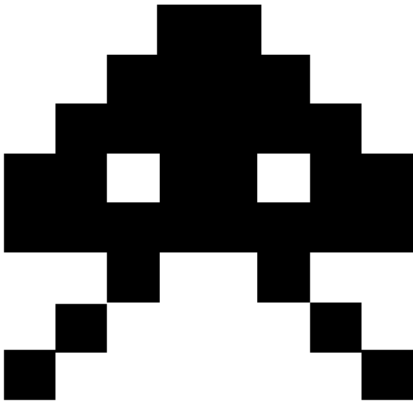


Now we can tell the computer to "turn on" pixels (we will deal with colours later in Section 2.0) at certain points in the coordinate system to create lines and shapes.

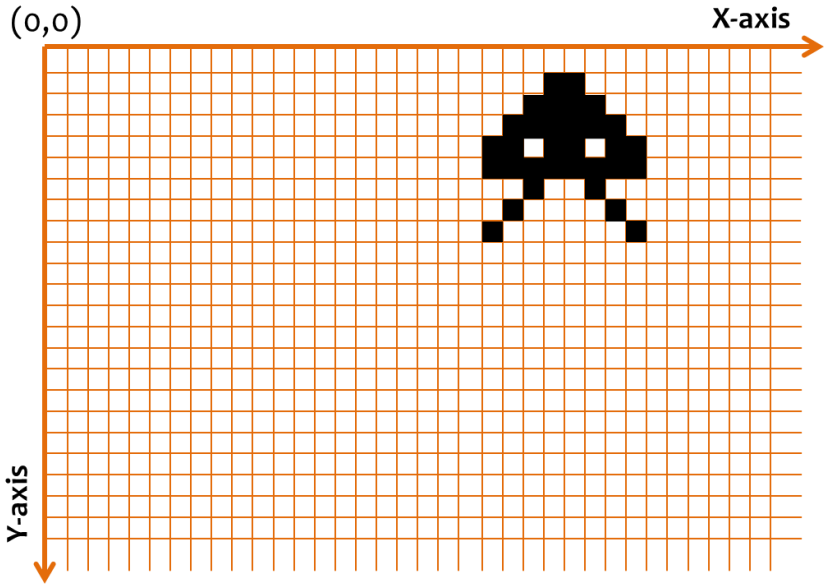


So in a simplistic sense the sequence of binary "turn on, turn offs" could be built to display little figures. On the left below when pushed to the display processor, with 1's meaning on and 0 meaning off, the binary sequence would be displayed as on the right:

0	0	0	1	1	0	0	0
0	0	1	1	1	1	0	0
0	1	1	1	1	1	1	0
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1
0	0	1	0	0	1	0	0
0	1	0	0	0	0	1	0
1	0	0	0	0	0	0	1



Putting this into our screen grid yields:

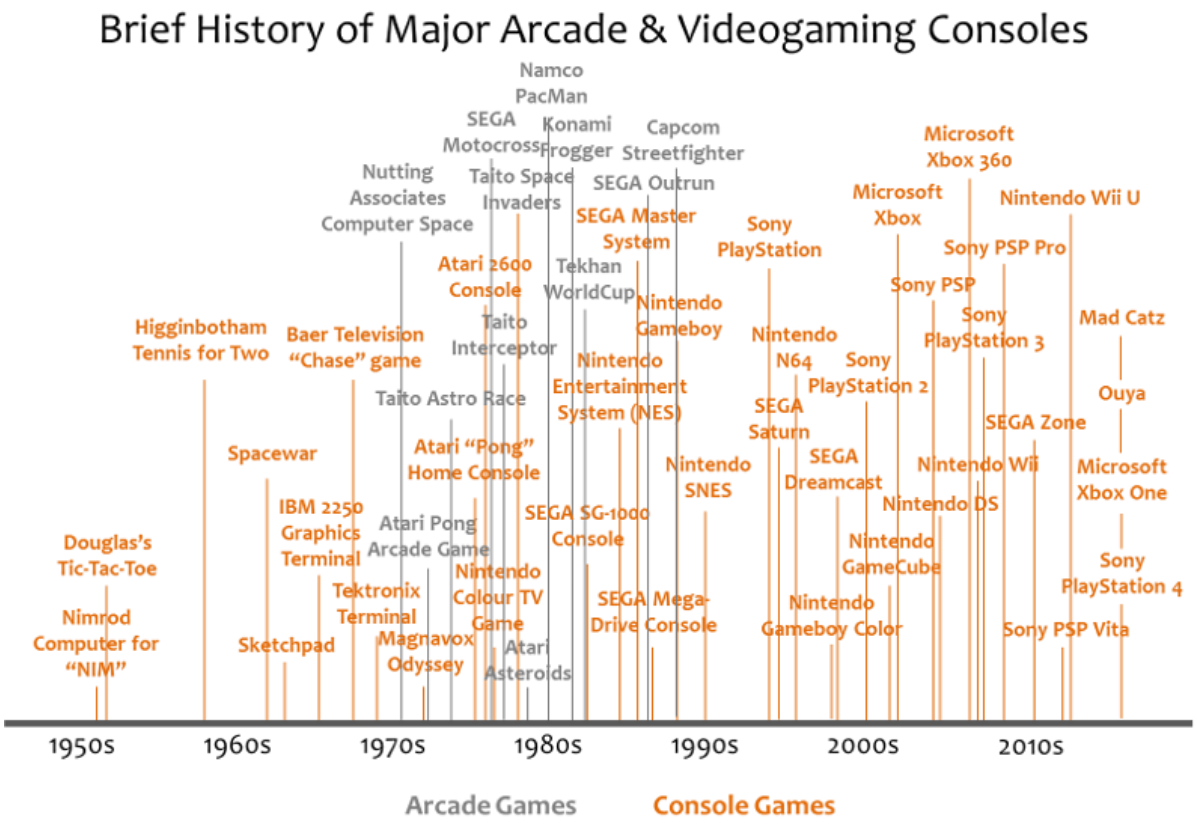


Once a programmer knows how to place the "0's & 1's" onto a display screen they can engage an appropriate time delay and mimic movement of lines and shapes by changing the placement of the "0's & 1's" over a time cycle. This gives us animation in the same way you could draw a stick figure on the edges of a book and flick the pages to create a sense of animation. The last piece of this jigsaw returns us to "Sketchpad" which in paradigm terms created the link between an external human movement and the computer display visualization. This innovation meant that the human could interact with and control the movement of the pixels without needing to understand precisely how this was done. It abstracted all of the complexities involved with doing this from the user. It didn't take long for other innovators to grasp the significance of interactivity and how it could be used to build an entire industry.

1.3 The Development of Videogames

As we have seen there are clear reasons why computers are useful within a working office or factory environments. Crunching the numbers, writing memoranda, writing documents and

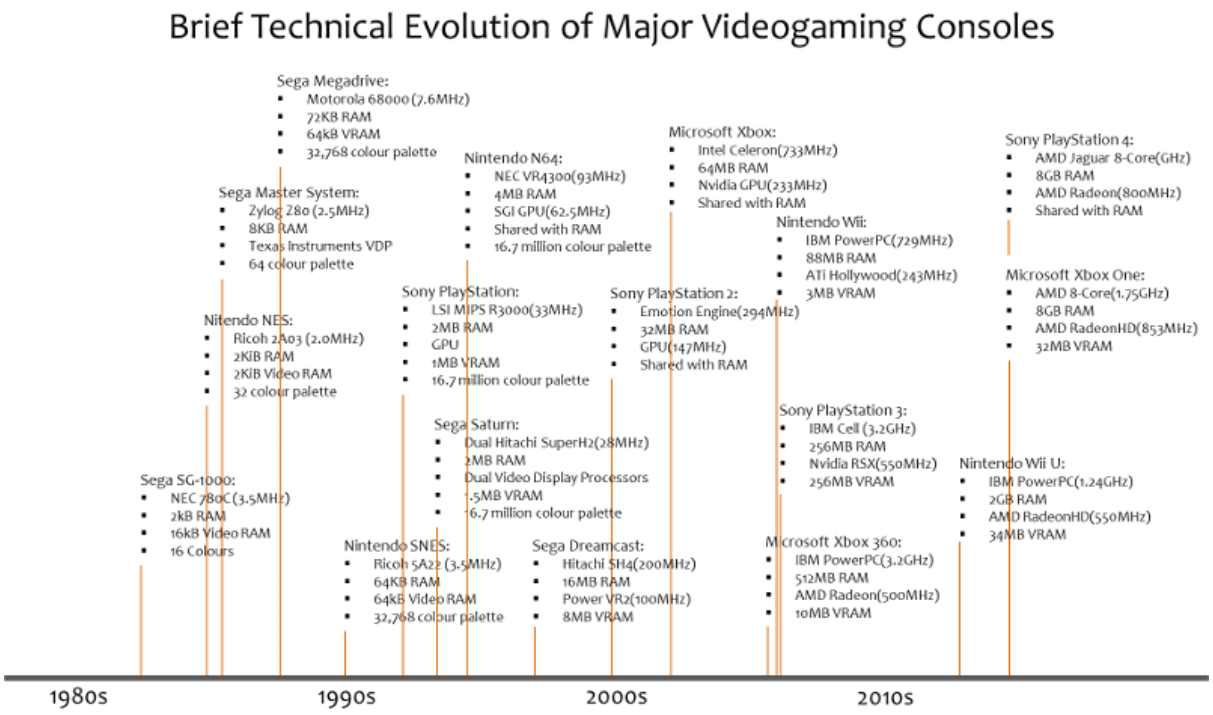
storing them electronically are all much easier with computers. Those capabilities though would not solely explain the reasons why home computers in the 1980s took off. Neither would an innate sense of curiosity and desire to learn to computer programming. A major reason for purchasing a home computer and often its main utility was for home users was to play computer games which harnessed the emergent graphical manipulations we examined above to create interactive games. They were preceded by character games such as "Tic-Tac-Toe" but the real breakthrough, in arcade terms was Computer Space from Nutting Associates, and the Atari "Pong" in home consoles. A brief history of the major arcade and home consoles developments through the past 60 years is presented below.



So we're back to why? Why create games? Why did Nutting Associates create "Computer Space"? The answer is as old as humanity itself – you could make money from these things! Even as far back as 1971 the component parts of a "Computer Space" machine were fairly commodity. You take a GE 15" television, some 74-Series TTL chips, diode arrays, a bunch of labour (reportedly 200 hours) and hey presto you get an arcade machine for a unit cost of about $1000 or just under. This means at $0.10c a go for about 5 minutes the machine breaks even in approximately 83 to 100 days with every $0.10c thereafter profit. This considerable motive invigorated investments in arcade machines and led to the explosion of arcades all over the world.

Certainly in the early decades of evolution the component parts which were pulled together to build the arcade and games consoles were available and the path trodden was clearly one of following the development of processing chips, memory and hard disk drives. During the late 1980s and early 1990s, a change ensued, with game manufacturers' (Note: It is worth noting that in parallel other industries were also asking chip designers and fabricators to 'push the envelope' in areas such as industrial designers for CAD/CAM systems, film production companies and medical research firms, etc.) beginning to drive investment in chip design, in order to gain a competitive edge on their competitors. This led to progressive advancements first in central

processing units and latterly in the development of graphics processing units. This evolution in technology can be visualized in the home video gaming consoles presented below:



Note: For simplicity I have not differentiated between GDDR, DDR and other specific RAM types.

The scene was now set for a major expansion and evolution in the ability to process computer graphics.

1.4 Sources & Further Reading

http://educationalstuff1.tripod.com/bsp.pdf

http://en.wikipedia.org/wiki/Computer_graphics

http://www.slideshare.net/senthil23051987/history-of-video-games-25234347

http://en.wikipedia.org/wiki/History_of_Nintendo#1956.E2.80.9374:_Toy_company_and_new_ventures

http://en.wikipedia.org/wiki/Sega

http://en.wikipedia.org/wiki/List_of_video_game_consoles

http://en.wikipedia.org/wiki/Arcade_game

http://en.wikipedia.org/wiki/Timeline_of_arcade_video_game_history

http://blogs.wsj.com/tech-europe/2011/11/14/worlds-first-business-computer-celebrates-60th-anniversary/

http://www.xerox.com/about-xerox/company-history/enus.html

http://www-03.ibm.com/ibm/history/history/history_intro.html

http://en.wikipedia.org/wiki/Digital_Equipment_Corporation

http://en.wikipedia.org/wiki/Wang_Laboratories

http://en.wikipedia.org/wiki/History_of_computing_hardware_%281960s%E2%80%93present%29

http://www.fundinguniverse.com/company-histories/data-general-corporation-history/

http://en.wikipedia.org/wiki/History_of_personal_computers#The_IBM_PC

http://www.webexhibits.org/colorart/jatte.html

http://www.hpl.hp.com/hpjournal/pdfs/IssuePDFs/1970-12.pdf

http://www.davemanuel.com/median-household-income.php

http://www.retrocrush.com/archive2/computerspace/

http://www.arcade-museum.com/manuals-videogames/C/ComputerSpace.pdf

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

http://www.slideshare.net/timothy.hall/history-of-cpu-architecture?from_search=1

http://www.slideshare.net/macasajaneg/11-computer-systems-hardware-1

http://www.slideshare.net/jabaktash/final-draft-intel-core-i5-processors-architecture

http://en.wikipedia.org/wiki/Von_Neumann_architecture

2. THE DEVELOPMENT OF COMPUTER GRAPHICS PROCESSING

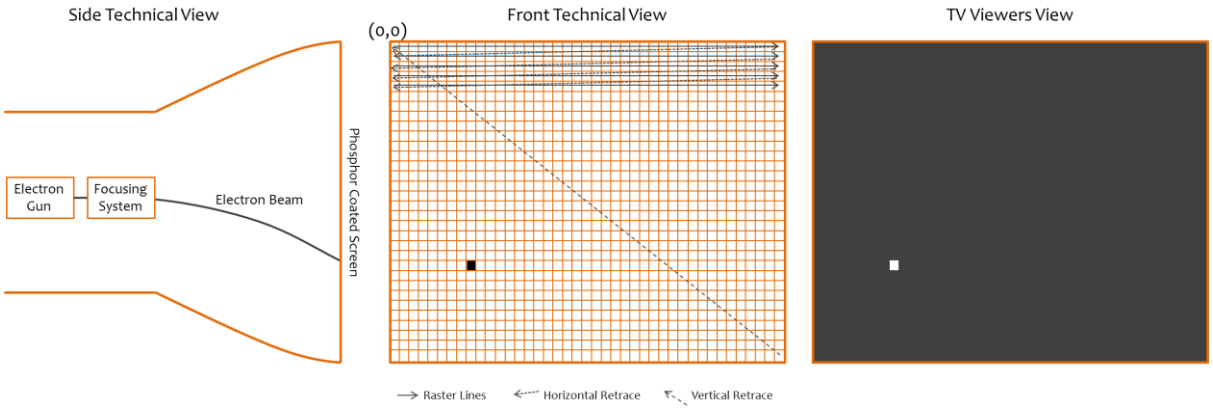
Unfortunately we have to return to some theory to fully understand the why of where we have ended up today with computer graphics. We have seen how pixels can be generated and how a simple Cartesian-like coordinate system can be used to generate images and pictures. But simple lines and points of colour were never going to be sufficient for the human eye. Another important distinction to make is that this paper is solely concerned with computer graphics which is defined as the pictorial presentation of objects based on their computer model rather than image processing with is the use of a computer to manipulate, enhance or pattern detect from actual pictures. Today we look back at the simplistic graphics of "Space Invaders" and others with a melancholy smile at their primitiveness.

2.1 The Requirements of Graphics – Making Pictures in 2D

There is an abundance of literature on Computer Graphics available if the reader wants to be fully versed in the subject (see Sources & Further Reading below). There are though just a few key additional underlying technological innovations to understand before these theoretical foundation sections can be coalesced into a form where their significance or relevance can be understood in the context of Cloud, BigData and analytics.

2.1.1 Rasterization

The first of these is rasterisation. Simply stated rasterisation is the process of converting a visual image into a stream of electrical pulses and was first used in rudimentary scanning technologies in 1881. In 1884, Paul Gottlieb Nipkow envisioned rasterisation for scanning images and transmitting them sequentially in a patent for a mechanical television system. The Nipkow scanning disk was actually used by John Logie Baird in 1925 to demonstrate the transmission of a moving silhouette image. This raster scanning technique was also employed in the production of images on the Cathode Ray Tube (CRT) display. Essentially the electron gun reads the "1's & 0's" described earlier (Sub-section 1.2) and scans across from left to right starting at (0,0) and tracing a path downwards line by line from top to bottom with 1 being shot and zero not. If it's 1 then an electron is fired and produces luminescence on the phosphorous coated screen. This happens continually to keep the image fresh:



So when computer graphics were being conceived we already had a workable and popular display technology which we could utilize – the television or more precisely the CRT. Moreover, the underlying conceptualization of raster scanning or rasterisation, as we have seen, is simply

the employment of a two dimension matrix of values with one value for each of the pixels. When stored as a file this is a bitmap file and for black & white looks like this:

0	0	0	1	1	0	0	0
0	0	1	1	1	1	0	0
0	1	1	1	1	1	1	0
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1
0	0	1	0	0	1	0	0
0	1	0	0	0	0	1	0
1	0	0	0	0	0	0	1

With raster displays and bitmap files storing pixel values in place the world of computer graphics was catching on fast.

2.1.2 The Framebuffer

In the 1970s to be able to continually refresh the screen in order to maintain (or change) the image the display unit would need to read in all of the pixels which even for 320x100 pixels would be 32,000 pixels, even at 1 bit per pixel for black & white this is 32kb or 4KB of framebuffer memory a huge amount in the 1970s (in addition to project moving images smoothly often a technique of double-buffering would be employed with the next frame being filled into a framebuffer while the current frame is being projected in essence doubling the framebuffer memory requirements). In order to maintain a screen efficiently raster scan systems started to employ a memory specifically to hold the frames which could act as a refresh buffer. This became the framebuffer.

Over time as main memory and then graphics memory became inexpensive and common place the function of the framebuffer became part of the graphics hardware. In particular modern computing systems employ all manner of virtual framebuffers (see Linux fbdev) and graphics acceleration techniques where the CPU off-loads graphics processing to the GPU which feeds a dedicated framebuffer (also called the VideoRAM or VRAM). Nevertheless the basic function of storing the matrix values of a frame of a computer generated image remains.

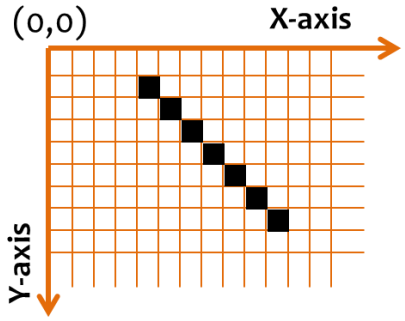
2.1.3 Anti-Aliasing (AA)

The primitiveness of early computer graphics is probably most visible in terms of the line drawing shown previously and provided once again below. The problem for human viewers is that of jagged edges. The problem becomes even more apparent when circles or eclipses are required with curved lines looking anything but. This is called aliasing and it stemmed in the early years from the binary nature of pixels i.e. either on or off, and even in later days from rounding-off by the rasterisation algorithms. Clearly the most obvious way is to provide AA is to improve the display resolution to the point where the human eye cannot discern the stepping but this expensive option was not available to designers in the 70s, 80s, 90s and 00s – even today it remains an expensive options. So to alleviate this stair step appearance:

i. Mathematical techniques were perfected; and

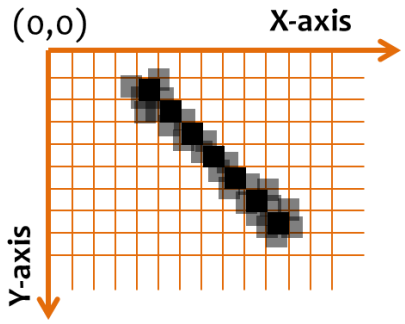
ii. Improvements in display technologies to enable pixels to have "grey" values per pixel between just on and off

Starting Point: Aliasing of the Line



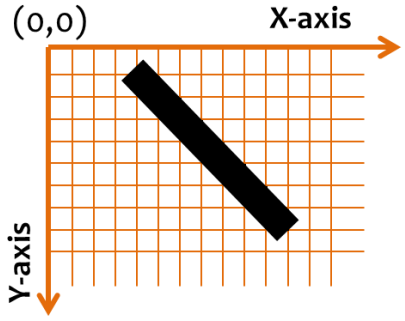
The first technique is often called super sampling and is an AA technique in which the image is rasterized at a higher resolution using a pseudoraster and groups of subpixels are averaged to obtain the values for surrounding actual pixels.

Over or Super Sampling of the Line



This can be visualized in the second picture. A mathematical algorithm can then be employed to isolate the specific areas that require "filling" to smooth the final displayable output.

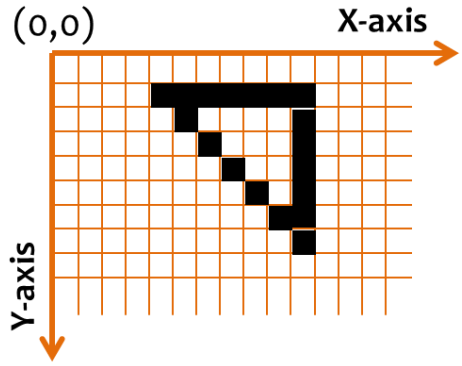
Anti-aliased or Smoothed Line



For the smoothing effect of anti-aliasing to have optimal effect the display technology should be capable (which most are today) of sub-pixel or in-between or shades of pixels. The Super-sampling in conjunction with enhanced display ability it creates a much smoother image.

2.1.4 Polygons & Vertex's

Once we have lines it is not too much of a jump to begin building shapes by organizing lines to appear in specific places. For instance from our example above if we now defined two additional lines we can create the effect of a shape or polygon – in this instance a triangle. A vertex is the corner of a polygon where two edges meet, commonly in computer graphics a triangle with every triangle then composed of three vertices.

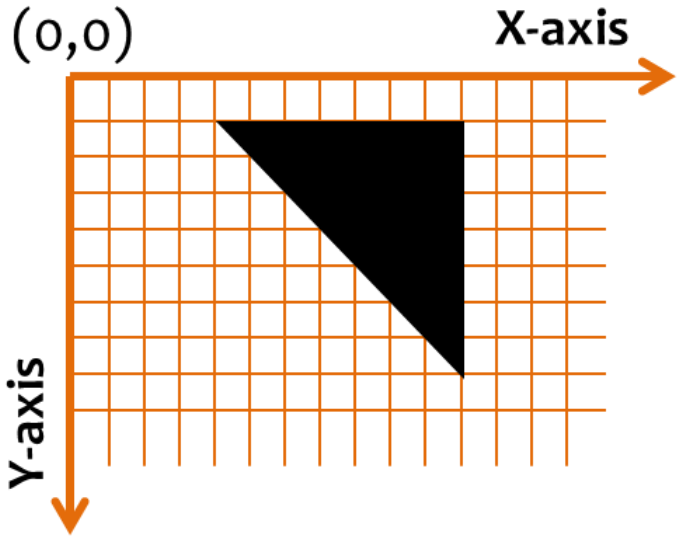


Three lines defined as:

- Line #1 From (x=5, y=2) to (x=11, y=8)
- Line #2 From (x=5, y=2) to (x=11, y=2)
- Line #3 From (x=11, y=2) to (x=11, y=8)

This produces the effect of polygon, in this instance a triangle.

Now, in mathematical terms, that we know the vertices we can fill in the space captured within the shape and carry out the anti-aliasing to complete the picture:



Polygons and vertices are the basis for 2D and 3D modelling in computer graphics and we will return to these in more detail below (Section 3).

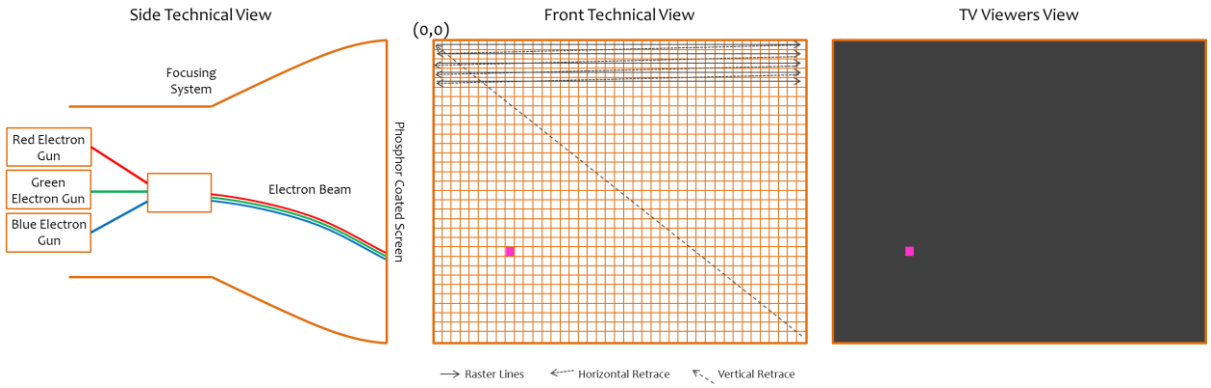
2.1.5 Colour Spaces: Sorting out Colours

In order to remain focused and not be thrown significantly off on a tangent it is noteworthy that colour as a subject is truly enormous so this section will be bounded specifically to its generation as part of computer graphics. Links to broader articles and books on colour are included below. By way of priming this section, colour is not an objective physically measurable attribute like mass, weight and the like. Rather it is distinguished as a property in that it is definable wholly in terms of human perception. This significant area of study is called Colorimetry.

So the final primitive aspect of computer graphics to be discussed in this Section is to return to colour. Black and white was a revelation and interactivity delivered the capability to engage users far beyond a community of developers. This paper has previously alluded to introducing the subject of colour so to finalize this two dimensional introduction we'll have a look at how graphics transitioned to colour, whose introduction brought incredible results for users, developers and not least gamers. As a property colour has an incredible psychological and

perceptual effect on human beings even interfering with our response and reaction time to stimuli (see Stroop Effect). Our lives from our first days are full of colour so it was absolute necessity as the mother of all invention that drove teams on to develop colour graphics and colour displays.

We can all recall early school classes, either science or physics, when we were first introduced to the concept of red, green and blue light streams and their additive effects, blue and green light mixing to produce turquoise, red and blue mixing to produce purple, etc. This mechanism formed the basics for colour CRT displays. Engineers knew that they could produce different electron guns that would deliver specific colours of electrons red, green and blue. In addition the phosphorus coating could be applied in a pattern of red, green and blue reactive phosphorus per pixel. Together these technological advances would be delivered in precisely the same way as described previously to effect the production of colour:



The required computing and graphics innovations to drive this mechanism were threefold:

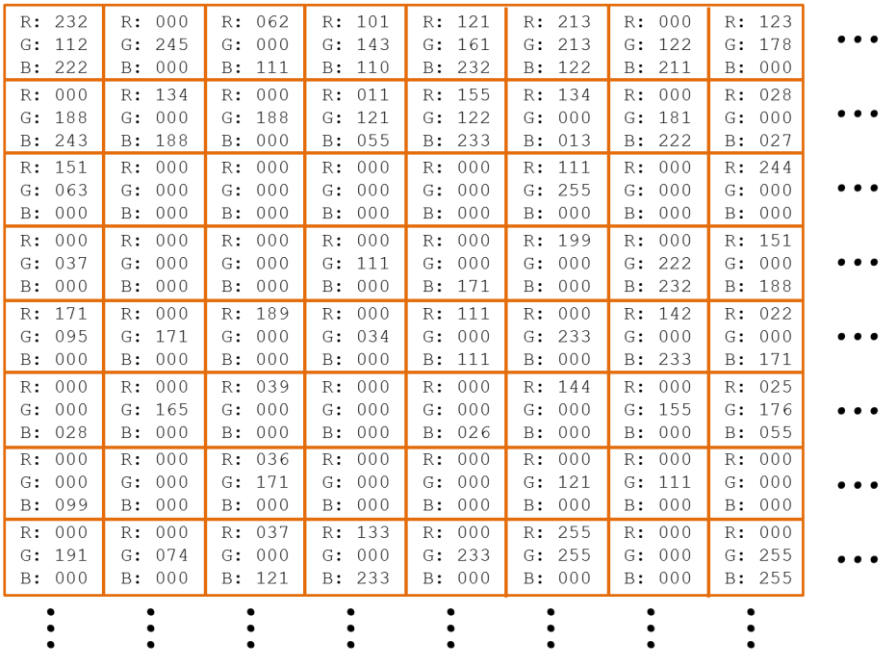
i. A defined colour space of Red, Green and Blue (RGB);

ii. A colour lookup table (CLUT);

iii. An increased size of framebuffer memory.

Early implementations were limited in terms of colour capability given the amount of memory and early implementations were 1-bit for two colours, 2-bits for four colours, 3-bits for eight colours and 4-bits for sixteen colours. Remember earlier we needed a framebuffer of 32kb or 4KB just for black & white, now if we needed sixteen colours that same screen required a framebuffer of 64KB.

As computer display graphics hardware evolved a refinement in colours could be achieved through an expansion of the RGB colour space in line with framebuffer memory expansion. Today using 8-bits per colour we can define 256 gradients or hues of a particular colour and build a three dimensional cube comprising every intersection from (Red 00000000, Green 00000000, Blue 00000000) to (Red 11111111, Green 11111111, Blue 11111111) in order to build an RGB colour space. Each of the intersecting values in the matrix also provides a specific value to yield the CLUT. This means that a 24-bit colour space (8-bits for each of Red, Green, and Blue) for a 1024 x 768 resolution screen requires ~2.4MB of framebuffer. The framebuffer can be visualized as a bitmap file below:



2.2 Sources & Further Reading

http://inventors.about.com/od/germaninventors/a/Nipkow.htm

http://warehouse13.wikia.com/wiki/John_Logie_Baird%27s_Scanning_Disk

http://inventors.about.com/od/tstartinventions/a/Television.htm

http://en.wikipedia.org/wiki/Graphics_processing_unit

http://en.wikipedia.org/wiki/Analog_television

http://en.wikipedia.org/wiki/Framebuffer

http://www.opengl.org/wiki/Framebuffer

http://ecomputernotes.com/computer-graphics/basic-of-computer-graphics/what-is-frame-buffer

http://searchstorage.techtarget.com/definition/video-RAM

http://www.amd.com/uk/Documents/49521_Graphics101_Book.pdf

http://www.cs.cmu.edu/~ph/nyit/masson/history.htm

http://design.osu.edu/carlson/history/lesson2.html

https://www.udemy.com/introduction-to-graphics-architecture/

http://en.wikipedia.org/wiki/History_of_computer_animation

http://en.wikipedia.org/wiki/Colorimetry

http://graphicdesign.spokanefalls.edu/tutorials/tech/computerdisplay/Display.htm

http://www.cis.usouthal.edu/~hain/CSC520_F09/GPUpresentation.pdf

http://www.iryoku.com/smaa/

http://www.psych.rochester.edu/people/elliot_andrew/assets/pdf/2007_ElliotMaierMollerFriedmanMeinhardt_Color.pdf

http://en.wikipedia.org/wiki/Stroop_effect

Boreskov, A Shikin, E 'Computer Graphics: From Pixels to Programmable Graphics Hardware' (2013, CRC Press)

Shirley, P Marschner, S et al 'Fundamentals of Computer Graphics' (2000, CRC Press)

Foley, J (Editor) 'Computer Graphics: Principles and Practice' (1996, Addison-Wesley Publishing)

Govil, S 'Principles of Computer Graphics: Theory and Practice Using OpenGL and Maya®' (2004, Springer)

Klawonn, F 'Introduction to Computer Graphics' (2012, Springer)

Johnson, Amos 'Basic Concepts for Computer Graphics' (2007, iBook Store)

Ryan, D 'History of Computer Graphics' (2011, Authorhouse)

Cohen, J 'Visual Color and Color Mixture: The Fundamental Color Space' (2001, Google Books)

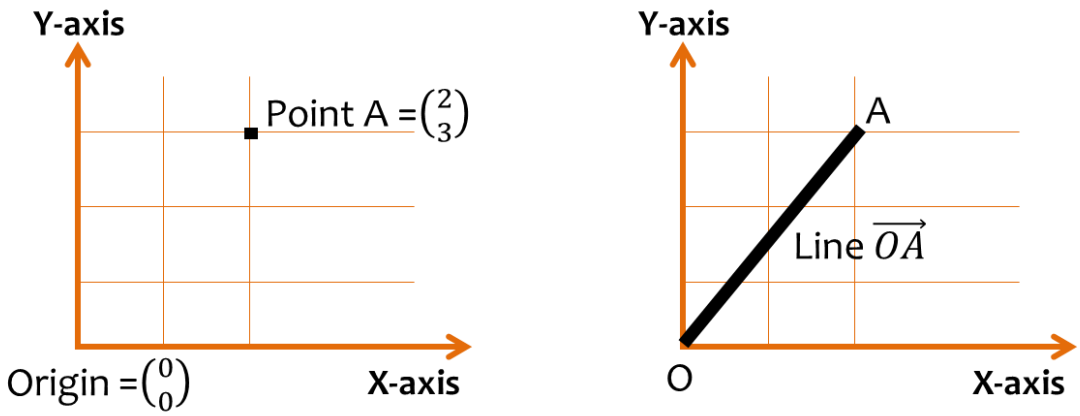
3. THE EFFICIENCY NEED FOR MATHEMATICS

In order to understand how this was all developing we have a few more layers of the proverbial onion to pull away and that involves getting into a little bit of the mathematics and back to Mrs McKenna (I really know now I should have listened better!).

3.1 Oh no not Mathematics .. Oh yes Mathematics!

We saw earlier that pixels could be defined in terms of a grid, for instance a 320x100 grid of pixels representing a display yields 32,000 possible pixel positions. We also saw how in a primitive way we could turn on certain pixels to build lines and shapes and in the very early days of generating a few lines here and there to demonstrate graphical capabilities this basic method would have sufficed. But to do any real work this hand cranking would be far too laborious. The answer was to develop an approach to drawing in two dimensions (2D) for lines, polygons, circles, eclipses and any shape using mathematical algorithms. These algorithms would use the computer to do the hard work of crunching numbers (this is sounding vaguely familiar –number crunching!) for us to produce our desired results. How? Well fairly easily.

Returning to our simple coordinate system we can define point O as an origin and as we add points into the coordinate system we could describe them as row vectors (2, 3) or column vectors $\binom{2}{3}$. Now given we know the origin O = $\binom{0}{0}$ using mathematics we can describe a line between them as being $\overrightarrow{OA}$.



Don't let the math symbols phase you they are just a) a simplified way for conveying ideas to others and b) more importantly we can now use mathematics to abstract us away from using numbers at all and work everything out using equations before plugging in real numbers - that makes a real difference! Why? Some reasons why:

1) We've been working on very simple scales of x=2 and y=3 with short lines (in this case $\overrightarrow{OA}$ equates to points $\binom{0}{0}$, $\binom{0.66}{1}$, $\binom{1.33}{2}$, $\binom{2}{3}$) but a computer screen even in the 1970s was 320x 100 pixels – so that would be 32,000 points. If we or the computer had to handle 32,000 numbers in long hand all the time we'd soon run out of patience and memory respectively;
2) Say we worked day and night to define a picture with lines and circles and had it all laid out, loaded into the computer, showed it to the boss and she said "we need to change

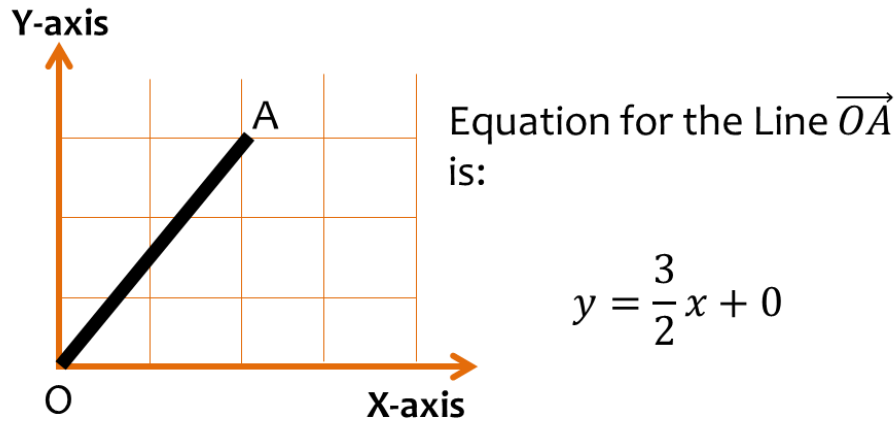
this line" or "we need to change this line to a curve". We would have to redo everything from scratch again;

3) Programming for lots of different types of shapes in long hand by defining each point of an ellipse or a curve of some sort would be very difficult;

4) 1 to 3 above would be surmountable for static pictures albeit with lots of hours of work, but what if we had a series of pictures and worse elements within those pictures moving, that's an awful lot of numbers and enormous amounts of work

If only we had a big number crunching machine and a way of defining lines, curves, ellipses, circles, polygons of all manner and size. We have both – a computer and mathematics. Archimedes is oft quoted as saying "give me a stick long enough and a pivot and I shall move the world", well in our instance give us the mathematics and a computer and we can produce any kind of picture you'd like.

Any straight line can be described mathematically as $y = mx + b$ where m is the slope of the line and b is the point at which it hits the y-axis, the intercept. Going back to our example this means we can describe our line $\overrightarrow{OA}$ in mathematics as:



This gets even cooler because this simple geometry (thanks Mrs McKenna!) means that if we know the start point and the end point of a line we can figure out the slope of it and fill in all the rest of the line. Let's do this for a new line:

What if we only know two points on the line? Say:

Point A $=\binom{1}{3}$ Point B $=\binom{5}{5}$

Can we work out the equation for this line $\overrightarrow{AB}$?

You bet we can, this is called the "point-slope" formula which says:

$$y - y_1 = m(x - x_1)$$

First we need m (the slope) which is:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \begin{matrix}(\text{The change in } y)\\(\text{The change in } x)\end{matrix}$$

$$m = \frac{5-3}{5-1}$$

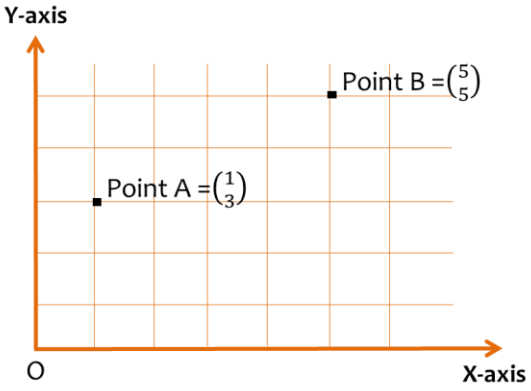
$$m = 1/2$$

Once we have the slope we can use either Point A or Point B to work out the equation. Lets use Point B = $\binom{5}{5}$:

$$y - 5 = m(x - 5)$$

So our equation is:

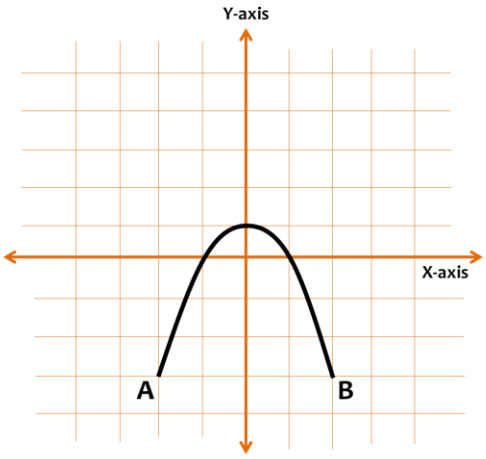
$$y = \frac{1}{2}x + 2\frac{1}{2}$$



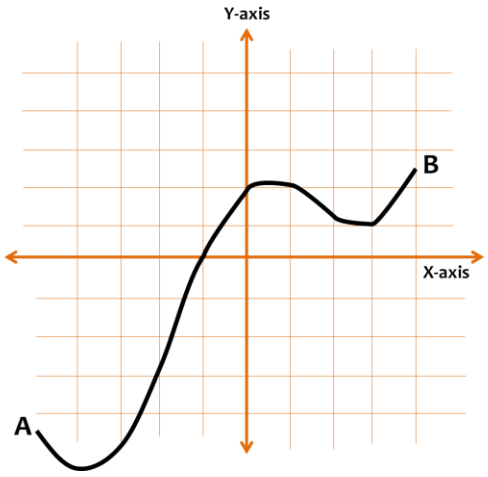
What does this all actually mean? Well put simply it means that we can avoid a lot of onerous work on straight lines now. We can simply state the end points and write a simple computer program to work out the slopes and the equations for us and fill in all the rest of the points.

The mathematics fun doesn't end there though we can describe curved lines this way too, here's some examples:

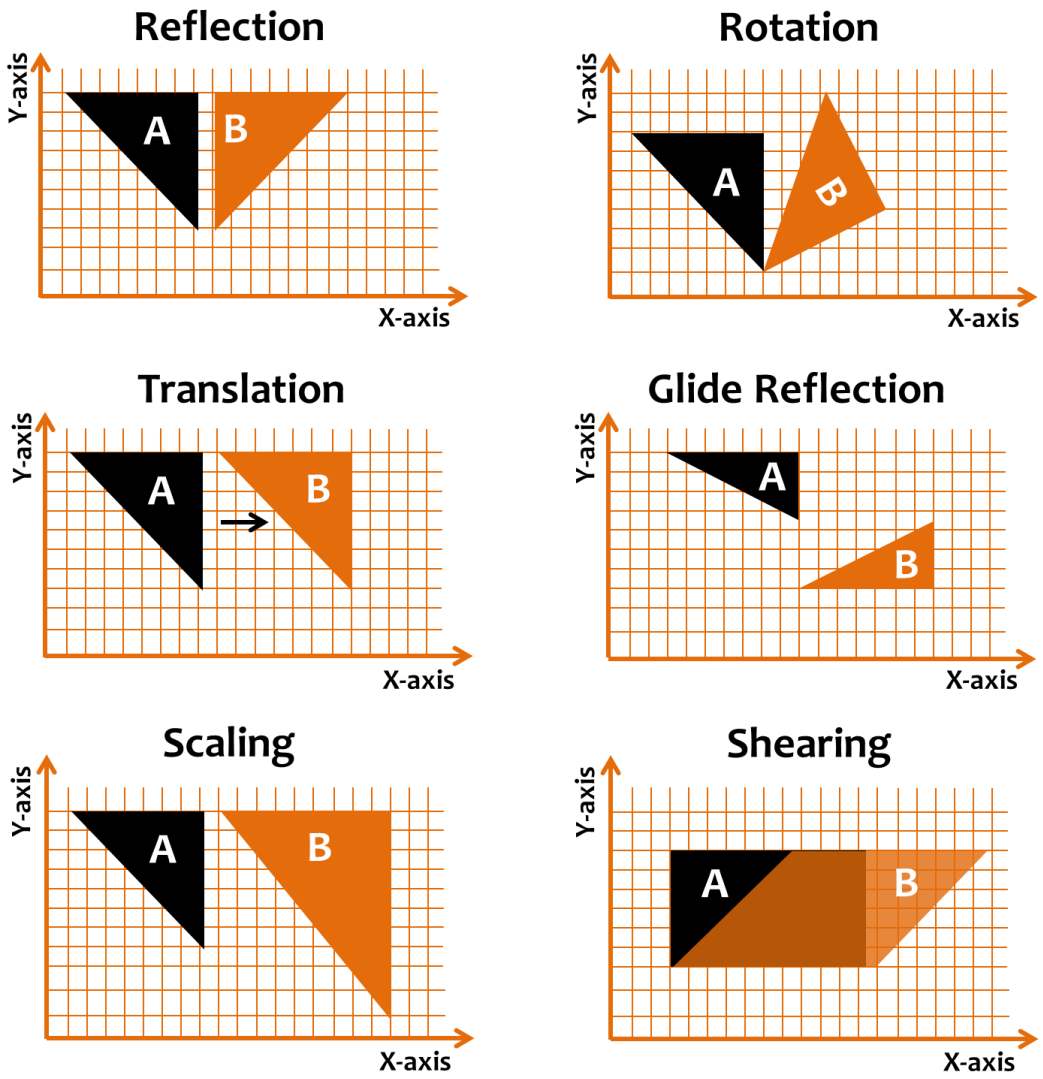
Equation for this Line $\overrightarrow{AB}$ $y = 1 - x^2$



Equation for this Line $\overrightarrow{AB}$ $y = x + 2\cos(x)$



It didn't take too long for computer graphics researchers to realize that dusting off the old geometric mathematics knowledge (Euclid, Descartes, Desargues, Hilbert and many more) and writing algorithms to generate pixels would propel forward the capabilities of computer graphics. These algorithms could do far more than simply drawing a line or a shape delivering extraordinarily efficient mechanisms for:



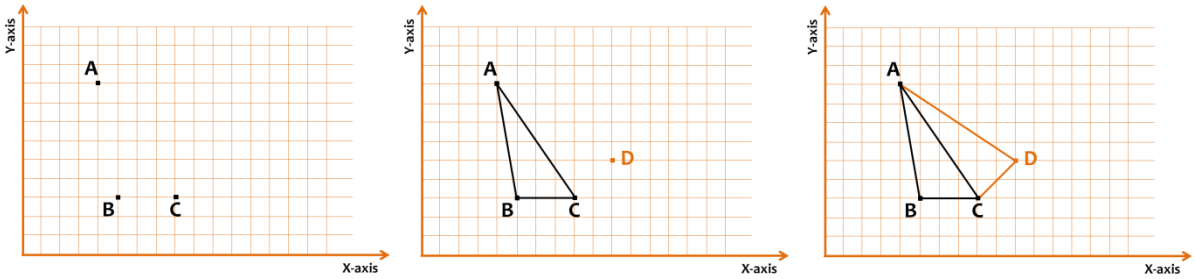
The benefits of this geometric approach coupled with ingenious algorithms that leveraged the power of a computers number crunching could do all manner of operations based on knowing the points or vertex's:

- Draw, reflect, rotate, transform (move), and scale lines & curves;
- Draw, reflect, rotate, transform (move), and scale shapes (polygons, circles, ellipses, etc.);
- Solid fill (basically add a colour in between the lines the shapes;
- Pattern fill (super impose a checkerboard or other pattern to the human eye) shapes;
- Clipping (cut a slice out of) lines and shapes; and of course
- Composites of more than one of the above.

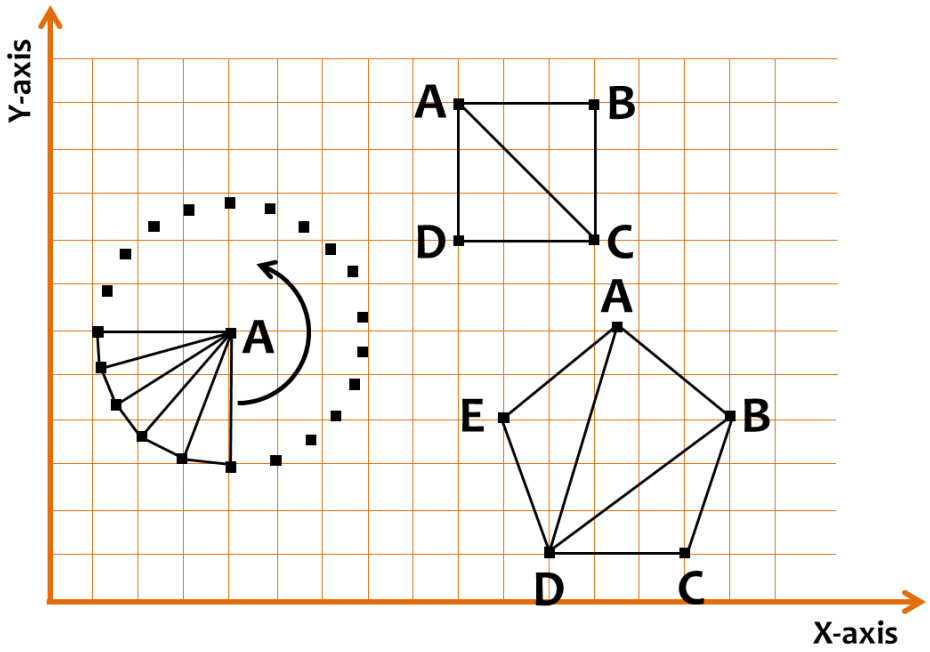
3.1.1 Return of the Humble Triangle

The final piece of the mathematics (promise!) brings us briefly back to the simplest of shapes (okay arguably a line is simpler shape but a line has no filling or volume) the triangle. As we have seen earlier a triangle can be described by three individual vertices. Two triangles positioned side

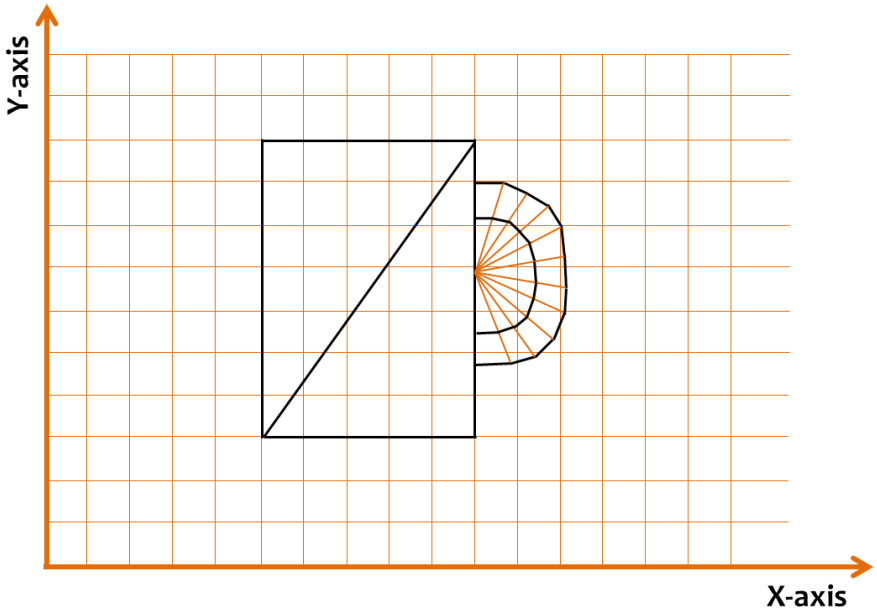
by side sharing an edge can be described by the addition of only one more vertex to create two more lines, visualized below (add point D to create another triangle):



We can also employ triangles to draw squares, rectangles, pentagons, other polygons we can even employ a fan technique to draw circles and curves using triangles – see below:

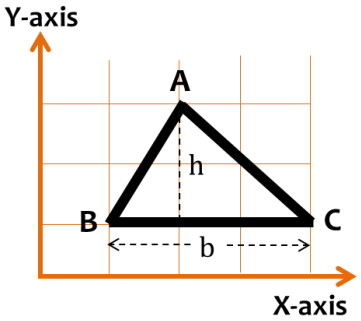


In addition to saving a lot of memory space, especially for large shapes, describing everything as triangles provides extensive processing efficiency. A number crunching computer can gobble up the work on triangles extremely effectively. For instance if we wanted to describe the surface of a complex object such as a cup (even in two dimensions) then the complex mathematics alone for the curved handle would push the processing upwards whereas using triangles we can draw the cup as:



Note: For simplicity this visualization shows 24 triangles in total with 22 of them the same but scaled to form the shape of the handle. In the real world we would use far more triangles and leverage anti-aliasing to create a smoother curve for the handle.

Another excellent advantage to triangles is that when we come to fill in surfaces we can work out the area of triangles extremely efficiently. Going back to school days the area of a triangle was simple:

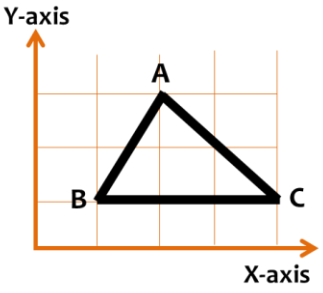


Equation for the area of the triangle is:

$$\frac{1}{2} b * h$$

b = length of the base
h = height of the triangle

Not bad but this requires calculation of the length of the base and the height. Another method is Heron's Formula:



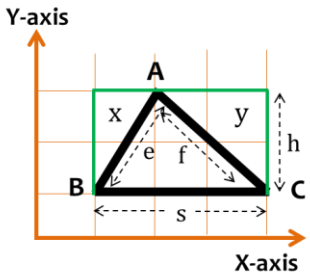
Heron's Formula for the area of the triangle is:

$$A(\Delta) = \sqrt{s(s-a)(s-b)(s-c)}$$

Where:

$$s = \frac{1}{2}(a+b+c)$$

Again not a bad approach but it requires as much if not more calculation and worse requires square rooting which is suboptimal for a graphic composed of many triangles. We needed a simpler method. A geometric approach could provide such a simple method:



A geometric approach involves drawing a rectangle around the triangle. This approach means that the area of the triangle is the area of the rectangle minus the two right angled triangles x and y:

$$\text{Area of ABC is} = (h * s) - (e^2 + f^2)$$

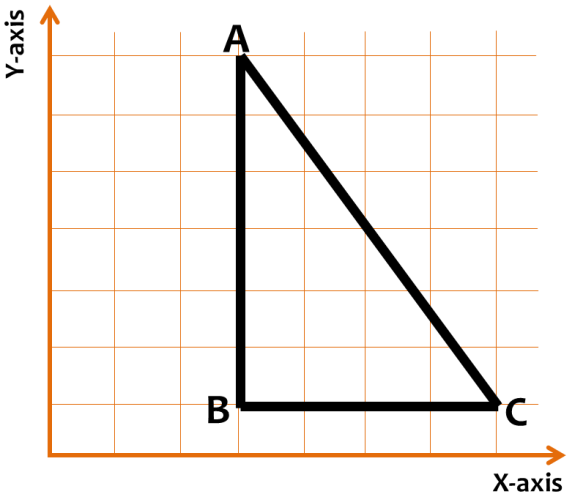
While a fantastically simple solution that does indeed solve the issues of square roots it now requires more compute to deliver the lengths of s, e, f, and h. What we needed was a method which uses simple addition, subtraction, multiplication and at worst division. Thankfully for all of us both Rene Descartes (1596 – 1650) and Pierre de Fermat (1601 – 1665) had foreseen the need in 1970 for computer graphics engineers to have such a simple method for working out the area of a triangle when we know the vertices A, B and C (okay that might be bit of a fib!). The method involves the linear algebra "determinant" of a square matrix. Basically the determinant "D" of a three by three matrix of any numbers is calculated as:

$$\text{Area of ABC is the positive value of } \frac{1}{2} Det|ABC|$$

Where the determinant of ABC is:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

"Hang on just a second" you're probably saying, "we don't have a three times three matrix of points. In our coordinate system we have only two points!" Let's work it out using a simple right angled triangle whose area is trivial for us to work out in advance:



Triangle ABC with vertices:

$$A = \begin{pmatrix} 3 \\ 7 \end{pmatrix}$$

$$B = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

$$C = \begin{pmatrix} 7 \\ 1 \end{pmatrix}$$

This triangle is not only right angled but we know easily its height (6) and base (4) so using the old fashioned method of $\frac{1}{2}(b * h)$ this gives us a known area of 12. So let's double check this with our determinant method:

$$\frac{1}{2}\begin{vmatrix} 3 & 7 & 1 \\ 3 & 1 & 1 \\ 7 & 1 & 1 \end{vmatrix} = \frac{1}{2}(aei + bfg + cdh - ceg - bdi - afh)$$

$$= \frac{1}{2}(3.1.1 + 7.1.7 + 1.3.1 - 1.1.7 - 7.3.1 - 3.1.1)$$

$$= \frac{1}{2}(3 + 49 + 3 - 7 - 21 - 3)$$

$$= \frac{1}{2}(55 - 31)$$

$$= \frac{1}{2}(24)$$

$$= 12$$

So we now know that: six simple multiplications, five additions and a division by 2, yields the area of any triangle when you know the three vertices. To a number cruncher that can be done fast, very fast.

3.2 Sources & Further Reading

http://vimeo.com/16292363

https://people.richland.edu/james/lecture/m116/matrices/applications.html

https://people.richland.edu/james/lecture/m116/matrices/area.html

http://en.wikipedia.org/wiki/Ren%C3%A9_Descartes

http://en.wikipedia.org/wiki/Pierre_de_Fermat

http://en.wikipedia.org/wiki/Determinant

Boreskov, A Shikin, E 'Computer Graphics: From Pixels to Programmable Graphics Hardware' (2013, CRC Press)

Shirley, P Marschner, S et al 'Fundamentals of Computer Graphics' (2000, CRC Press)

Foley, J (Editor) 'Computer Graphics: Principles and Practice' (1996, Addison-Wesley Publishing)

Govil, S 'Principles of Computer Graphics: Theory and Practice Using OpenGL and Maya®' (2004, Springer)

Klawonn, F 'Introduction to Computer Graphics' (2012, Springer)

Johnson, Amos 'Basic Concepts for Computer Graphics' (2007, iBook Store)

Ryan, D ‘History of Computer Graphics’ (2011, Authorhouse)

Cohen, J ‘Visual Color and Color Mixture: The Fundamental Color Space’ (2001, Google Books)

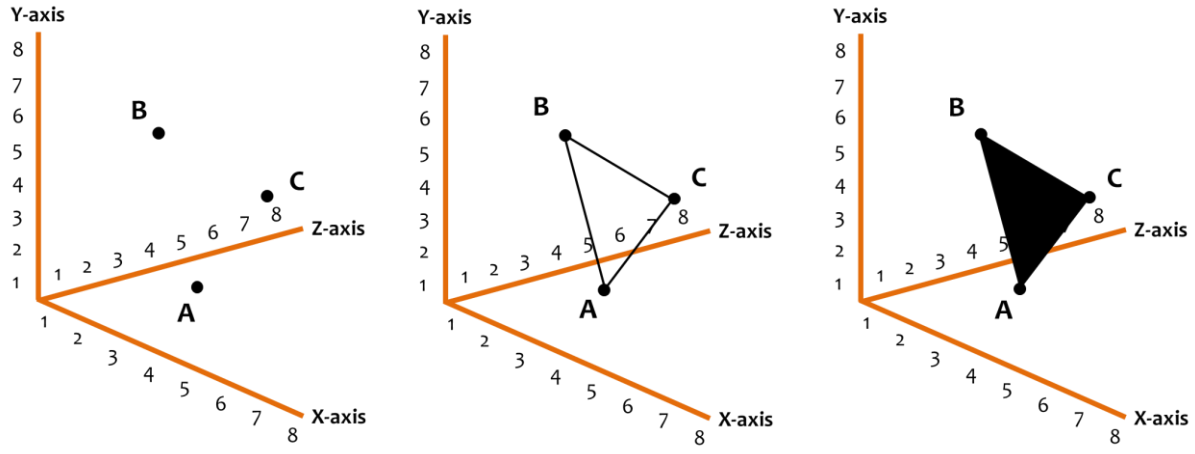
4. THE IMPACT OF THREE-DIMENSIONS (3D)

This paper has covered a lot of groundwork so far and it would seem an opportune moment to recant where we're at before delving into the world of further 2D and 3D graphics. So far on the journey this paper has outlined the basic evolution of computing and its importance for businesses in delivered large scale numerical calculations, introduced at a high level the basics of a CPU architecture that prevails to this day albeit with the recent introduction of multiple cores, identified an early need for visualization and interaction between humans and computers, realized this need through a Cartesian-like coordinate system using pieces of light called pixels and a matrix or grid structure as a representation of the display screen, juxtaposed to the employment of these developments in videogames, briefly reviewed the development of the videogaming and home console market and finalized with how that markets growth and hunger for newer, faster, better drove us to innovate to deliver rasterisation, framebuffers, anti-aliasing, polygons, vertex's, added a splash of colour, reviewed the mathematics involved and described a world of pictures made from humble triangles.

Back in the early 1970s the scene was set then for a major expansion in computer graphics in particular in three dimensions (3D). Ed Catmull and Fred Parke led the way with a ground breaking animated hand in 1972 which delivered a glimpse of what could be possible. In parallel with the push during the 1970s and 80s from videogaming and arcade companies the cinematographers were also pushing on the limitations of Computer Generated Imagery (CGI), notably in Westworld, Futureworld, Star Wars (all 1970s) , Tron, The Last Starfighter and Labryinth (1980s) to name but a few.

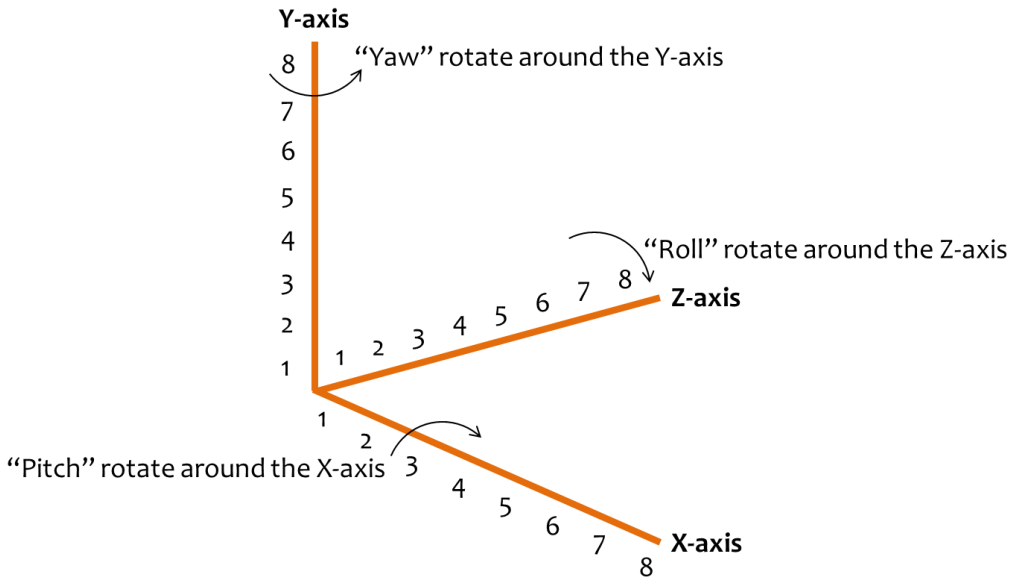
4.1 Introducing 3D

In a lot of ways to get from 2D and all the areas reviewed above to 3D is fairly straightforward. At the highest level of abstraction you simply add another plane to the x-axis and y-axis, call it the z-axis and you've got a 3D space:



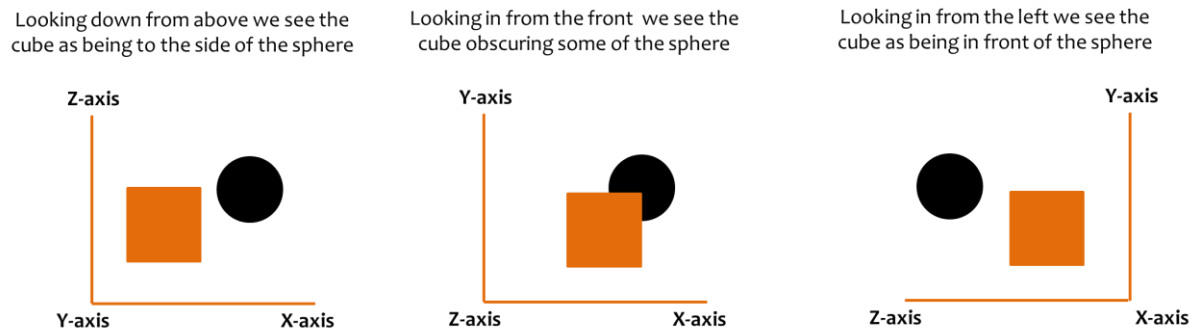
In this example the three vertices describe points in a 3D space at A = (5,3,1) B=(3.5,7,0) & C=(3,4.5,6) which gives us a triangular surface in 3D space. We can perform all of the operations we looked at earlier on this surface such as scaling it, rotating it, reflecting it, shearing it, etc. The mathematics gets more complex but in conceptual terms it remains the same. What does change significantly are rotation and transformation. In 2D we could chose a point or a vertex and rotate

around that point or vertex. What early 3D graphics specialists realized is that in 3D we have more options now:



The object could turn or rotate on any of the axes. In transform terms which direction are we now moving this triangle. Before we had left or right, up and down or a combination. Now we have left, right, up, down and of course back and forth. The combined effect of what we saw as a "Glide Transform" earlier is now far more complex with the ability to glide back, to the right and upwards simultaneously. Other significant differences driving the enhancement of 2D and the adoption of 3D was the need for greater levels of realism in images. This means working out techniques for shadows, lighting & multi-light sources, perception of depth, surface textures (especially hair), patterns, ordering and hidden surface removal. Hidden surface removal is a issue particular to 3D which is concerned with the obscuring of parts of surfaces and objects from the observers viewpoint. This is best visualized below:

Relative to the view of the observer you get a different view of a scene in 3D



This is an idealized view of two objects, the complexity increases when there are three, four five etc.

Before the reader gets worried I promised in sub-section 3.1.1 that that was the last of the mathematics and it is. It suffices to point out at this stage that in the Sources & Further Reading section many authors have produced comprehensive texts outlining the mathematics of 3D and from a non-mathematicians point of view "hairy" doesn't begin to do it justice. The mathematics

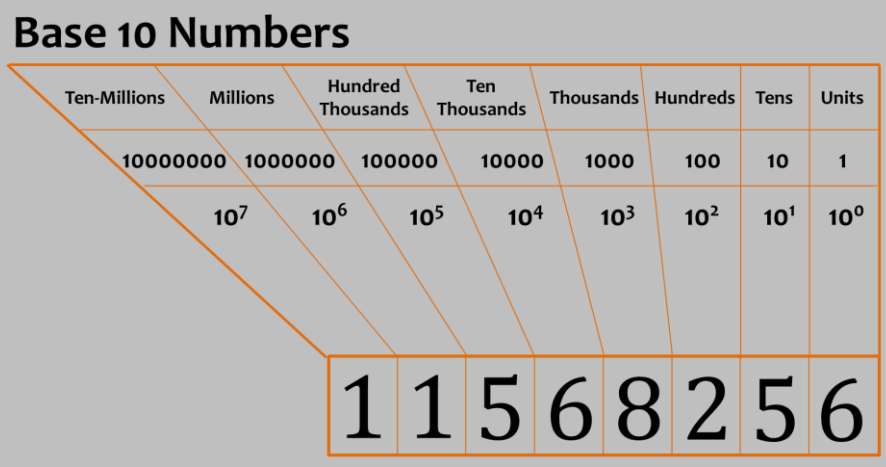
involved is incredibly deep and simultaneously brilliant. For the purposes of this paper we have what we need in Section 3 above - "triangles" and "mathematics".

A distinction needs to be made at this juncture between CGI for cinema and 3D for videogames. This distinction is huge. Both present significant challenges for computer graphics. The key differences from the point of view of this paper are:

- Cinema computer graphics seeks ultra-realism and take the time necessary to employ techniques such as Ray Tracing which actually seek to trace the path of millions of rays of light as they would bounce, reflect and scatter through the frame. This technique delivers incredible levels of reality but can take hours (even more than a day depending on the image) to produce just one frame;
- In stark contrast videogames need images that are as realistic as can be achieved within significant time constraints. They need frames in real-time, in fact they need 30 to 100 frames every second to produce seamless gameplay. That is an order of magnitude in difference. The goal of any videogame today (2013) is to get to the quality of Toy Story at 60 frames a second.

Quick Tangent on Floating Points & FLOPS

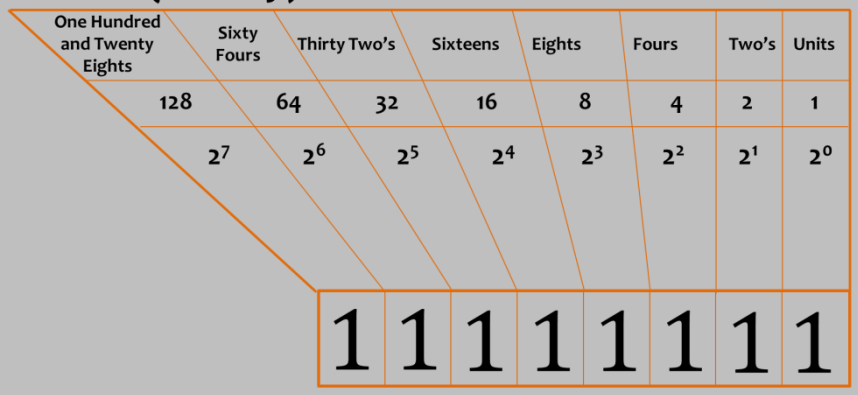
Up to this point there has purposefully been no mention of the types of numbers we're talking about when doing all the calculations for computer graphics. Let's handle it now. Firstly, computers work on binary numbers but we humans work in decimal world. For general use of a computer this is of little significance as the computer converts numbers to and from decimal for our convenience. For instance our day-to-day world of numbers is presented like this:



In our world we write the number ***"eleven million five hundred and sixty eight thousand two hundred and fifty six"*** as **11,568,256**

In the world of the computer numbers are presented in binary:

Base 2 (Binary) Numbers



In our world this sequence of 1's converts to the number ***"two hundred and fifty five"***. So in our writing **255**. So we needed 8 bits of information to represent the number two hundred and fifty five.

That's a lot of bits of information for such a small number.

What about our the 11,568,256 then? How would this look in binary?

2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0

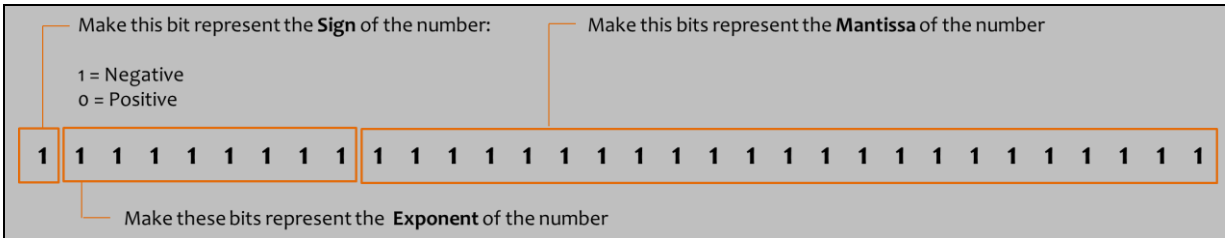
Wow this needed twenty four bits of data.

So following the logic through, given a lot of computers are only 32-bits so what's the biggest number they could hold?

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

4,294,967,175

So we can represent numbers in binary which is fine but while four billion is a large number but it's not anywhere near what could be described as huge, especially in finance terms. Another question is where do we put the decimal point to denote fractions, for instance say £10.55p – how would we represent the 55p? This is where the early computer designers got really clever. They developed **"Floating Point Numbers"**. The point of all the foregoing in this "Quick Tangent" was to highlight the need for floating point numbers. We needed a mechanism for doing money (especially) and beyond money for handling very large and very small calculations. This is where floating point numbers come in. Given the limitations highlighted above the early computer designers decided to approximate real world numbers as accurately as they could (admittedly at the loss of some accuracy) and accept a built in rounding error. To do this they split up the 32-bits seen above into sections:



This means we can deal with much larger numbers using exponents. What are exponents? Very simply this is expressing a number by those powers of 10 we saw earlier. For instance the numbers 1,000 and 1,000,000 can be expressed in exponent for as:

$$1000 = 1 \times 10^3$$
$$1000000 = 1 \times 10^6$$

In our example from earlier we hit a limit for 32-bits at the number 4,294,967,175 so a number such as 8,589,934,233 would have required 33-bits which was unachievable. Now we can use exponents :

$$8.589934233 \times 10^9$$

The floating point representation would be:

When it comes to money or decimal point numbers we can also use exponents. For instance the numbers 1234.567 or 0.1156:

$$1234.567 = 1.234567 \times 10^3$$

$$-0.1156 = 1.156 \times 10^{-1}$$

If that was it for exponents then it wouldn't be bad, but exponents are like the gift that keeps on giving. Exponent mathematics comes with a very handy set of laws for us to utilize in computing in general and in computer graphics. Using X to denote any number we can list the laws:

Law#1	$x^1 = x$	**Law#6**	$(x^m)^n = x^{mn}$
Law#2	$x^0 = 1$	**Law#7**	$(\frac{x}{y})^n = \frac{x^n}{y^n}$
Law#3	$x^1 = \frac{1}{x}$	**Law#8**	$x^{-n} = \frac{1}{x^n}$
Law#4	$x^m * x^n = x^{m+n}$		
Law#5	$\frac{x^m}{x^n} = x^{m-n}$		

For computers this is excellent because it tells us that if we express long numbers with exponent element we can simplify the mathematics. For instance in decimal world:

Multiply:

5,768 multiplied by 7,142

If we express both numbers in scientific notation using exponents:

$(5.768 * 10^3)$ multiplied by $(7.142 * 10^3)$ is the same as:

(5.768 multiplied by 7.142) * $10^{(3+3)}$

And for divide:

5,768 divided by 7,142

If we express both numbers in scientific notation using exponents:

$(5.768 * 10^3)$ divided by $(7.142 * 10^3)$ is the same as:

(5.768 divided by 7.142) * $10^{(3-3)}$

It doesn't look much but if we recall that every floating point number is already:

i. Expressed in the computer as a binary number; with
ii. A binary representation of its exponent

then it means we can:

i. Binary multiply or divide; and
ii. Binary add and subtract the exponent values.

In this way a whole new world of precision operations for working with the very small and the very large can be achieve within the 32-bit constraint. Today's computers are now 64-bit and the nomenclature is to call 32-bits a "Short Real" and 64-bits a "Long Real". For computer graphics world there are two significant considerations to bear in mind:

i. The shear amount of calculations
ii. The precision of those calculations

Alvy Ray Smith is quoted as saying "reality is 80 million polygons" and in our world of computer graphics while not quite at that level it's still 1 million or more triangles for every frame. This means an overwhelming amount of floating point calculations per second which brings us nicely to the measure used to understand the speed of GPU's which is Floating Point Operations per Second (FLOPs or often FLOPS) which differ entirely from the old fashioned measure of the speed of a computer which was Instructions per Second (IPS).

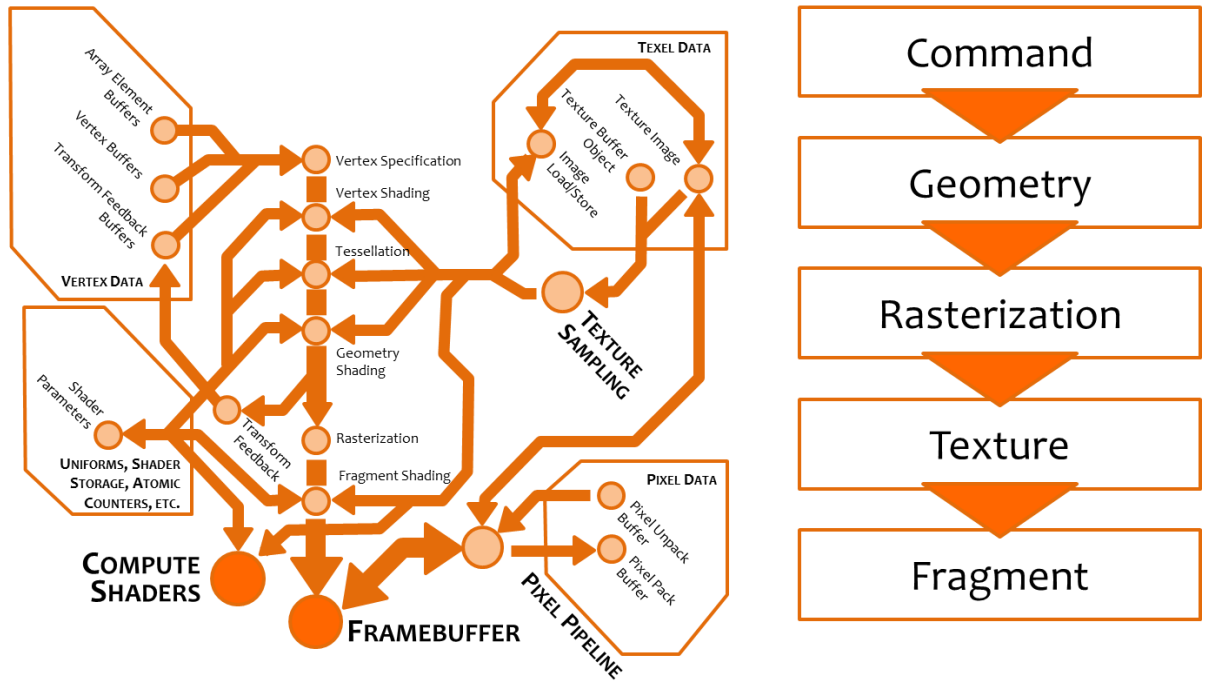
4.2 Introducing Rendering & the OpenGL Pipeline

Computer graphics through the 1980s became the eminent domain of Silicon Graphics Inc (SGI) who from venture funding in 1982 became the global leader in graphics with their Integrated Raster Imaging System (IRIS) series machines. These machine offered developers' access to the IRIS Application Programming Interface (API) Graphics Language (GL). An alternative open standard called PHIGS was adopted by Sun Microsystems, Hewlett Packard and IBM. By the late 1980s the market clearly needed a standard for computer graphics that developers the world over could leverage to be able to program graphics that would run on many different hardware platforms. Indeed, the requirements were such that if computer graphics didn't come up with a standard and a means of abstracting the complexity we reviewed above the whole field was at risk of becoming too niche which would slow development and we wouldn't have got anywhere near the likes of Doom, Quake, and Medal of Honour to name but a few.

In January 1992, SGI released OpenGL which was based on IRIS GL but OpenGL's code was entirely available for review and critically for extension. The core concept brought to bear with the creation of OpenGL was that of the rendering pipeline. Rendering is the process of generating an image or frame from a mathematical model. There are innumerable publications covering the aspects of the OpenGL pipeline in extensive detail listed below. The remainder of this Sub-section will provide a more abstracted view of the rendering pipeline in order to relay the concepts as opposed to many approaches that focus on the ability or desire to program. The second lecture in the Udemy series "Introduction to Graphics Architecture" by UC Davis is highly recommended.

OpenGL takes a certain approach to rendering an image. That approach could have been very different from the final pipeline. In totality this pipeline or sequence of events can be visualized below. The sub-sections below provide more information as to what precisely those operations are.

An Abstract View of OpenGL4.4 Pipeline



4.2.1 Command

The command components of the OpenGL pipeline are concerned with:

- Identifying and buffering the commands to be run
- Interpreting the commands
- Unpacking and performing format conversions
- Setting and maintaining the graphics state

4.2.2 Geometry

This is where much of the work is done including:

- Vertex specification – checking the attributes of each vertex are extant against the vertex array object (VAO) stored in the vertex buffer object (VBO);
- Transformation and projection
- Clipping, culling and primitive assembly
- Tessellation – the subdivision of patches of vertex data into smaller primitives ;
- Lighting computed per vertex

4.2.3 Rasterization

Building the raster:

- Set up per triangle
- Sampling
- Interpolation (deciding the fill colours between the end points)

4.2.4 Texturing

Now the colours of each fragment are known we can apply a texture – this is very compute intensive and involves overlaying a chosen texture onto the fragments. This is done fragment by fragment and either a new computed value for the fragment is computed or no new value is required:

- Texture transformation and projection
- Texture address calculations
- Texture filtering

4.2.5 Fragment

Final image aspects:

- Texture application
- Tests – Owner (test whether the pixel is within the ownership of OpenGL), scissor (if we're working in a certain window then the rectangle that is the window is the only valid display area to render to – other fragments are discarded), depth (compare the depth to the depth of the sample being written to), alpha (combining image with background) and stencils (comparing values with the stencil buffer and discarding the fragment if necessary);
- Blending (take the fragments output from the fragment shader and combine with the colours in the colour buffers);
- Dithering (creating the illusion of colour depth by intentionally applying a random error effect) and logical operations
- Final frame build.

4.3 Sources & Further Reading

http://en.wikipedia.org/wiki/Ray_tracing_%28physics%29

http://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29

http://www.sgi.com/company_info/overview.html

http://cs.wellesley.edu/~cs110/lectures/M01-color/graphics.pdf

http://en.wikipedia.org/wiki/OpenGL

https://www.udemy.com/discover

http://en.wikipedia.org/wiki/Floating_point#Floating-point_arithmetic_operations

http://kipirvine.com/asm/workbook/floating_tut.htm

http://www.binaryconvert.com/result_float.html?decimal=045046049049053054

http://en.wikipedia.org/wiki/Exponentiation

http://www.doc.ic.ac.uk/~eedwards/compsys/float/

http://en.wikipedia.org/wiki/Instructions_per_second

http://en.wikipedia.org/wiki/FLOPS

http://www.top500.org/blog/lists/2013/11/press-release/

http://www.opengl.org/registry/doc/glspec44.core.pdf

Glassner, E (Editor) 'An Introduction to Ray Tracing' (1989, Morgan Kaufmann)

Pharr, M Humphreys, G ' Physically Based Rendering: From Theory to Implementation' (2010, Elsevier)

Shirley, P Marschner, S et al 'Fundamentals of Computer Graphics' (2000, CRC Press)

Foley, J (Editor) 'Computer Graphics: Principles and Practice' (1996, Addison-Wesley Publishing)

Govil, S 'Principles of Computer Graphics: Theory and Practice Using OpenGL and Maya®' (2004, Springer)

Klawonn, F 'Introduction to Computer Graphics' (2012, Springer)

Johnson, Amos 'Basic Concepts for Computer Graphics' (2007, iBook Store)

Ryan, D 'History of Computer Graphics' (2011, Authorhouse)

Cohen, J 'Visual Color and Color Mixture: The Fundamental Color Space' (2001, Google Books)

Overton, M 'Numerical Computing with IEEE Floating Point Arithmetic' (2001, SIAM)

5. THE DEVELOPMENT OF GRAPHICAL PROCESSING UNITS (GPUS)

From the perspective of this paper the key points of outlining a brief overview of the history of videogames, introducing computer graphics, describing CPUs, journeying through the fundamental mathematics of 2D graphics, highlighting the added complexities in the evolution to 3D and finally laying bare the functions and components of the OpenGL pipeline and their execution calculations & computations on the primitives of 2D and 3D graphics was to make clear that:

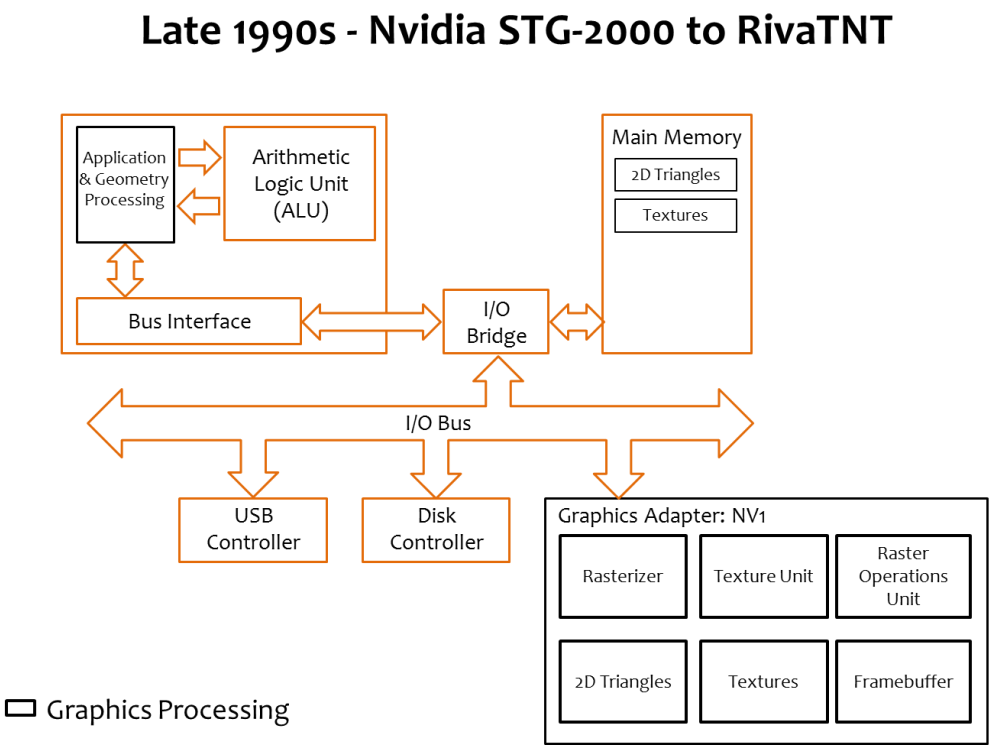
1) The market for videogames at first evolved in lock-step or just behind computing capabilities but quickly began to demand lead the technology;
2) Almost every aspect of the producing the final frame that we see on our screens involves the triangles and mathematics we reviewed earlier (albeit even more in depth and complicated);
3) In general every operation within the "pipeline" employs large amounts of floating point operations;
4) The growth in screen sizes required an increase in the computational ability due to the necessary increase in pixel numbers;
5) The introduction and greater use of colours also increased in the computational requirements due to the necessity to colour every fragment and pixel;
6) The evolution of the algorithms for anti-aliasing, blending, texturing etc. also increasing the requirements for processing; and
7) The desire in videogames to achieve near-reality levels of graphics to engage end customers and drive revenues and margins.

These demands (and more given the application usage that drives the requirement for display – for instance, Microsoft Excel enables the end user to create graphs, bar charts and complex pivot tables which all require processing and then displaying) were at the outset of computing being facilitated through the Von Neumann based CPU architecture described earlier. By the mid-1970s the graphics requirements alone were beginning to impact the application computing imperative for having a computer in the first place. It was becoming clear to computer manufacturers and engineers that a new way of handling the graphics element, or at the very least assisting the CPU in managing the graphics element was needed.

Early assistance was provided by the introduction of the video address generator chip from Motorola – the MC6845 – which generated the signals to interface with a raster display and although providing memory it did not actually generate the pixels. The Monochrome and Colour Display Adapter (MDA/CDA) cards in IBM's first PC's (1981) leveraged this Motorola chip. The use of co-processors to offload some specific work from the CPU was well trodden technology in this time period, for instance math co-processors were almost ubiquitous in PC designs so it was not long until the conceptual notion of using a type of video co-processor for graphics processing took hold. In 1983, Intel delivered the iSBX275 Graphics Controller Multi-module Board which leveraged the Intel 82720 Graphics Display Controller (GDC) specifically designed to drive raster-scan computer graphics – the first cost effective microprocessor to actually generate the raster display and manage graphics memory. This was quickly followed in 1985 by the Commodore Amiga 1000 which was the first PC to be released with a custom graphics microprocessor delivering a 640x400 screen resolution with 16 display colours from a palette of 4,096. It also feature (what they called) a custom animation microprocessor to enable high-speed animation.

As computer technology entered the 1990s the first need was greater than ever for graphics processing with the recent introduction of Microsoft Windows 3.0 (1991) and the tremendously successful Windows 3.1 (1992). In response to the growing graphical requirement S3 Graphics introduced the S3 911A in June 1990 with a Windows accelerator delivering 256 colours. This evolution of hardware GUI acceleration for processing in 2D and even 3D continued until the mid-1990s when (in a similar way to the earlier noted impact on CPU performance in the pre-hardware accelerated days) the impact on CPU performance even with co-processors and graphics acceleration cards was clearly worsening.

In 1995, NVidia launched their first product the NV1 which was the first microprocessor product to integrate GUI acceleration, wave-table synthesis (digital sound), video acceleration, 3D rendering and precision game port into a single chip. In parallel ATI (founded in 1985 and had been delivering graphics acceleration card technologies to IBM and Commodore) released their Rage3D graphics card. Arguably both these products remained accelerators but they heralded the beginning of a major expansion in functionality. In particular the NV1 can be visualized as:

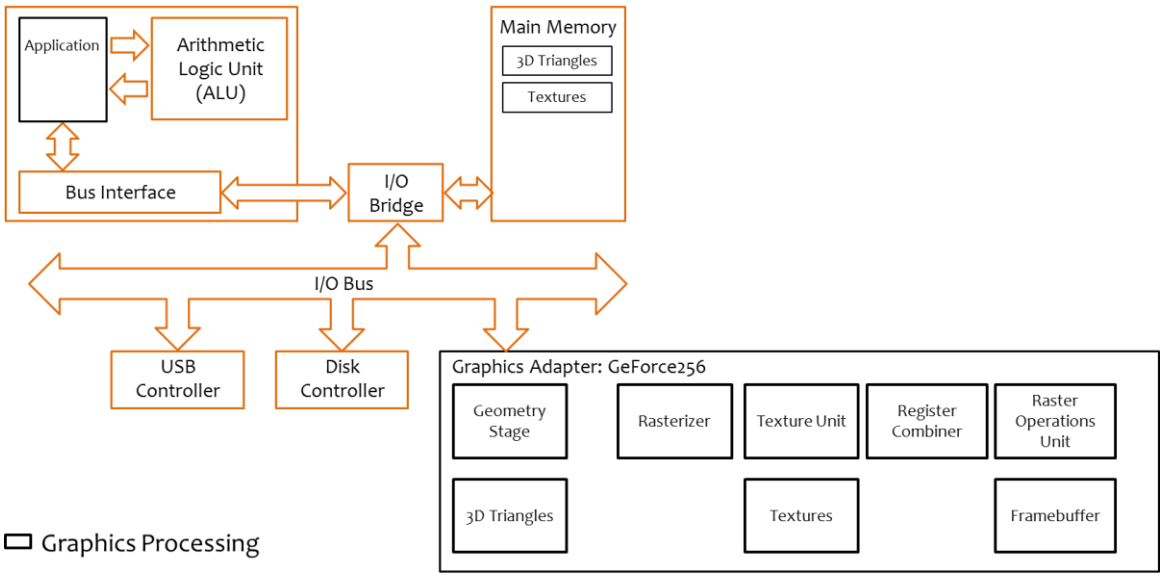


As we can see the NV1 cleverly took much of the latter stages of the work of raster, texture and framebuffer filling but the core geometric operations continued on the CPU, essentially meaning the NV1 continued to be an accelerator.

5.1 The Arrival of the GPU

In October 1999, Nvidia announced and launched the GeForce256 the world's first GPU. For the first time a single processor came into existence with integrated transform, lighting, triangle setup, clipping, and rendering. The chip was capable of 10million triangles per second. This fundamentally altered the architecture of computing:

Nvidia GeForce256



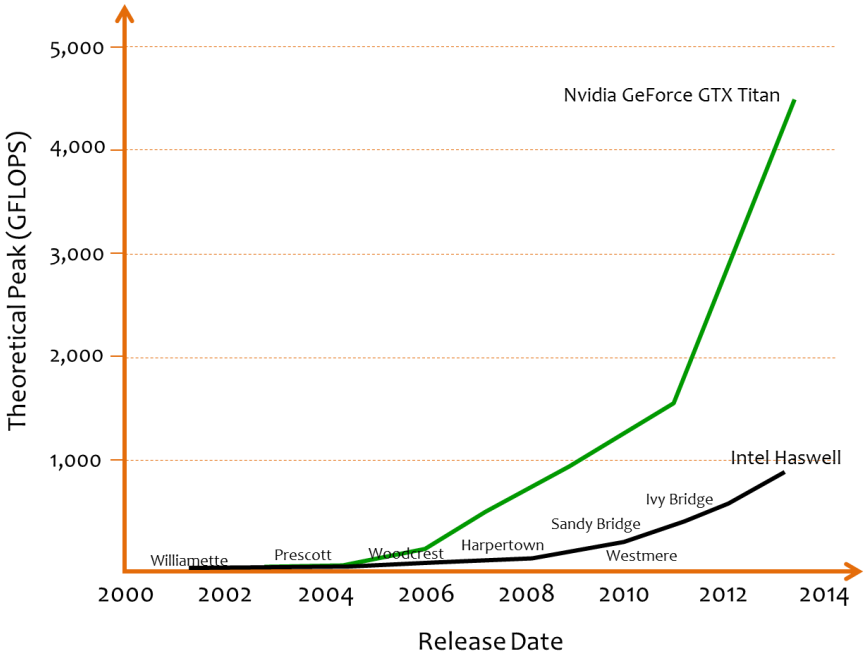
With subsequent evolutions, Nvidia added hardware functionality and programmable components which by 2006 delivered GPU hardware that carries out the following tasks:

- Pixel shading;
- Multi-texture;
- Programmable vertex shading;
- Bump mapping (instead of interpolated vertex) to compute lighting per pixel;
- Cubic texture mapping;
- Projective texture mapping;
- Volume texture mapping;
- Hardware shadow mapping;
- Anti-aliasing – super-sampling and multi-sampling;
- Multiple vertex and pixel shaders;
- Programmable pixel shaders;
- 64-bit colour;
- 64-bit floating point processing;
- High dynamic range imagery; and
- Real time tone mapping.

5.2 A Deeper Understanding of CPU's

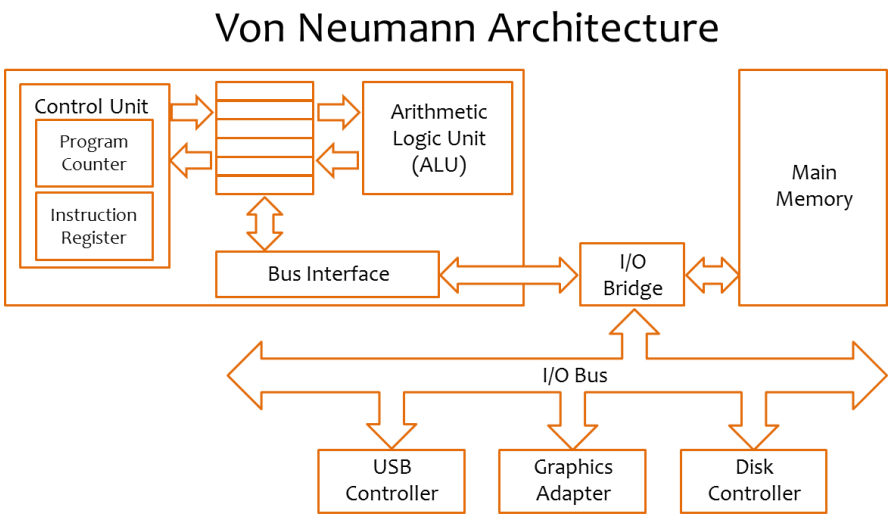
When Intel introduced the multi-core processor in 2005 it provided the impetus needed by general purpose computing for faster processing. Already pushing at the physical limits of Moore's Law the mechanism applied to drive more computing was the introduction of multiple cores within the CPU. This breathed new life into the CPU and chip designers and fabricators

could work towards core multiples as a mechanism for speed enhancement and while not quite linear every additional core delivers increases in performance. Still though the GPU was running away with the raw floating point speed battle:

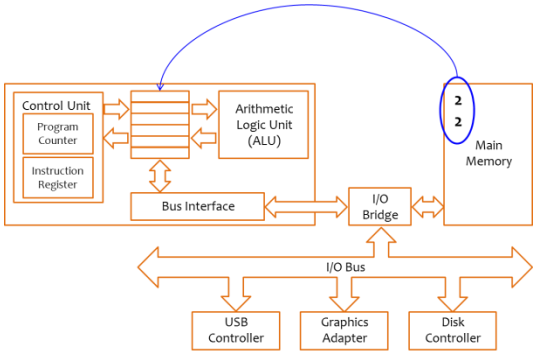
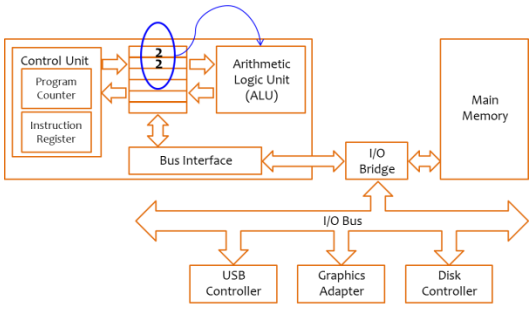
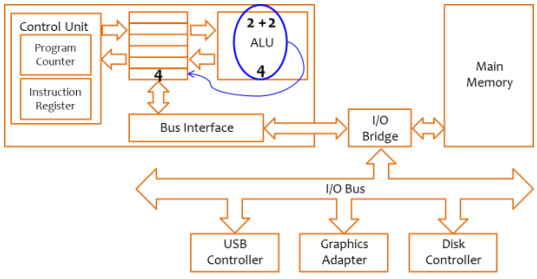
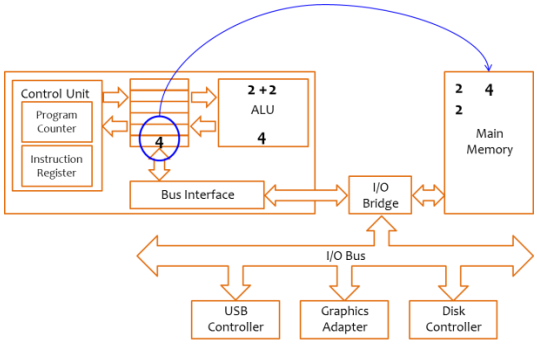


It's easy to think "well if this is the case then why would anyone ever invest in Intel CPU's? Clearly we should all go to Nvidia as soon as possible." If only it were that simple. The fact is that for pure floating point computation the GPU is clearly better, but remember from earlier it sees the world as triangles, polygons & floating point numbers and consequently is highly optimized for this task with a specific architecture. In contrast the CPU is designed for general purpose computing the number crunching, database searching and batch runs we saw much earlier. The key point is that the CPU can handle everything that the GPU can do (albeit more slowly) but the GPU ***cannot*** handle everything a CPU could do but does do triangles, polygons and floating point operations vastly more quickly.

To understand how dramatically different the two architectures are we need to return to the Von Neumann architecture we saw in Section 1 above with a slightly deeper view:



Let's get a little deeper and walk through a simple example of adding two numbers to help visualize this process (we're going to ignore binary representation but clearly the numbers would be in binary):

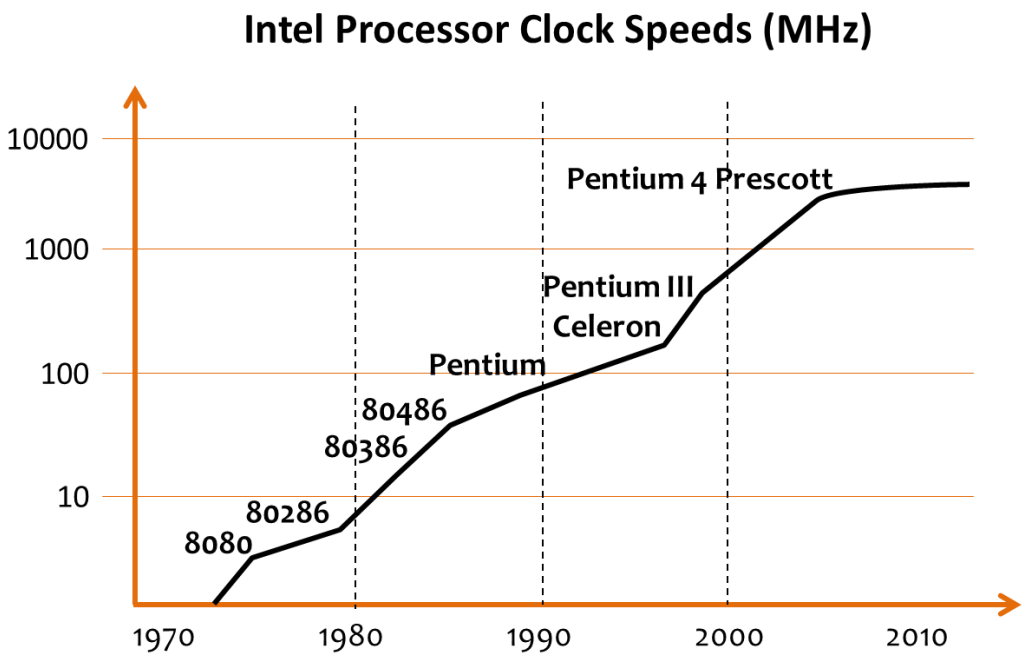
English	Pseudo Machine Code	Visual
Load the two numbers;	1) **Fetch** the first instruction (call it) #1 and load it into the instruction register and increment the program counter; 2) **Decode** the instruction in the instruction register; 3) **Execute** the load using the memory access hardware; 4) **Fetch** the second instruction #2 and load it into the instruction register;	
Read the two numbers in the registers into the ALU;	5) **Increment** the program counter; 6) **Decode** the instruction in the instruction register; 7) **Execute** the load using the memory access hardware; 8) **Fetch** the third instruction and load into the instruction register; 9) **Increment** the program counter;	
Add the two numbers in the ALU; and Write the result into a register;	10) **Decode** the instruction in the instruction register; 11) **Execute** the add instruction in the instruction register in the ALU; Read in the registers with the numbers, add them and put the output in a new register; 12) **Fetch** the fourth instruction and load into the instruction register; 13) **Increment** the program counter;	
Write from the register back into main memory.	14) **Decode** the instruction in the instruction register; 15) **Execute** the store using the memory access hardware.	

That's a lot of work just to add two numbers but it is very quick. The key points for the reader here are:

- There is a special register denoted the instruction register;
- Instructions are decoded before execution;
- A program counter is used to track the progress through the execution of the program;
- A CPU runs on a clock cycle with each pulse of the clock determining the events;
- Instructions are executed sequentially – unless the programmer has introduced a branch in the program. A branch is best described as a jump to another piece of code in the program for a given reason. When this happens the Control Unit loads the instruction register with the instruction identified by the branch instruction (think of a `GOTO` in programming terms);
- The sequence above takes four cycles to complete in lock step with the English.

5.2.1 Speed up the Clock

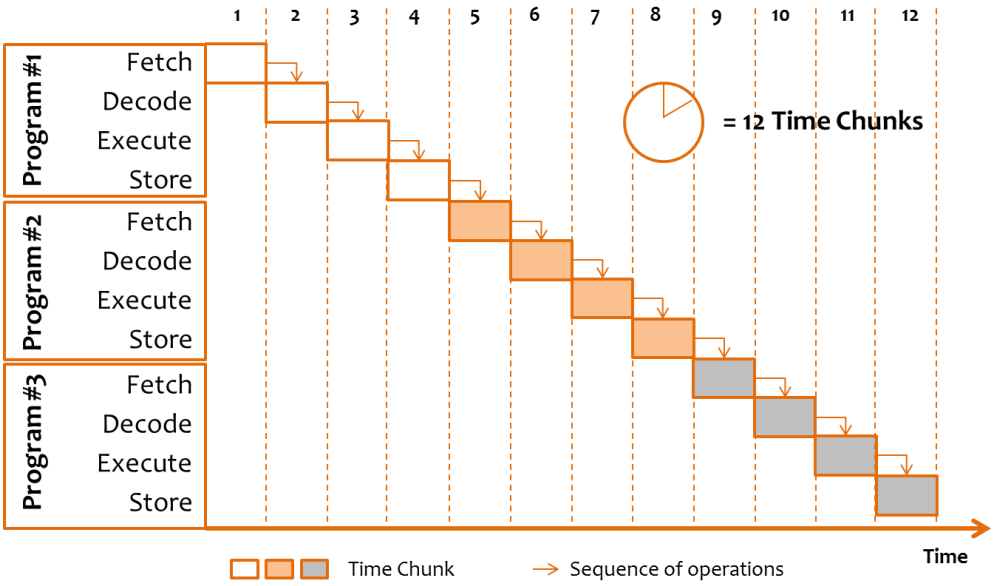
As we saw this architecture has a flaw in that the CPU can only work on one instruction at a time with each instruction fetched from memory and then executed. This deep dive allows us to cast our minds back into those of the original computer scientists and forensically examine (as they did) what could be done to enhance or speed up this execution process. No prizes for guessing the easiest way is to speed up the clock which is what everyone did from the 1980s through to 2005:



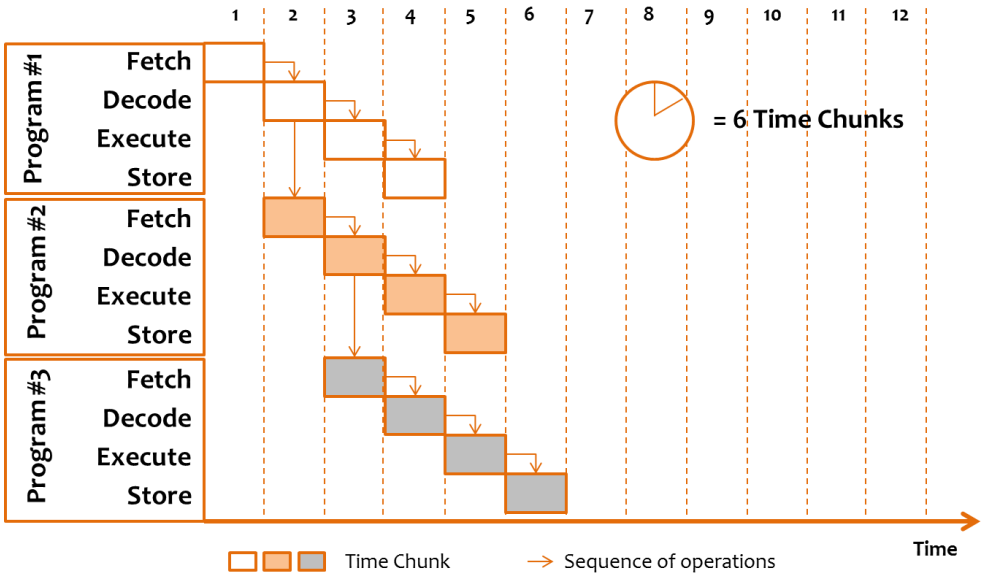
As you can see from the graph in 2005 we hit a problem and that problem is insurmountable. It is the physical limit of the chip itself. By 2005 the speed was so high and the channels for electricity to travel so narrow that we simply could not continue to evolve on clock speed alone. A new paradigm was needed. That paradigm was multi-cores.

5.2.2 Pipielining

Before covering the multi-core solution there remain a few items that were evolving in CPU's during the same period as clock speed. Clock speed was one answer to the "how do we do it faster?" question. Another answer is pipeline execution. In the example above its clear that the same things are happening quite regularly, specifically **Fetch**, **Decode** and **Execute**. This can be visualized as:



From a time perspective the problem is one of sequencing or more precisely the requirement for the CPU to work sequentially when processing instructions. This was not lost on computer engineering pioneers and they realized that a method could be put in place (pipelining) to shave time off the cycles by getting the next Fetch instruction to run in parallel with the Decode of the current instruction. This is much clearer visually:



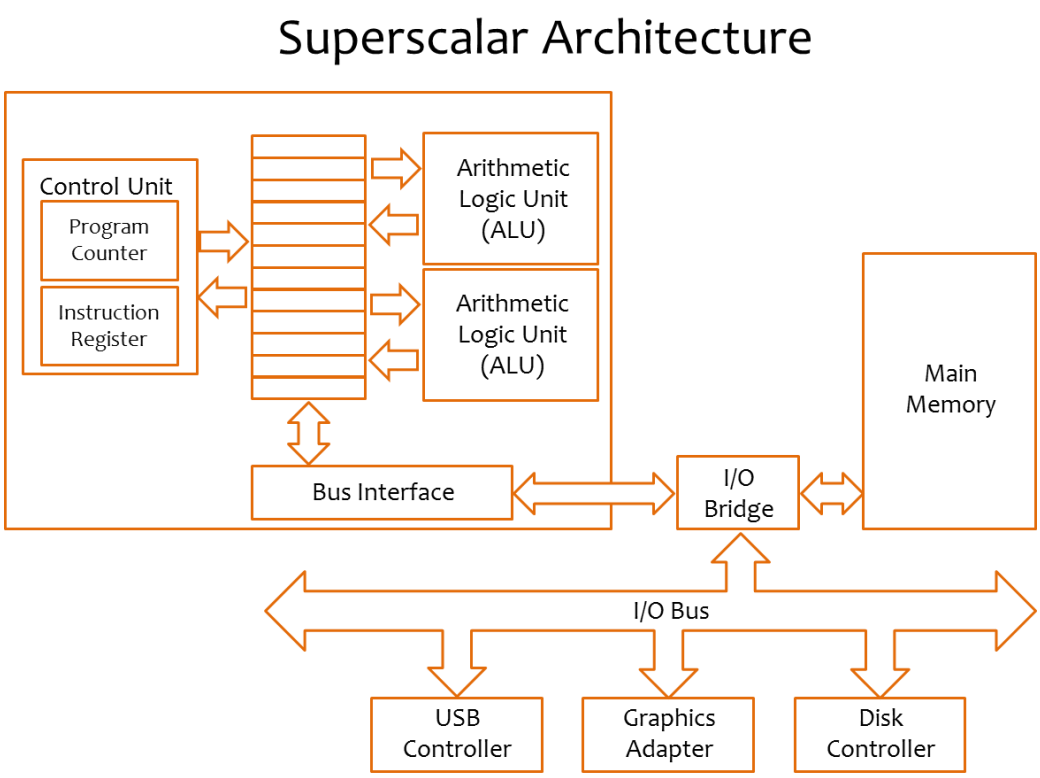
This highlights some significant areas to consider:

- Serial vs. Parallel – the ability to run certain tasks in parallel reduces the elapsed time for the group of programs (in the example 12 time chunks vs. 6);
- Program time remains the same – it is worth noting that every instruction took the same amount of time to actually run i.e. 4 time chunks. So the overall effect was that the three programs ran quicker in elapsed terms but each still took the same amount of time chunks to execute their instructions.

So now we have two levers to improve performance: the clock and pipelining. In the case of pipelining theoretically we could keep splitting tasks such as Fetch into sub tasks and refine the pipeline to have 8, 16, etc. stages to increase the parallelism and shorten the overall elapsed time to complete groups of programs thereby increasing the speed of processing in our CPU. But as we all find out in life "there's no such thing as a free lunch" and in the same way that we hit a hard physical limit to increasing the clock, we also hit a limit of the amount of pipelining we can do. The reason for this is that we've also increased the complexity in regards to the Control Unit logic required to orchestrate all of this pipelining.

5.2.3 Superscalar

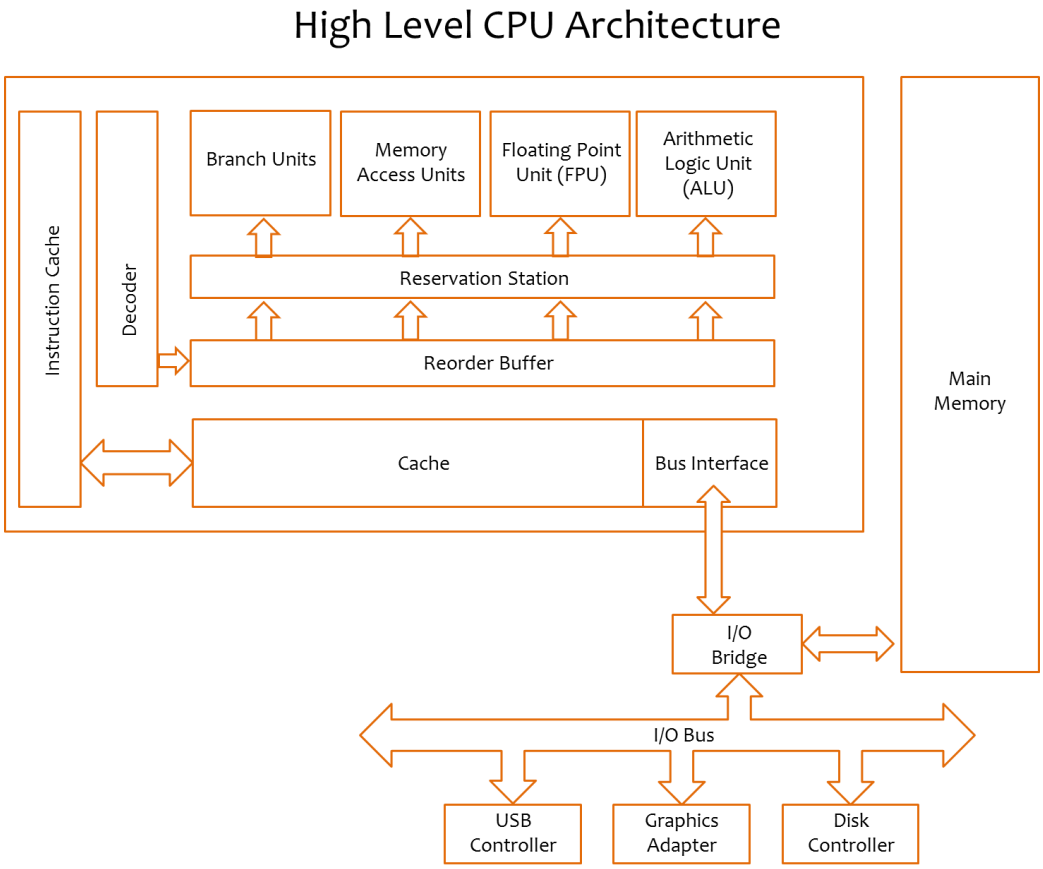
While engineers were working on clocking and pipelining the ability to cram more logic onto a chip was increasing. So far we've considered pushing the boundaries with a single ALU but engineers realized that they could put more than one ALU onto the chip and execute two adds, multiplies, etc. in parallel – this would mean more than one scalar or number operation at a time and became known as a superscalar chip. In 1990, IBM stepped up to deliver the RS6000 which was the world's first superscalar CPU. Amending our Von Neumann architecture to be superscalar reveals:



Designers also noted that different types of operations could also be achieved by designing specific logic to handle integer (rounded numbers), floating point numbers, memory accessing

activities and many more. One of the key problems with adding more ALUs and FPUs in addition to needing more registers and memory is again sequencing. For instance if Progam#1 has code that says ADD A, B & C and Program#2 says ADD C, D & E. In this case we can't execute the ADD instruction for Program#2 until Program#1 has provided the value for C. If we did try the program would crash as its out-of-sequence. This means complexity and it means more logic in the hardware to control and orchestrate everything that we want to achieve.

All of this complexity required much more control logic in the hardware to coordinate the activities of pipelines, memory access, and to prevent out-of-sequence instructions. A reorganization of the CPU was required. So a little bit more realistically yet still at a highly simplified level, the mid-200s CPU architecture could be viewed as:

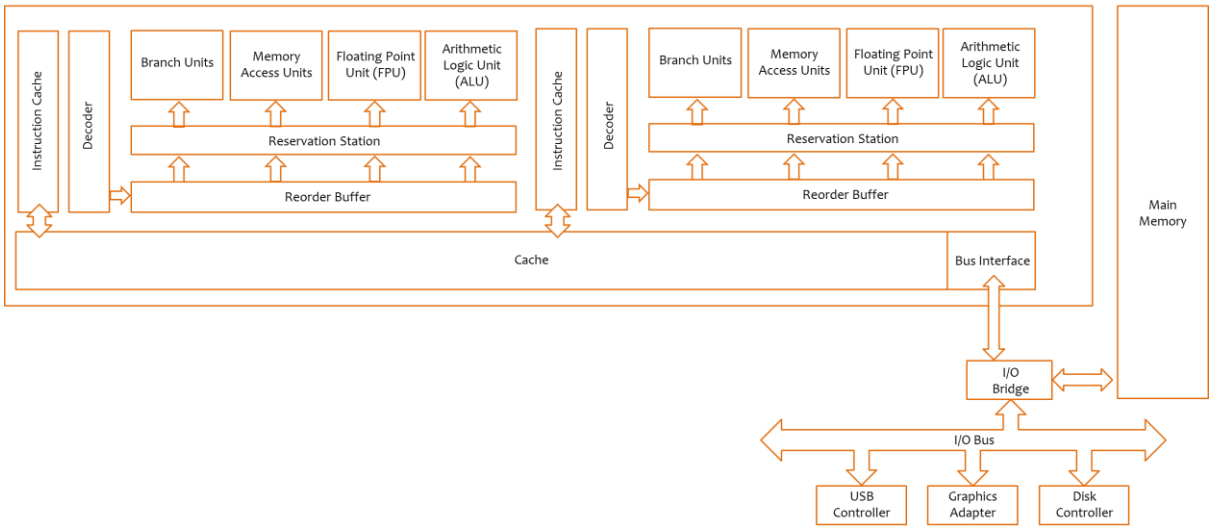


The new components all play a part in speeding up the processing capabiltiies:

- Caches – provides an on chip area of memory which speeds the process of retrieving data or instructions to be processed;
- Reservation station – provides a capability to rename register entries in order to do dynamic instruction scheduling;
- Reorder buffer – we identified earlier that it is impossible to re-order or execute certain instructions out of sequence and this of course holds true. However there are many instructions which could be processed out-of-order and this leverages the reorder buffer to do so;

As with the no free lunch earlier all this complexity was having an impact and the miniaturization of components was reaching a limit. This led to the creation of multi-core or chip multi-processing configurations:

High Level Multi-Core CPU Architecture



As can be seen (and was most likely expected) the evolution in complexity from where we began several pages ago to the modern dual, quad or more core CPUs is staggering and this paper is only really covering the high level conceptualization of these capabilities. There remains one last area to be clear on before returning our attention to the GPU.

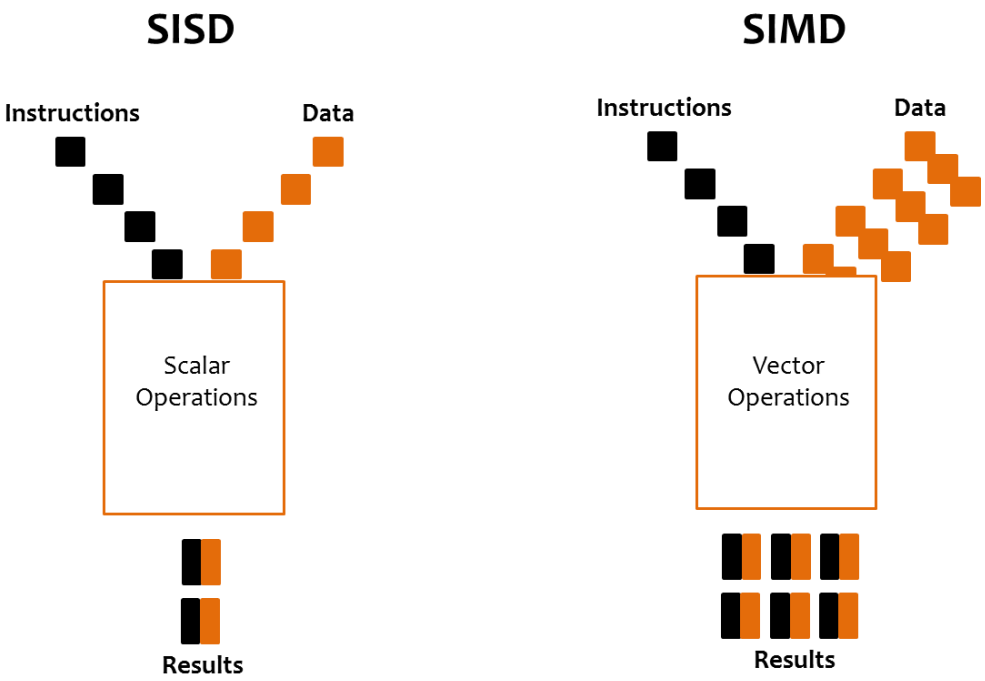
5.2.4 Multi-Threading

Briefly mentioned earlier in Section 1 "Threading" requires further consideration. Threads are comprised of multiple instruction streams or sequences of instructions within a computer program which can be executed independently of each other. Essentially the main program can be split into threads and each thread can run independently being cached, pre-fetched and executed once the processor is available. This again is a means for the CPU to introduce parallelism in pursuit of increased speed and execution of the program.

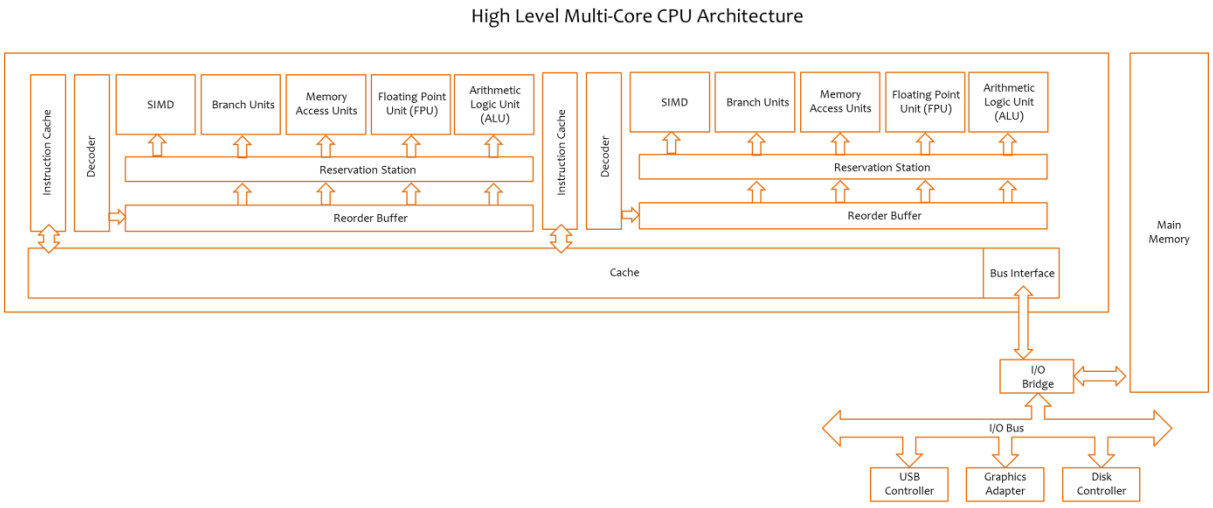
Threads exhibit the following characteristics: they have their own program counter and their own set of registers, but they are a software mechanism for achieving parallelism. The purpose of mentioning threads within this paper is simply for the reader to be i) aware they exist and ii) how they can take advantage of both multi-core machines and superscalar architectures to gain throughput.

5.2.5 Single Instruction Single Data(SISD) & Single Instruction Multiple Data(SISD)

The program example in the table at the outset of this sub-section provided a simplistic view of the FETCH->DECODE->EXECUTE->MEMORY->WRITE operations on scalar or integer numbers – in this case an ADD. Many numbers though are not simply integers or floating point numbers. A good example is our triangle vertex in 3D space where the point in space is described in our three coordinate axis x, y & z. So to identify the point we needed to have *Point = (2, 2, 2)*. If we wanted to carry out a "Transform" operation on this with a standard CPU we would need to sequence a set of instructions to ADD (let's say 2) to each of the three spatial coordinates to give us *transform point = (4, 4, 4)*. This would be a lot of work and a lot of time and that is because the standard ALU we have spoken about thus far was a Single Instruction Single Data (SISD) ALU. What engineers needed was to be able to manage vectors, matrices and other multi-entry numbers in a much more efficient manner. This became the Single Instruction Multiple Data (SIMD) ALU which can use the fact that the operations on the entries in the matrix or vector can be carried out in parallel. This can be visualized as:



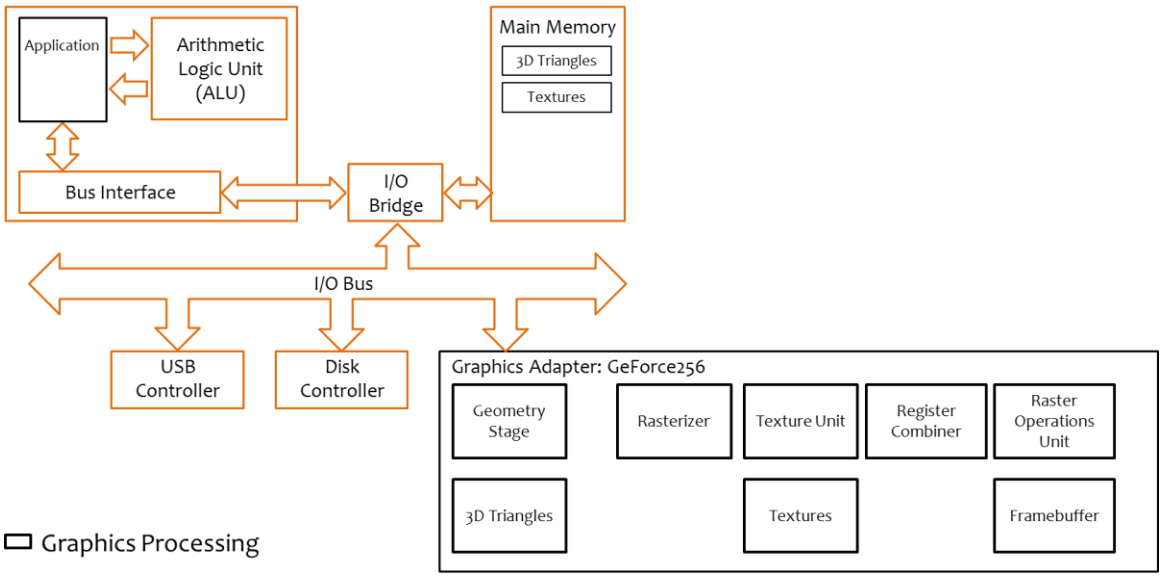
Our triangles from Section 3 above are perfectly suited for SIMD execution where a single instruction can be applied to all the elements simultaneously. In fact almost all floating point calculations can gain from the use of SIMD. Because SIMD works on matrices or vector numbers it is often called vector processing. Because of the extensive speed enhancements from the SIMD Intel added to their chip sets with Streaming SIMD Extensions (SSE) to carry out vector processing. So our final view of a multi-core CPU can be visualized as:



5.3 A Deeper Understanding of GPU's

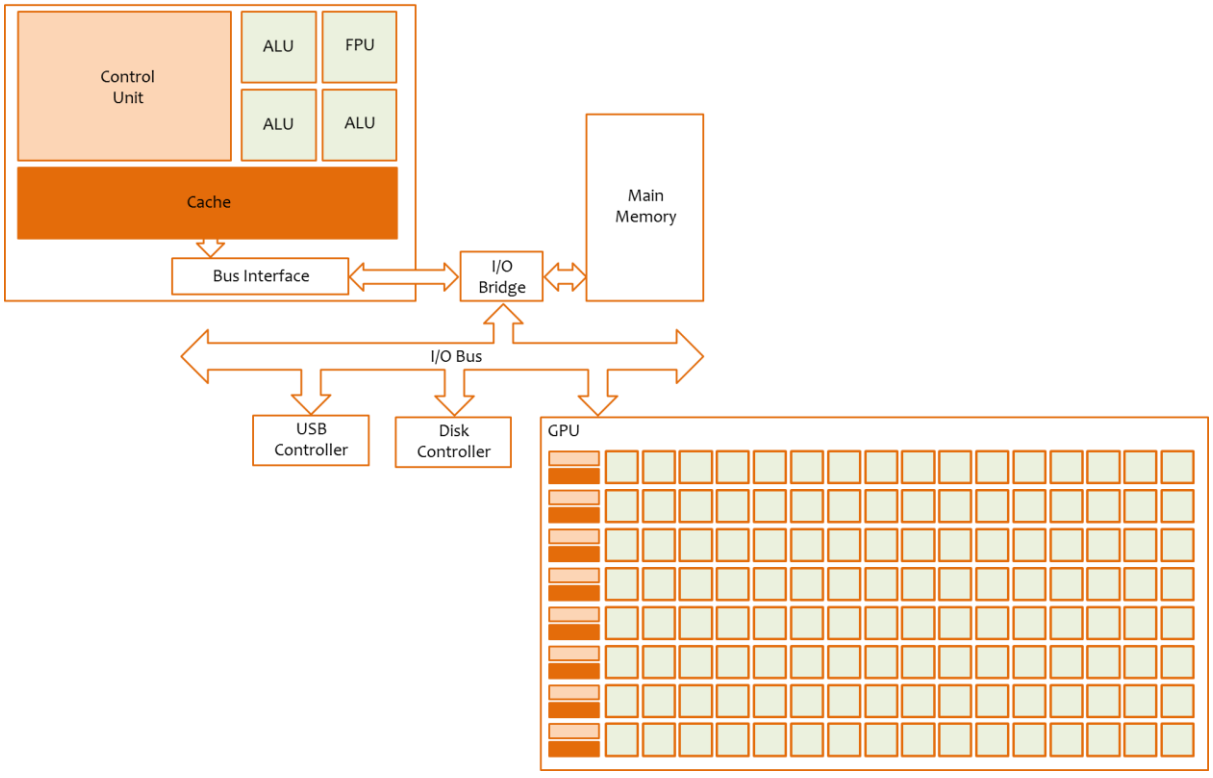
By way of introducing the functions of a GPU this paper presented earlier a highly simplified visual of a GPU which in fairness mixed functions and technology on the same graphic:

Nvidia GeForce256



Given our recent deeper investigation of the CPU it is now time to understand better the internals of a GPU that make the "functions" such as Geometry Stage, Rasterizer, etc. work from a computer architecture viewpoint. A more realistic circuitry or transistor aligned representation for visual comparison purposes of the GPU would be:

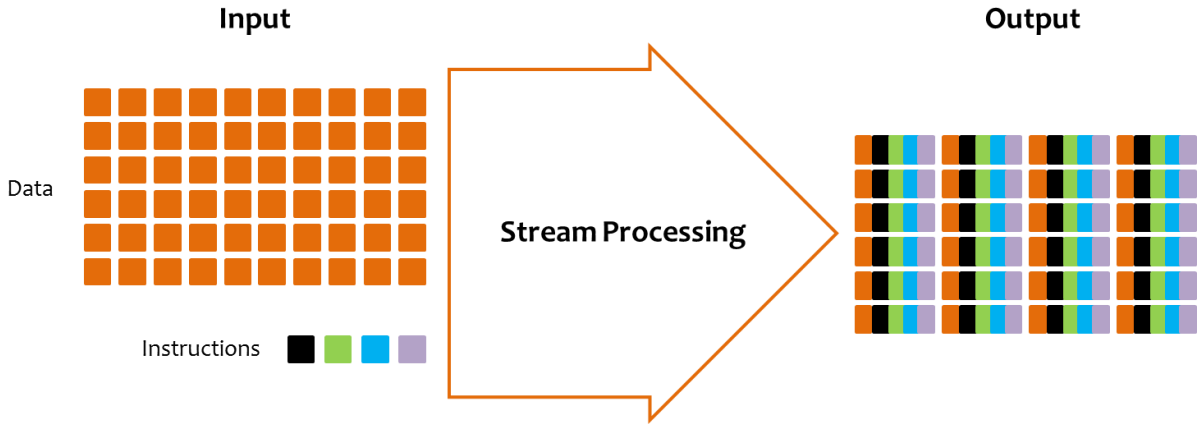
CPU & GPU



The reason for including this visualization is simply to make clear at the highest level that there are huge architectural differences between CPU's and GPU's and I believe this image makes that very clear. The actual architectural differences are such that it is clear that we won't (probably) ever be using GPU's for mass scale general purpose computing but will likely be using a hybrid CPU-GPU for doing so in the medium term (3-5 years). There is one final technical concept to understand first though – Stream Processing.

5.3.1 Stream Processing

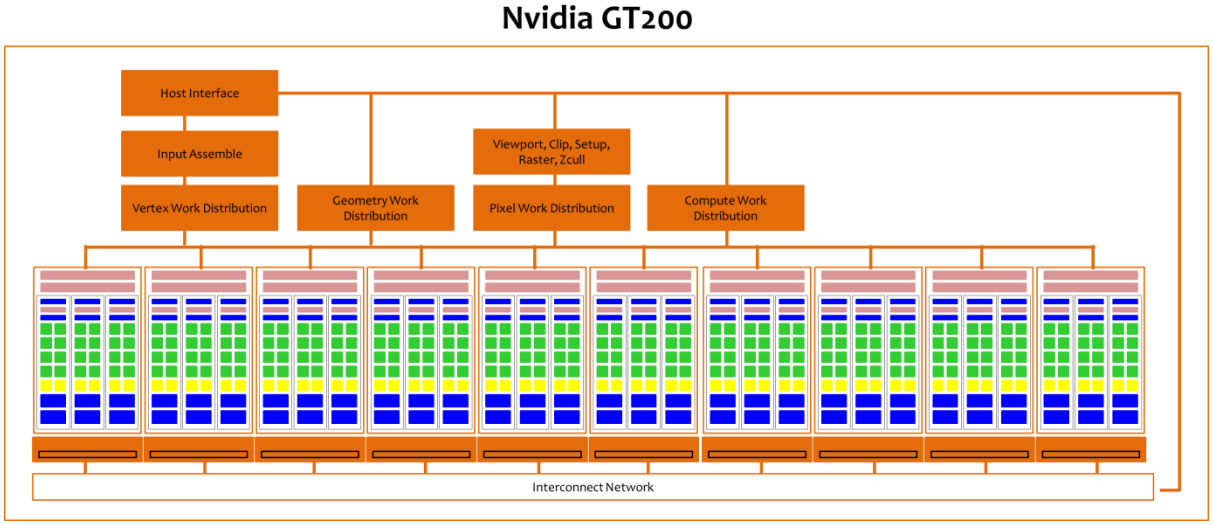
Stream processing is closely related to SIMD and can be visualized in a very similar way:



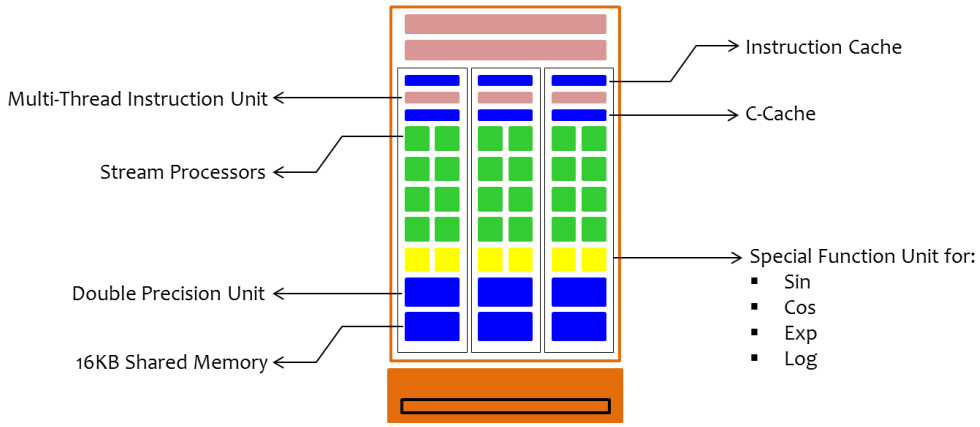
Simply stated a stream processing comprises of a set of input data on which a series of instructions is carried out in a pipeline. In sequence terms the stream processor is configured or loaded with the series of instructions to be carried out and then the data is fed through as a stream. This technique can process very large volumes of data very quickly and is employed extensively in GPU's.

5.3.2 The Nvidia GT200

The architecture of the Nvidia GT200 can be visualized as:



A deeper look under the cover reveals all of the components we covered earlier:



5.4 The Key Architectural Differences: CPU vs. GPU

This brings to an end the comparisons of CPUs and GPUs and how they developed over time. Those developments were originally aimed at very different goals with the outcome being two very different architectural approaches, namely:

CPU	GPU
▪ Low latency – cannot tolerate a delay between initiation of an action and demonstrable effect (think mouse, keyboard); ▪ Low throughput – small amount of work done in a given chunk of time; ▪ Utilises many cache levels to minimize latency; ▪ Memory management is optimized for pre-fetch and out-of-order execution; ▪ Control unit is heavily utilized to manage flows, addresses, etc; ▪ Designed for tens of complex threads to run concurrently with each thread scheduled explicitly; ▪ SISD and some SIMD utilized; ▪ Designed for general purpose computing.	▪ High latency – can tolerate a delay between initiation of an action and demonstrable effect; ▪ High throughput – high amount of work done in a given chunk of time; ▪ Low usage of caches; ▪ Designed to run tens of thousands simple threads concurrently with hardware managing the scheduling of threads; ▪ SIMD model utilized; ▪ Stream processing architecture; ▪ Designed for graphics rendering.

Both CPU's and GPU's have a place in our computing world whether for number crunching or graphics rendering. That said, it may not be lost on the reader that given the shear performance advantages and the trajectory of that evolution the question has become – "couldn't we utilize the GPU for more than just graphics?"

5.5 Sources & Further Reading

http://www.techspot.com/article/650-history-of-the-gpu/

http://pdf1.alldatasheet.com/datasheet-pdf/view/4159/MOTOROLA/MC6845.html

http://www.df.lth.se/~pi/compis/datablad/Intel82720.pdf

http://www.amigahistory.co.uk/1000spec.html

http://en.wikipedia.org/wiki/S3_Graphics

ftp://download.nvidia.com/developer/presentations/2004/Perfect_Kitchen_Art/English_Evolution_of_GPUs.pdf

http://www.vintage3d.org/nv1.php#sthash.3GORiH7W.dpbs

http://textfiles.com/computers/PRESSRELEASE/ati_rage.txt

http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units

http://www.bit-tech.net/hardware/cpus/2008/04/30/david-kirk-interview/1

http://www.amd.com/uk/Documents/GCN_Architecture_whitepaper.pdf

http://malideveloper.arm.com/downloads/WhitePaper_GPU_Computing_on_Mali.pdf

http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html

ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf

http://cseweb.ucsd.edu/~marora/files/papers/REReport_ManishArora.pdf

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

http://en.wikipedia.org/wiki/CPU_cache

http://developer.amd.com/wordpress/media/2012/10/GPU_Architecture_final.pdf

http://www.cis.usouthal.edu/~hain/CSC520_F09/GPUpresentation.pdf

http://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

http://en.wikipedia.org/wiki/Comparison_of_ATI_graphics_processing_units#FireGL_Series

http://en.wikipedia.org/wiki/Comparison_of_Intel_graphics_processing_units#Second_generation

http://www.theregister.co.uk/2013/05/01/amd_huma/

http://www.amd.com/uk/products/technologies/hsa/Pages/hsa.aspx#1

http://www.datacenterjournal.com/it/gpu-replace-cpu/

http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

http://web.engr.oregonstate.edu/~palacijo/cs570/final.pdf

Stokes, J 'Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture' (2007, William Pollock)

Baer, JL 'Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors' (2010, Cambridge University Press)

6. INTRODUCING GPGPUS, CUDA & OPENCL

The website www.eamonnkillian.com provided a high level review of the relevance of video games and their influence through pushing the boundaries of our societies and technologies. It is easy to think of games as "for children" or "something simplistic". Hopefully this paper, which examined in some detail one singular facet of the envelope that's being pushed by videogames manufacturers and their hardware partners – namely Graphics, can dispel some of the naivety in persons outside of the videogaming industry. Before concluding though there are major collateral benefits which have spun-off from the efforts of the videogaming industry. Developers in areas as diverse as the search for the Higgs Boson through to Oil & Gas Exploration and Medical MRI enhancement have achieved astounding results on the foundations laid by the Videogame industries relentless search for better images.

6.1 Brief Introduction to CUDA & OpenCL

The notion of harnessing the extraordinary processing speed of a GPU was certainly not going to be left unaddressed and with the advent of programmable stream processing it was a certainty. It is clear from the foregoing that while certain low latency tasks will always require a general purpose CPU approach there are many areas of today's number crunching batch processing runs that could benefit from the approach of a GPU. The mitigating factor is that this requires extensive reprogramming of the base applications. That is no mean feat! In addition for many of the early years the only approach to programming a GPU was through the OpenGL application programming interface (API) framework we briefly covered in Section 4. What was needed was a programming framework that would expose the GPU to programmers familiar with higher level languages (HLLs) such as Fortran, C, C++, of Java. Necessity stepped in once again with the introduction of CUDA and OpenCL:

- **CUDA™ -** Nvidia's Computer Unified Device Architecture (CUDA™) is a parallel computing platform makes it possible for programmers to harness the power of highly parallel graphics processing as part of their standard kit bag;
- **OpenCL-** Open Computing Language (OpenCL) is a programming framework from Khronos (of OpenGL fame) that provides a heterogeneous platform for parallel computing.

Today the design of many of the worlds' supercomputers is probably pointing the direction forward for general computing as a whole. This design takes the best of both CPU and GPU worlds that we reviewed in Section 5 and builds a symbiotic architecture that leverages the strengths of both to achieve optimum results. The website Top500.org lists the worlds' top 500 supercomputers using the FLOPS measurement we outlined earlier. The Top 5 at the time of writing (December 2013) are:

1. Tianhe-2 with 33.86 Petaflops – no GPU usage but a highly parallel co-processor the XeonPhi;
2. Titan Cray with 17.59 Petaflops – 18,688 AMD CPU's with 18,688 Nvidia Tesla K20X GPU's;
3. Sequoia IBM BlueGene/Q with 17.17 Petaflops – no GPU usage, leverages IBM PowerPC A2 processors;
4. Fujitsu K with 10.51 Petaflops – No GPU usage, leverages 80,000 SPARC64 VIIIfx processors;
5. Mira BlueGene/Q with 8.59 Petaflops – No GPU usage, leverages 65,536 PowerPC 770 processors.

For Tianhe-2 that's 33,860,000,000,000,000,000 (thirty three quadrillion) floating point operations per second - Wow!

One might look at this list and note that the GPU has only achieved one spot in the Top 5 supercomputers which of course is factually correct. However GPUs have taken 54 slots (39 Nvidia, 11 Intel and 3 AMD) in the Top 500 with their fastest growth space in academia up 680% in 2012 from 2011.

6.2 Applications of GPU Computing

This mixed CPU/GPU hardware architecture has gained momentum in recent years in the world of supercomputing. Once the sole bastion of names such as Cray, IBM, NEC, Fujitsu, and Intel the world of high performance and supercomputing is entering what could be described as a renaissance period. From the Cray-1 onwards engineers knew that supercomputing was always going to be about parallelism. Since the Cray-1 in 1976 clocking in at 80Megaflops to today's Petaflops capabilities much has clearly changed. Two of the most interesting facets in supercomputing are:

1) The Student Cluster Competition which was won in 2013 by a team from China's National University of Defense Technology (NUDT) at 8TFLOPS, second place to Team Kraut at 7.4TFLOPS and third to Team Venus at 7.2TFLOPS each leveraging GPUs;
2) The Commodity Competition won by Team Sun Devil at $4.99 per GFLOP, second Team Rock at $5.01 per GFLOP. The most interesting fact is that the commodity teams must spend less than $2500 to enter and they achieved an astounding 495 & 474 GFLOPS respectively and incredible achievement

These staggering achievements are pointing to the introduction into mainstream business and education of commodity high performance computing (HPC) employed to deliver huge increases in computational capability at minimal costs.

6.2.1 What can GPU supercomputing do?

So what can GPU supercomputing do? When you spend several tens of millions of dollars on a high-end supercomputer you get computers capable of almost anything we can conjure up. But in today's post financial market meltdown world we need cost effective solutions. Here are just a few examples:

- Modelling the blood flow of individual patients in a non-invasive manner then leveraging GPUs to enable complex fluid dynamic simulations of the flow to identify areas where the flow is sluggish (based on endothelial sheer stress on the arterial wall). This helps cardiologists to identify where to place stents to alleviate the blockages and ultimately prevent a heart attack;
- In financial markets the use of derivative products has grown dramatically over the past decade. The use of derivatives also introduces a complexity when evaluating the financial position of the company. In financial markets the smallest error can cost millions. The complex mathematics involved used to take a number cruncher overnight to resolve and what traders needed was a near-time or real-time solution. Using GPUs companies have seen a 30 to 100 times performance improvement enabling broader simulations of derivative price movements to be modelled in advance and decision criteria to be set;
- The search for Oil and Gas deposits has always been expensive but with the scarcity of supply today this is ever more so. Seismic imaging companies have invested in innovative approaches that evaluate underground conditions to determine the expected geological

structures of the ground above. These solutions require massive scale computing and by employing GPUs the processing time required has increased by up to 600 times to make prospecting much more cost effective.

6.3 And Finally - A Very Quick Convergence Sojourn into BigData & MapReduce

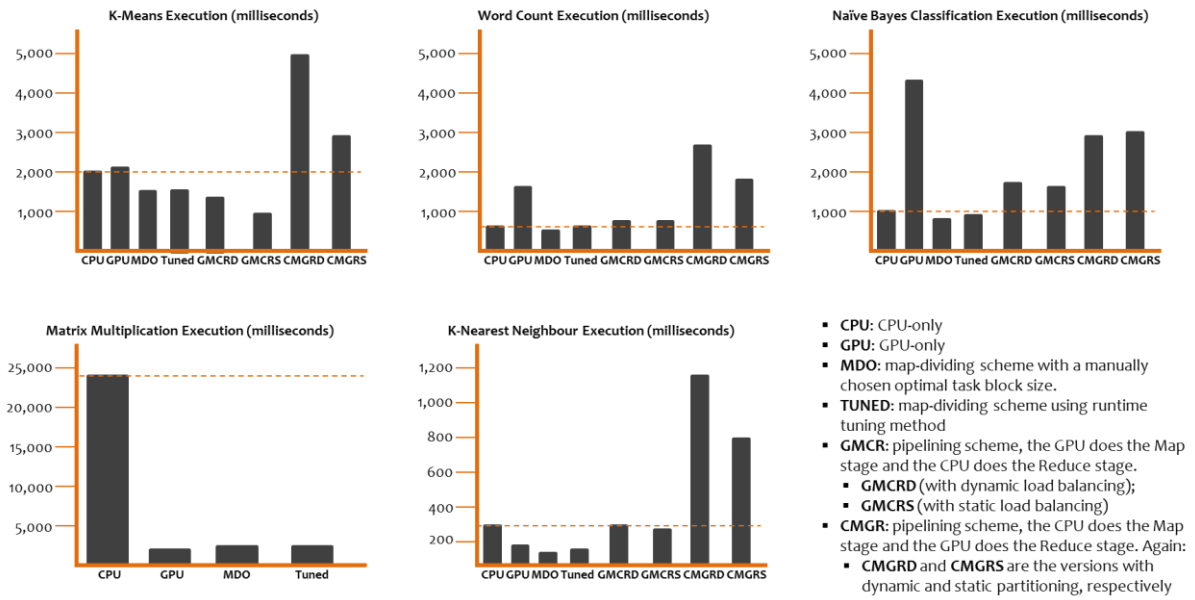
There will be a separate paper on MapReduce on the BigData pages of www.eamonnkillian.com in the coming weeks. In the meantime for the purposes of this paper it suffices to understand that MapReduce is a programming model made up of three steps i) Map, ii) Group and iii) Reduce. MapReduce has fundamentally altered the approach to indexing massive amounts of unstructured data. Developed by Google as a means to helping index the Internet, it was adopted by Yahoo and latterly by Apache as a Foundation Project. At a high level MapReduce provides a programming framework to process large unstructured or structured data sets in a highly parallelized manner utilizing hundreds and thousands of compute nodes in a cluster configuration. MapReduce is one of the most important tools in the armory of data scientists in managing, manipulating and working with really big data sets. As such improvements in performance are critical to the further exploitation of BigData.

Efforts to date to build a universally acceptable GPU MapReduce capability are ongoing and the challenges are well understood (https://sites.google.com/site/mapreduceongpu/home/why-how), namely:

- Synchronization issues;
- Workload allocation across GPU threads;
- Limited string processing & file manipulation;
- Memory constraints for datasets larger than GPU memory; and
- Implementations are often for solo GPUs rather than clustered GPU's.

The latest attempts to address these constraints are highlighting significant strides forward with recent results from Ohio State University leveraging a coupled CPU-GPU architecture benefit highlights across five commonly used MapReduce applications, namely:

- K-means Clustering – a method used for modeling of probability density functions by the distribution of prototype vectors;
- Word Count – indexing and quantifying the words in a given input set;
- Naïve Bayes Classifier – a simple classification technique to return the probability of class membership based on the independent evaluation of each contributing variable;
- Matrix Multiplication – multiplying two nxm matrices together;
- K-Nearest Neighbour – another classification method.



This reveals that the paradigm of a joint CPU-GPU approach leveraging each for their respective strengths yields between a 3.25 to 28.68 times' performance improvement. Efforts continue with MapReduce frameworks such as: Mars, Panda, OpenACC, GPMR, DisMaRC, MITHRA and MapCG bot name a few.

6.4 Sources & Further Reading

https://developer.nvidia.com/what-cuda

http://www.nvidia.com/content/GTC-2010/pdfs/2131_GTC2010.pdf

http://mc.stanford.edu/cgi-bin/images/f/f7/Darve_cme343_cuda_1.pdf

http://gpgpu.org/

http://mc.stanford.edu/cgi-bin/images/f/f7/Darve_cme343_cuda_1.pdf

https://www.olcf.ornl.gov/wp-content/uploads/2013/02/Intro_to_CUDA_C-TS.pdf

http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf

http://www.math-cs.gordon.edu/courses/cps343/presentations/Intro_to_GPGPU.pdf

http://www.bit-tech.net/news/hardware/2013/11/15/amd-firepro-s10000/1

http://ixbtlabs.com/articles3/video/cuda-1-p3.html

http://www.theregister.co.uk/2013/12/12/sc13_cluster_competition_linpack_results/

http://www.nvidia.co.uk/object/harvard-university-uk.html

http://www.nvidia.com/object/sci_comp_finance.html

http://en.wikipedia.org/wiki/MapReduce

https://sites.google.com/site/mapreduceongpu/home/why-how

http://www.cs.kent.edu/~zwang/schedule/dc8.pdf

http://www.cse.ust.hk/gpuqp/Mars.html

http://conferences.computer.org/sc/2012/papers/1000a050.pdf

http://www.math.le.ac.uk/people/ag153/homepage/KNN/OliverKNN_Talk.pdf

http://www.mathwarehouse.com/algebra/matrix/multiply-matrix.php

http://en.wikipedia.org/wiki/Naive_Bayes_classifier

http://home.deib.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html

Hwu, W (Editor) 'GPU Computing Gems Jade Edition' (2011, Elsevier)

Hwu, W (Editor) 'GPU Computing Gems Emerald Edition' (2011, Elsevier)

Kirk, D & Hwu, W 'Programming Massively Parallel Processors: A Hands on Approach' (2010, Elsevier)

7. CONCLUSION

Apologies first for the length of this paper but it is hoped that the non-programming, non-IT literate reader will have gleamed much in the way computer graphics processing has developed, not least because videogames demanded more power, more bullets, more interaction and more graphics realism.

We've been on a long journey from the earliest inculcations of CPU bound graphics processing in the 1970as through to the development of specialized GPUs. On that journey we've reviewed the requirements of graphics processing and more specifically how graphics are generated with the world viewed as 80 million pixels, the importance of polygons and in particular triangles as the base geometric primitive for graphics processing. We've briefly touched on the enhanced requirements necessary to deal with 3D graphics and introduced through a birds-eye view the OpenGL pipeline. We then went deeper in to the architecture of CPU's and GPU's to understand that parallel processing drove GPU's into a certain form whereas general purpose computing and number crunching drove a need for something entirely different. We have also noted that Moore's Law has reached its boundaries in terms of what a single core on a CPU can do which drove CPU's to begin an evolution to multi-cores but that the base design of GPU's continues to enable astounding performance gains in floating point calculations.

Latterly, we've reviewed the recent efforts to embrace the parallel computing power of the GPU for general purpose tasks, highlighted some of the incredible achievements it has delivered and noted the on-going efforts to employ GPU's within BigData and Analytics through GPU MapReduce.

In summary then it is hoped that the reader leaves this paper with a greater degree of clarity on:

vi. Why GPU's were developed;

vii. Why triangles are so important to graphics processing;

viii. Why high degrees of parallelism are becoming increasingly important;

ix. How GPU's are being utilized to deliver significant gains in industries and market sectors far beyond the original design criteria for the GPU; and

x. Why GPU's cannot wholly replace CPU's and that the future is most likely a symbiosis of the two capabilities leveraging each for their inherent strengths.