

Potential of Commodity Graphics Hardware for Scientific Computation

Author: Andrew A. Murdoch (andrewm@epcc.ed.ac.uk)

Institution: EPCC, The University of Edinburgh

Date: 5th November 2002

Introduction.....	2
Chapter 1 – Background on 3D Rendering	3
The Graphics Pipeline.....	3
Transformation & Lighting	3
Clipping	3
Fragment Rasterisation	3
Scan line conversion	4
Texturing.....	4
Hardware Acceleration	4
Graphics APIs	5
Direct3D	5
OpenGL	5
Chapter 2 – Current and Future Trends	6
Programmable Rendering Hardware.....	6
High Level Shading Languages	6
High Dynamic Range Rendering	7
Problems of Range and Accuracy.....	7
Representation of Values other than Colour	8
Emerging Support	8
Improved performance.....	9
Fewer internal restrictions on programmability.....	9
Flexible High Level Language support.....	9
Faster communication back to host.....	9
Chapter 3 – Typical 3D Applications	10
Entertainment.....	10
Design / Visualisation	11
Convergence of Commodity Hardware	11
Chapter 4 – Computation	12
Background.....	12
Programmable Graphics Architecture.....	12
Graphics Processor Programs	13
GPU Program Input and Output	13
Comparison with PC System	14
Previous Work	14
Test Case – Horizontal Edge Detection	15
Performance Analysis:.....	15
Implementation and Performance Issues	17
Combined Performance	17
Conclusion	18
Further Work	18
References.....	19

Introduction

The aim of this report is to explain the potential value of commodity graphics hardware in the context of desktop scientific computation. Much effort has been put into improving performance at the high end of scientific computation. Many tasks, especially those involving visualisation or real-time processing of data, can be performed using desktop machines, such as high end workstations which include multi-processor systems.

Graphics hardware has traditionally been used for visualisation, entertainment and interactive design. However, it is possible to use such equipment for computational purposes. This situation can take advantage of the advances and rapid improvement in performance of graphics technology. In this report the potential of these devices and possible applications will be discussed.

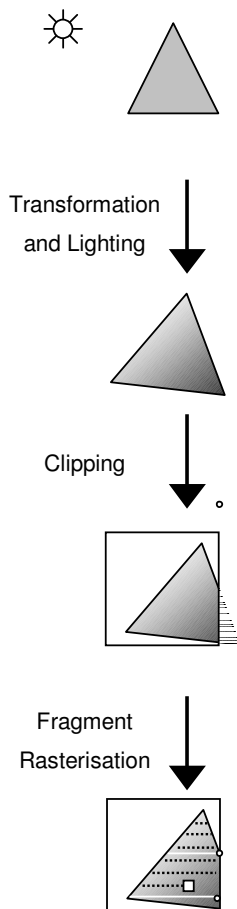
Chapter 1 – Background on 3D Rendering

Before looking at possible applications for graphics technology, it is useful to be aware of techniques and terminology in 3D rendering.

In computer graphics there are a number of methods by which three dimensional (3D) scenes may be rendered, such as ray tracing, volume rendering and numerous scan line based polygon algorithms. The most dominant with regard to real time implementations is z-buffered polygon rendering. In this method, polygons represented as collections of 3D points are drawn into a 2D area of display memory (the *framebuffer*) subject to various viewing parameters such as the virtual camera position and the colour and position of lights.

The Graphics Pipeline

The typical polygon rendering process can be interpreted as several distinct phases; transformation and lighting, clipping and fragment rasterisation, each of which passes data to the next until an image is produced in the frame buffer.



Transformation & Lighting

To display a 3D object on a 2D surface, it must be projected onto a plane representing the viewing area. Therefore, a transformation matrix is applied to each of the vertices of a polygon. This transforms the model into viewers coordinate system. At this stage, local lighting calculations are performed. This involves taking the surface normals of each polygon and calculating shading information based on the position of lights in the environment and the surface properties. The x and y coordinates are then divided by the distance from the viewer (z), giving the object perspective.

Clipping

The transformed polygon may overlap the edges of the display, or indeed may not be visible at all. The parts that are not visible are removed, which can result in new vertices at the edge of the screen. This process is known as clipping. Values such as colour and depth are subsequently interpolated from the original vertices. Clipping also occurs in the z -axis, in which parts of objects that are too close or too far away are removed. The resulting polygon is considered to be composed of a set of fragments.

Fragment Rasterisation

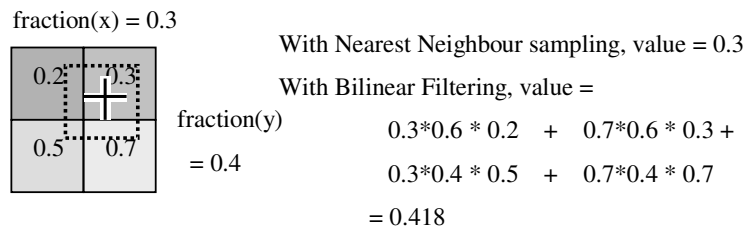
Fragment rasterisation is the drawing of 2D primitives into the pixels of framebuffer memory. This consists of scan line conversion and texturing:

Scan line conversion

The fragments of a polygon are written to the display in a process known as *scanline conversion*. Starting from the top of each fragment and progressing along each horizontal span in turn, each pixel has tests applied to it. Examples of such tests include assessing whether the depth value is less than the existing value of that pixel and whether it should be drawn according to the corresponding value in the bit-based stencil buffer. Once these tests have been passed, the pixel is assigned to the colour of the current fragment. Blending of the new pixel value with the existing one may occur depending on the blending mode and the opacity of the polygon.

Texturing

A textured polygon is one that has one or more images associated with it. Each vertex of the original polygon is given texture coordinates, referring to positions within the texture image. During the process of fragment rasterisation, the texture coordinate of each point is based on interpolation from the vertices. This coordinate is used to access the texture and provide a colour for that pixel. There are a number of ways by which this can be performed. In the simplest form the value taken can be the value stored at the texture coordinate. This is known as *nearest neighbour* sampling. Another texture sampling method is *bilinear filtering*, in which the four values surrounding the texture coordinate are sampled and weighted according to the fractional position of the coordinate within the texture image. More sophisticated methods exist such as variants of *trilinear filtering* and *anisotropic sampling* which attempt to produce a better representation of the rendered texture.



Example of simple texture sampling methods

Taken together, both scan line conversion and texturing of the fragments is referred to as fragment rasterisation

Hardware Acceleration

Currently, the order of transformation of 3D into 2D, stated above is the most commonly adopted, and historically the subsumption of the above functions has been from the end of the pipeline backwards. Shaded scan conversion and texturing were accelerated, followed by fragment processing, clipping and finally vertex transformation and lighting.

Most modern PCs with a discrete graphics card have full hardware transformation and lighting support.

Due to the importance of the graphics hardware in all stages of rendering, the graphics chip is now frequently referred to as a *GPU* or Graphics Processing Unit.

Graphics APIs

Although initially in the development of consumer level graphics hardware a number of generally proprietary APIs proliferated this situation has since changed, to the benefit of programmers and users of 3D applications. The two main standards for real-time graphics programming are Direct3D from Microsoft and OpenGL, a standard originally based on SGI's GL API.

Direct3D

Direct3D, produced by Microsoft is a Windows only graphics API comprising of a software renderer and a hardware abstraction layer for accelerated rendering. DirectX, the multimedia library containing Direct3D is currently at version 8.1 because a new version is released every time hardware features are added to the standard. Many consumer 3D cards are classified according to their Direct3D acceleration level, e.g. the NVidia GeForce 2 MX is a DirectX 7 part (it features transform and lighting support), while the ATI Radeon 8500 is a DirectX 8.1 part (it supports limited programmable vertex transformation and pixel shading).

OpenGL

OpenGL is mature library, available on many platforms, including Windows, Unix, Apple Macintosh and others. The OpenGL Architecture Review Board (ARB) administers the standard and ensures compliance of implementations and is comprised of members from several hardware manufacturers, applications providers and universities. Changes to the API are made infrequently and after significant consideration. However, an extension mechanism exists that allows hardware manufacturers to expose new features in the API before they become standardised.

Chapter 2 – Current and Future Trends

There are a couple of emerging trends of significance to computation with graphics hardware.

Programmable Rendering Hardware

In a move to continue competing on features and offer content producers (predominantly games companies) more flexibility, graphics card manufacturers are making their geometry transformation and fragment rasterisation stages more programmable. However, this has occurred in a piecemeal fashion with manufacturers producing incompatible extensions leading to limited flexibility in the rendering pipeline. Fortunately, standards are emerging which unify these facilities to define a more comprehensive set of functions and capabilities of graphics cards.

High Level Shading Languages

Outside of real time interactive graphics, shading languages are commonplace and have mature tools for the creation and debugging of novel surface shaders. In particular, the RenderMan Shading Language, used in the RenderMan product by Pixar¹ has been in use for over 10 years and has helped create the computer-generated films “Toy Story”, “A Bug’s Life” and “Monsters, Inc.”. High performance real time rendering places more constraints on the shading language. The underlying hardware often has a limited number of temporary registers that may be used. A limited number of textures may be applied to each new pixel being written and there may be an instruction limit on the length of programs. For these reasons, we cannot directly apply these existing high-level shading languages to PC graphics cards. Despite this, digital content creators can use many familiar techniques and concepts with the emerging generation of shading languages. Although all in their infancy and subject to change there are three high level shading languages for commodity graphics hardware:

- Direct3D 9, the currently awaited new version of Microsoft’s library, adds support for a high level shading language (HLSL) with a C-like syntax for specifying vertex and fragment programs.
- OpenGL 2.0, the successor to OpenGL is planned to feature high-level languages for vertex and fragment programming with few limitations on program size or number of intermediate values. An early release of OpenGL 2.0 is available from 3DLabs for their latest product. More information can be found at: <http://www.3dlabs.com/support/developer/ogl2/index.htm>
- Cg is a shading language developed by NVidia Corporation. It has a C-like syntax, similar to the DirectX 9 high level shading language but can be targeted at different graphics libraries and hardware through the use of compile and runtime profiles. Despite some attempts to involve other manufacturers, it is largely a proprietary language and most closely maps to NVidia hardware.

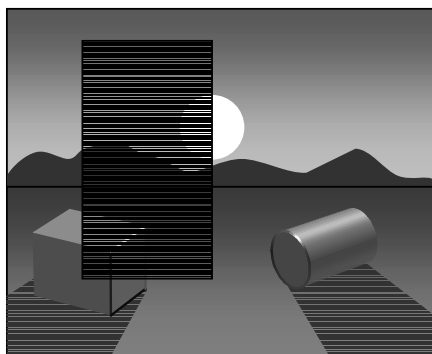
Some crossover of the above exists, as potentially Cg could output low-level code for OpenGL or Direct3D profiles. Additionally converters are likely to be written that directly translate shaders from high-end rendering applications such as 3D Studio, RenderMan, etc. to one or more of the above languages.

High Dynamic Range Rendering

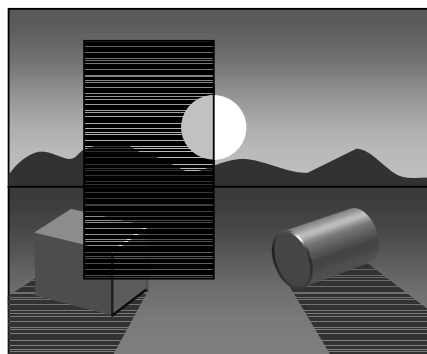
Problems of Range and Accuracy

One of the restrictions of the classic polygon renderers has been the specification of colour as a triplet of 8 bit (or less) values representing the red, green and blue (RGB) components which make up every displayable colour. These values are normally accepted to represent the range (0,1), where zero represents no colour and one represents the maximum value. While for many purposes this is sufficient, e.g. the display of photographs, user interfaces, etc, when rendering natural scenes this is not always enough. The human eye can detect variations in shades of light over a huge range of intensities, from a darkened room to midday sunlight.

The gamma value of computer monitors defines the logarithmic scale of brightness for each colour component, of which 256 distinct values may be represented. However, there is still not enough variation to smoothly cover this range. Furthermore, the maximum brightness that may be represented is limited. Although it would not be desirable for a display device to produce intensities of similar magnitude to the sun, it is useful to be able to represent them. For example, we may be rendering a view of a scene that is partly covered by a tinted piece of glass with 50% opacity. While most objects or parts of scenery might be darkened, certain things, such as the sun, should not be. If we treat our framebuffer as a camera film, sensitive over a specified range, even at 50% reduction the sun and other very bright objects should still appear maximally bright. Paul Debevec has carried out useful work on this subject (<http://www.debevec.org/Research/HDR/>).



Basic scene with limited brightness
(sun intensity = 1)



Basic scene with unlimited brightness
(sun intensity = 1.5)

Representation of Values other than Colour

Another aspect of the restriction of the range of colour values in graphics card occurs in situations where the texture or framebuffer data do not directly represent colour. The values may represent other characteristics such as shininess, perturbations in surface normals and lighting. When the value represents a vector, unit vector components must be mapped from the range $(-1, +1)$ to the $(0, 1)$ range, representing colour, and intermediate calculations must be appropriately scaled and biased to prevent saturation of the values. In addition some current hardware allows intermediate values with wider ranges, such as $(-8, +8)$ and 16 bit accuracy, but this is not standard and still does not improve the accuracy of stored values. A polygon with a texture representing gradual curves using only 24 bits of accuracy (8 bits per component) may show step-like artefacts once it is rendered. These would be especially evident if one uses reflective texture coordinate generation, a technique for efficiently rendering reflective objects.

Emerging Support

Both of the two major commodity graphics hardware manufacturers, NVidia and ATI, are shipping or have announced products that can store textures and internal image buffers with higher precision than previous generations, supporting 16 and 32 bit floating point RGB components. Unfortunately, the hardware in graphics cards for scanning through the visible framebuffer and displaying the image on an external display is unable to convert floating point values rapidly to output signal voltages. Despite the inability to render to a floating point framebuffer with these new cards, it is possible to render to a higher precision off-screen image buffer and quickly copy to a lower precision displayable framebuffer. During this copy process, brightness modifications and other useful effects such as saturation control and sepia toning may also be performed with nominal performance cost.

With regard to the future of commodity graphics hardware we can anticipate certain trends (ordered to some extent by certainty):

Improved performance

There has been no slackening in the rate of improvement of performance of graphics hardware, both in terms of vertices transformed and pixels shaded. At a rate almost double that of Moore's Law¹, graphics performance has doubled approximately every 9 months for over 5 years.

Fewer internal restrictions on programmability

Currently vertex and pixel pipelines have fixed maximum lengths for their programs and also fixed maximum numbers of particular instructions (such as texture fetches). Similarly vertex programs may have simple branching and function call support and pixel programs are more limited, having no branching support at all. Future generations are likely to reduce or remove these restrictions either by extending the architectures or by automatically subdividing programs at the driver level to give the appearance of unlimited length programs.

Flexible High Level Language support

Languages with C-like syntax are already in existence or at the standardisation stage. OpenGL2 will support a high level shading language, which will allow the replacement of various parts of the standard pipeline. With a well-defined high-level language, it becomes easier to write pre-processors allowing the embedding of more easily understandable and maintainable code into application source code. Although ultimately a compiler could be developed to transform segments of C code into a shading language, this is a major undertaking and similar approaches have had limited success (e.g. Intel's Reference Compiler with SSE2 support).

Faster communication back to host

Although an AGP card is capable of transferring a reasonably large amount of data from the host machine, e.g. for an AGP X4 device: 1065MB/s, the practical bandwidth back to the host, despite the bus specification being symmetric, can be 1/100th of the download bandwidth. This is assumed by many graphics programmers to be a limitation of graphics drivers rather than a hardware restriction. For this reason it is hoped that some effort is put into improving this situation.

¹ A rule of thumb coined by Gordon Moore, a co-founder of Intel Corp. Roughly stated, the density of transistors in microchips doubles every 18 months, usually and so far accurately assumed to mean that application performance on such a device also doubles in that time interval.

Chapter 3 – Typical 3D Applications

We have so far examined the techniques and architectures used to perform real-time 3D rendering, but we have not covered the applications currently exploiting it. These fall into two main categories, each with their own particular hardware requirements:

Entertainment

This category is largely composed of 3D games, of which the “first person shooter” is among the most popular.

The game “Quake” by iD Softwareⁱⁱ is a good example. Games usually have simple character models and scenes that make up for lack of geometric detail by using large, detailed textures. Often effects such as smoke and light coronas further enhance the appearance.

The requirements of such games can be characterised as needing only limited geometry processing but very good texturing rates. In Quake, environments had a physically accurate but computationally intensive illumination map applied to all static surfaces in addition to the standard surface texture. Applying more than one texture to a primitive, or *multi-texturing* as it is called, was not widely supported at that time, but because of the popularity of the game and others using the same method, multi-texturing became standard, despite previously being an exotic feature of high end graphics workstations.

Subsequently, iD Software has released a number of successors to Quake, the predictably named, Quake 2, which added coloured lighting and more detailed character models and Quake 3 Arena, which had a simple shader language for applying more complex effects to surfaces. These *engines*, as the 3D technology separate from the particular game logic is known, have been widely licensed by game development houses.



Figure 1 - Quake 3 Arena sample screenshot

Because of the number of vendors in the graphics hardware market, the economy of scale and competition from the console market, prices for 3D cards, have been kept relatively low and the pace of innovation has been high.

Design / Visualisation

For some time there have been graphics accelerators available for professionals involved in design, engineering and scientific visualisation. The inclusion of the OpenGL API in Windows NT made it an attractive platform for the producers of high-end 3D applications and their users who previously were restricted to higher cost proprietary Unix workstations. Most such applications involve inspection and manipulation of highly detailed models, rendered with precise accuracy. Except for the creation of digital content for film and television, the models are not typically textured, though they may have multi-source real-time lighting applied to aid comprehension.

AutoCAD by Autodeskⁱⁱⁱ is a tool commonly used by designers and engineers for 3D modelling. This is sold as different versions for different design tasks, such as mechanical, architectural and product design projects.

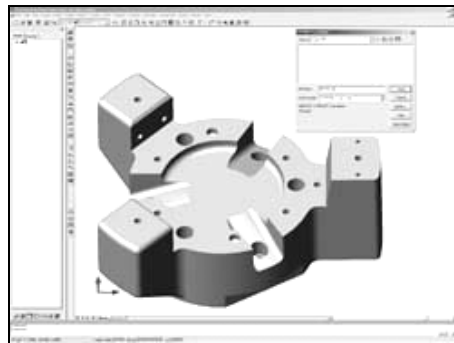


Figure 2 - AutoCAD Mechanical Desktop example

For scientific visualisation, users often load datasets into applications such as AVS Express^{iv} or MatLab, or for specific tasks use software libraries such as VTK^v (Visualisation Tool Kit) to quickly create project specific visualisation applications. A new area technique within simulation is *Computational Steering*, where the users directly control the parameters of a simulation and can rapidly inspect the results.

3D hardware for design and visualisation has traditionally had good support for transforming and lighting large polygonal datasets, frequently with no texturing support. Due to the specialised target market, hardware prices have been comparatively high - hundreds to thousands of pounds.

Convergence of Commodity Hardware

In recent years due to the rapid improvements in consumer hardware, the manufacturers have been competing in the professional market with variations of their mainstream products. Often these are virtually identical to their mainstream product apart from the supplied drivers, which are rigorously tested to meet the certified requirements of the professional applications.

A niche still exists for high-end professional hardware and 3DLabs and Intergraph continue to produce products without directly competing in the consumer market, though they play a part by licensing technologies and through involvement in standards bodies.

Chapter 4 – Computation

Background

The aim of 3D graphics is usually to produce lifelike scenes and objects, either for simulation or entertainment purposes. To create images indistinguishable from reality would require the simulation of interactions between all of the photons and atoms in a scene, to correctly show phenomena from subsurface scattering in skin, wax and marble, to anisotropic multi-spectral specular effects observed in polished metals and oil films.

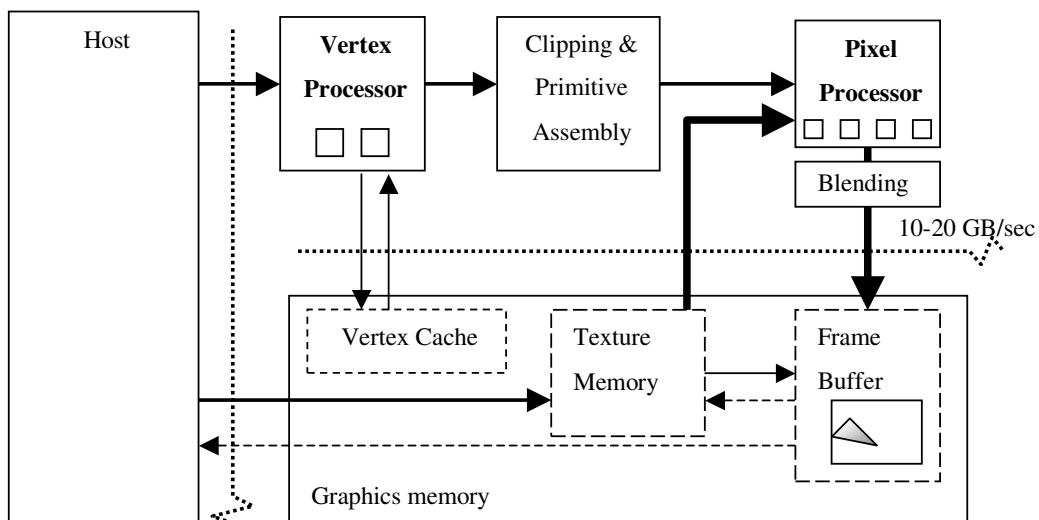
It is not currently possible to perform this level of simulation, so graphics programmers employ a wide variety of “tricks”, or special effects to approximate these phenomena. As graphics has become more sophisticated, so these techniques have become more complex and better at producing the desired effects. This has required graphics programs and graphics hardware to perform a wider variety of operations, some combinations of which may not have been fully anticipated by the original equipment manufacturers. Effectively graphics chips have become limited though highly efficient computation devices.

Programmable Graphics Architecture

Firstly we will consider the structure of the typical programmable graphics sub-system.

In Chapter 1, the classical polygon pipeline was described. Going into more detail, the diagram below illustrates the most important parts (labels in bold denote programmable units).

The host is the computer that controls the graphics card. It issues commands to the card, and also streams of vertices belonging to shapes to be rendered, e.g. triangles strips, lines, etc. The vertex processor performs some transformation on the vertices (controlled by a vertex program) and passes them on to be clipped. Once clipped to the viewport (the bounds of the displayable area) and any other defined clipping planes, the resulting primitives are passed to the pixel processor that takes care of drawing the primitive in the right place in the framebuffer, with some colour or texture applied according to the results of a pixel program.



If a required texture is not present in texture memory, it is fetched from the host machine.

The distinction between texture memory and frame buffer memory is not a strict one. In fact, many vendors offer extensions that allow their graphics cards to render directly to a texture as if it was a frame buffer (though the texture itself may not be used as a texture source while doing this).

Graphics Processor Programs

In terms of executing user-defined programs, graphics processors operate like simple RISC machines. Datatypes supported are short integers (e.g. 16 bit), floating point variables and vectors of up to 4 components containing floating-point values.

Instructions have a triadic format with two operands and one result, similar to RISC instructions.

Instructions can operate on the vector types, e.g.

```
ADD v1, v2, v3 ; v3 := v1 + v2
```

The above operation would make each element of v1 equal to the sum of the corresponding elements in v1 and v2.

The components of a vector may be *swizzled*, that is arbitrarily shuffled during an operation, e.g.

```
MUL v1.xxx, v2.zyx, v3.xyz
```

This operation would make v3 equal to reversing the components of v2 and multiplying them by the first element of v1.

Conditional program flow can be simulated by using predicated variables. These are variables set as the result of comparison operations and used to select between multiple results. For example, we can simulate the operation of a maximum instruction with the following code:

```
CMPGE r1, r2, rC ; if(r1>=r2) rC = 0xffffffff; else rC = 0x000000;
AND r1, rC, r3 ; r3 = r1 & rC (result is r1 or 0)
ANDN r2, rC, r4 ; r4 = r2 & rC (result is 0 or r2)
OR r3, r4, r5 ; r5 = r3 | r4 (result is r1 or r2)
```

This is less efficient than a dedicated instruction, but does allow us to make use of conditional expressions and simulate branches in program flow. This technique is sometimes used in high performance software to prevent unpredictable branches from disrupting instruction execution.

Program Input and Output

GPU programs operate by having input data from the previous stage in the graphics pipeline placed in their input registers, and then after executing they put their output into their output registers, to be forwarded to the next stage.

In the case of vertex programs, the data inputted and outputted are vertices being transformed. For pixel programs, the input data is composed of the world coordinate, the surface normal, the primary and secondary fragment colours and the current texture coordinates. The output of the pixel program is the colour, depth and stencil values for that pixel in the framebuffer.

Comparison with PC System

Having established the architecture of typical state of the art graphics hardware, it is interesting to compare real world published specifications with standard computer systems. In this case I will compare the ATI Radeon 9700 Pro with a generic Intel Pentium 4 based PC.

Comparison of PC System and Graphics Hardware resources

	PC System	Graphics Hardware
Total Memory (MB)	256 – 512	128
Memory Bandwidth (GB/sec)	2.13	19.4
Clock speed (MHz)	2000	325
Data parallelism	ALU 2x integer MMX: 4x short, 8x byte SSE/SSE2: 8x short, 16x byte 4x float, 2x double	Vertex Units 4x4 float Pixel Pipelines 8x4 float 8x4 byte

Of note is the much greater bandwidth to local memory of the graphics hardware and also the greater parallelism, offsetting the lower clock speed. Not included in this table is the fact that graphics processors have instructions capable of performing geometric operations. Also the latencies of graphics hardware instructions are not currently published.

So, with this architecture, we can consider which problems lend themselves to this structure.

Previous Work

Work has already been done in this area, generally but not exclusively addressing graphical problems. Two implementations of ray tracing have been done, one by Carr et al [2002]^{vi} and one by Purcell et al [2002]^{vii}. M. Rumpf and R. Strzodka have written several papers^{viii} describing the use of graphics hardware for complex image processing.

Given the bandwidth available between the pixel processor and graphics memory seems to be the most valuable resource to make use of. Image and volume processing involves accessing large amounts of data rapidly.

If we were to have data arranged as 2D images, or independent rows of 1D data, we could hold our data in textures and use them to create results or other intermediate values by rendering to similar sized 2D textures.

Image processing is an obvious area where this style of processing is commonly used.

Test Case – Horizontal Edge Detection

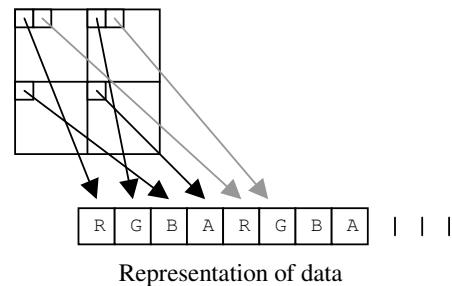
In simple image processing application, a common preprocessing stage before higher-level recognition functions is to do edge detection. This involves finding the absolute difference between adjacent pixels.

In ANSI C, the source code might look like this:

```
int x;
unsigned char src_image*, dst_image*;
..
for(x=0; x<width; x++) {
    dst_image[y*width+x] =
        (unsigned char) abs(src_image[y*width+x] - src_image[y*width+x+1]);
}
```

For a GPU version, we would set things up such that we were rendering to a texture, and have two texture units with the same source texture bound. Furthermore the images that are assumed to be composed of 8 bit intensity values. These will be represented such that pixels from each quarter of the image are stored in the red, green, blue and alpha channels of the texture. A halo (a border of data from surrounding areas) will be present in the quarter images.

```
Edge Detection Code:
vec4 img_x0 = texld(x, y);
vec4 img_x1 = texld(x+1, y);
output = abs(img_x0 - img_x1);
```



Note, that this version is performing 4 edge detect operations per pixel processor. Most programmable GPUs will have 4 – 8 parallel pixel processors, thus 16 – 32 edge detections would occur in parallel.

Performance Analysis:

C version:

For the standard C version, we might assume the compiler generated code might look as follows:

```
movzx eax, byte ptr [esi]      ; eax := src_image[x]
movzx ebx, byte ptr [esi+1]    ; ebx := src_image[x + 1]
sub eax, ebx                   ; eax := eax - ebx
```

```

    cmp eax, 0                ; if(eax < 0) {
    bge not_negative:
    neg eax                  ;   eax := -eax
not_negative:                ; }
    mov [edi], al            ; dst_image[x] = (unsigned char) eax

```

If we assume 3 extra instructions for loop overheads, this loop uses 10 mostly dependent instructions. If we assume a Pentium 4 processor running at 2.0 GHz, with all memory operations in first level cache, the performance would be as follows:

(All operations 32 bit, most dependent, so 10 instructions = 10 cycles)

$$\begin{aligned}
 & 2.0 \text{ GHz} / 10 \text{ cycles} \\
 & = 200\text{M iterations} / \text{sec.}
 \end{aligned}$$

Analysing the memory performance of this loop we have:

2 byte reads, 1 byte write per iteration.

With a 133 MHz 128 bit DDR memory interface (2.1 GB/sec) the performance would be as follows:

$$\begin{aligned}
 & 2.1 \text{ GB/sec} / 3 \text{ bytes} \\
 & = 700\text{M iterations/sec}
 \end{aligned}$$

This implies that even if we were to create a parallel SSE2 version of the inner loop, doing 8 operations in parallel, we would not be able to sustain more than 700M edge detection operations per second for large problem sizes. Using expensive RDRAM memory with a bandwidth of ~4 GB/sec could potentially double this figure, but would require dramatically increased system costs.

For our nominal GPU pixel program, we would imagine a similar instruction sequence to that of the compiled C code.

```

texld v1, tex0, x, y
add x, 1, x
texld v2, tex1, x, y
sub v1, v2, v1
abs v1, v1, output

```

If we assume 5 instructions for this iteration, handling 4 pixels per iterations, with looping handled by the pixel program framework, and 8 pixel processors running at 325MHz we get the following performance measure:

$$\begin{aligned}
 & 325\text{MHz} * 8 \text{ pixel processors} * 4 \text{ pixels/iteration} / 5 \text{ instructions/iterations} \\
 & = 2,080\text{M iterations/second}
 \end{aligned}$$

The bandwidth requirements of this would be $2080M * 3 = 6,240$ MB/sec. With a memory interface supplying 10 – 20 GB/sec, this does not restrict the maximum instruction rate.

Implementation and Performance Issues

The approach for the graphics program used in the test case is not without potential difficulties. For the performance described, it requires all data to be in graphics memory in the format described. It is not clear that this format would be suitable for other operations, such as blurring or thresholding.

Also, the code used is not standard C, though a high level shading language could be used. There are two ways to more easily exploit the graphics processing resource:

- Use libraries that implement a variety of useful algorithms within a particular domain. Image processing is an obvious area where this would be applicable.
- Define a macro language for C programs which abstracts the capabilities of the hardware. This could be done by defining array manipulation macros to construct textures behind the scenes and apply graphics programs derived from templates of expressions.

Combined Performance

Overall performance may not depend entirely either on the host system or the graphics hardware. It is possible to imagine e.g. an image-processing library that has both native system and also graphics hardware implementations of various filters. Such a library could on an appropriately equipped system, execute both implementations simultaneously. The host processor would lose some performance by having to issue commands to the graphics card, but the proportion of these compared to the volume of data potentially processed would more than make up for the reduction in speed.

Ideally, the library would perform some kind of load balancing strategy to ensure optimal utilisation of available resources for the particular task being performed.

As the price of system processors is highly non-linear with regard to clock speed, a large premium always being charged for the fastest part, use of a graphics processor could be a way to boost application performance for little additional investment.

Conclusion

Although this is a brief report and there has not been time or resources to conduct detailed testing of performance with real world sample applications, however, we have seen that GPUs have significant computational resources at their disposal and for some applications could provide an inexpensive boost to performance over standard desktop processors.

The real issue appears to be how much of peak performance is available for computation and how difficult is it to implement algorithms using programmable graphics hardware. If the actual performance is a fraction of the impressive peak, and programming is very difficult then using programmable graphics hardware is not a viable proposition.

With ten times the memory bandwidth and floating point performance apparently greater than a current desktop machine, it definitely seems worthwhile investigating the possibilities further.

Further Work

No assumptions about actual performance are possible without performing real tests. A reasonable test would be to choose a common though simple benchmark, involving a task such as a Fourier transform or a sequence of image processing filters. The performance could be compared against a current model of PC system or workstation, running both standard C and also hand-tuned versions.

It would also be useful to investigate how effectively the host system can perform useful work while the graphics processor is operating.

References

- ⁱ PIXAR, INC.
<http://www.pixar.com>
- ⁱⁱ ID SOFTWARE.
<http://www.idsoftware.com>
- ⁱⁱⁱ AUTODESK, INC.
<http://www.autodesk.com>
- ^{iv} ADVANCED VISUALISATIONS SYSTEMS, INC.
<http://www.av.s.com>
- ^v VTK BY KITWARE, INC.
<http://www.kitware.com/vtk/index.html>
- ^{vi} CARR, N. A., HALL, J. D. AND HART, J. C. 2002. The Ray Engine. Tech. Rep. UIUCDCS-R-2002-2269, Department of Computer Science, University of Illinois,
http://www.cs.uiuc.edu/Dienst/UI/2.0/Describe/ncstr1.uiuc_cs/UIUCDCS-R-2002-2269
- ^{vii} PURCELL T.J., BUCK I., MARK W. R., HANRAHAN P. 2002 Ray Tracing on Programmable Graphics Hardware, ACM Transactions on Graphics. 21 (3), pp. 703-712, 2002. (Proceedings of ACM SIGGRAPH 2002).
<http://graphics.stanford.edu/papers/rtongfx/>
- ^{viii} RUMPF M, STRZODKA, R. PDEs in Graphics Hardware – Collected papers
<http://numerik.math.uni-duisburg.de/research/research-sites/robert/gpu/gpu.htm>