GPU for CAD

Thomas R. Kurfess[1], Thomas M. Tucker[2], Koushik Aravalli[3] and P. M. Meghashyam[4]

[1]Clemson University, kurfess@clemson.edu
[2]Tucker Innovations, tommy@tuckerinnovations.com
[3]Clemson University, karaval@clemson.edu
[4]Clemson University, mpanyam@clemson.edu

ABSTRACT

Due to the explosive market growth in computer gaming, the underlying technology of Graphical Processor Units is also exploding in terms of new capabilities and raw processing power. While the primary target of the growth in GPU capabilities is computer games, computer-aided design applications stand to gain substantial benefits as well. This paper outlines the key features of the new breed of GPU that are significant for the CAD researcher and commercial developer.

Keywords: GPU, Programmable Shaders, Texture Maps, Depth Buffers

1. INTRODUCTION

Graphics Processor Units have evolved from being fixed function pipelines to fully programmable, floating point pipelines. They are highly optimized for fast rendering of geometric primitives and image generation in computer gaming. Recently, they have been used for applications beyond graphics such as general purpose computation as well as scientific computation including fluid flow simulation, cloud dynamics, finite element simulation and many other applications. In this paper, we discuss methods of utilization of various features of commodity graphics processors in Computer Aided Design.

In Section 2, we begin with a description of modern graphics architecture, the working of vertex and pixel shaders and their application in CAD. The next section gives an insight of the role of depth buffers in traditional graphics and their applications to CAD. Section 4 covers other features of the GPU in non-graphic applications. Section 5 discusses the current trends in the development of graphics hardware. In the last section, we have given a brief overview of a shader debugger and the steps to get started with a graphics software development kit.

2. PROGRAMMABLE SHADERS

Roughly five years ago, graphics card manufacturers began allowing portions of the graphics pipeline to be modified. The existing graphic cards were successful in offloading 3d vertex transformations and lighting on CPU and expanding the set of math operations for combining the pixel colors. But the drawback was these operations were not programmable. The NVIDIA’s GeForce3, MS-XBOX and ATI’s Radeon 8500 were the first to launch vertex programmable and pixel configurable hardware. The later generation graphics cards provided both vertex and pixel level programmability. Existing graphics cards now in the market can perform more than 500 GFLOPS, which is way ahead of currently available higher end CPU.

Using any CAD package, when a model is created, the object rendered on the screen is a triangle mesh approximation of the actual model. Creating and transforming the vertices of these triangles to obtain real time visual effects is computationally expensive on a present day CPU running CAD packages. The Graphics Processor Unit has a fixed pipeline through which it is possible to process groups of vertices in parallel. The graphics pipeline is a cycle of stages operating in parallel and in a fixed order. The only input to the starting stage is vertices and the succeeding stages take the output value of their preceding stage for the next set of operations.

The graphics pipeline above is a programmable pipeline. The earlier designers hard coded only specific programs onto the GPU chips and the end users were restricted to only these mentioned programs for rendering the scene. To overcome this constraint programmable shaders are developed. Programmable shaders, both vertex and pixel shaders, in the pipeline are responsible for the fast processing operations. Shaders are graphics programs, which describe many complex properties of the vertices. Earlier shaders were written in assembly language, which restricted the programmer to use its fixed functions. The introduction of programmable shaders made it possible to work with a set of vertices at a time.

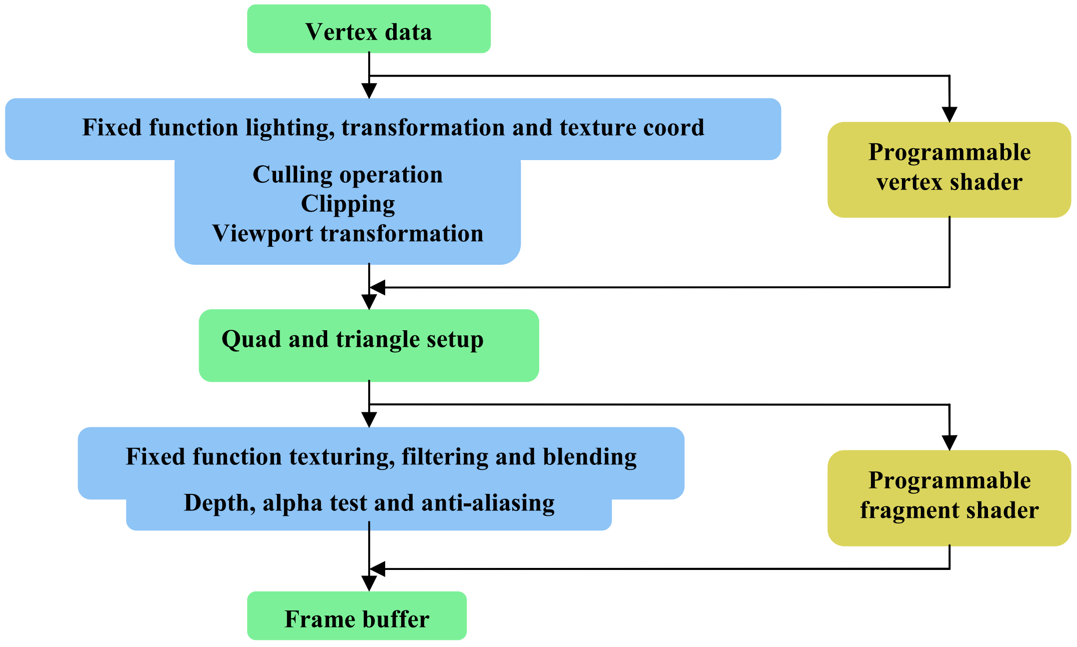
Vertex data
Fixed function lighting, transformation and texture coord
Culling operation
Clipping
Viewport transformation
Programmable vertex shader
Quad and triangle setup
Fixed function texturing, filtering and blending
Depth, alpha test and anti-aliasing
Programmable fragment shader
Frame buffer

Fig. 1: Graphics pipeline.

The data entering the graphics pipeline as per-vertex data undergoes vertex processing, primitive processing and pixel processing. In these stages transforming, clipping, culling, per-pixel color adding, alpha testing, depth testing, stencil testing, and blending are performed on each vertex and pixel. The vertex shaders and the pixel operation by pixel shaders program mentioned operations on vertices.

2.1 Vertex Shaders

Vertex shaders allow the traditional portion of the graphics pipeline where vertices are transformed to be modified enhancing the scope for the programmer to create his own effects like [20] or increasing the computational power of the existing graphical effect. Vertex shaders are prime examples of new functionality in Graphics Processing Unit. Data from a single vertex can be employed to control the effects by loading new set of instructions into shader memory. A vertex shader makes it possible to create an effect on a single vertex. Specified vertex data is sent into vertex shader, mathematical operations are applied on it and modified values are the output from the vertex shader. The following diagram presents basic parallel architecture of programmable vertex shader.

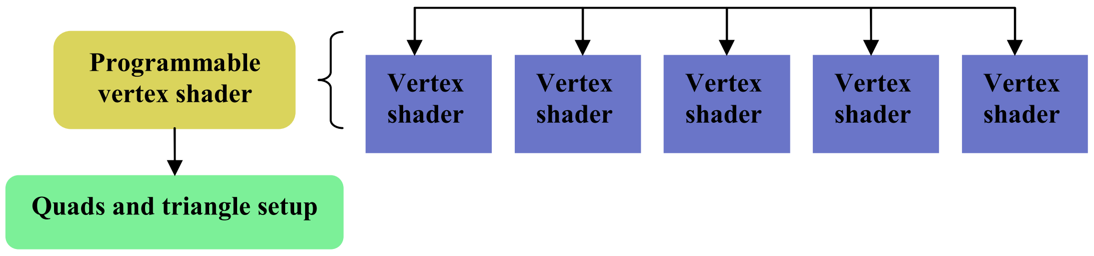

Fig. 2: Vertex shader hardware design.

In terms of the architecture of the GPU, the vertex shaders are a set of registers performing actions in parallel. Typically there are at least 16 registers working independently. Generally when a whole set of vertices are sent into vertex shader the modified output values will be interpolated across the model being rendered. A frequently performed operation in CAD is modification of the viewing transformation of the model. The transformations along with the existing lighting effect, texture effects, and other surrounding environment of an assembly is computationally expensive process. All the vertices specified have to be transformed in the assembly mesh of the model. Taking advantage of the graphics hardware for this action, vertex shader can be programmed to perform the transformations on all the vertices in parallel stored in the vertex buffer. The capability of the vertex shader to perform operations in parallel depends on the number of instructions the graphic card can handle. Vertex shaders are of different configurations depending on the number of instruction slots the graphics card hold. Tab. 1 below highlights the features and the significance of the vertex shaders “vs_2_x”, “vs_3_0” and “vs_4_0”.

Vertex Shader Features	**Shader 2.0 vs_2_x**	**Shader 3.0 vs_3_0**	**Shader 4.0 vs_4_0**	**Description**
Shader Length- Max no. of instructions executed	256 instruction	65535 Instructions	65535 Instructions	More instructions allow more detailed character animation
Dynamic branching- conditional statements	No	Yes	Yes	Saves Performance by skipping animation and calculation on irrelevant vertices
Instancing support	No	Required	Required	Allow many varied objects to be drawn with only a single command
Vertex texture	No	Any number of lookups from up to 4 texture	Any number of lookups from up to 128 texture	Allow displacement mapping and particle effects

Tab. 1: Vertex shader features.

A summary of the inputs, outputs, and traditional functionality provided by a vertex shader is shown below:

Input:

Vertex position
Vertex normal
Primary and secondary colors
Texture co-ordinates

Output:

Transformed, homogenized vertex position
New vertex color
New texture co-ordinates

Traditional Functionality:

- Applying Projection and model view matrix to vertex coordinates.
- Applying texture matrices to texture coordinates.
- Rescaling of normals and transforming them to eye coordinates.
- Automatically generating texture coordinates.
- Performing light and material calculations on each vertex

2.2 Pixel Shaders

The new set vertices from the vertex shader are ready to be rasterized according to the resolution of the render target. There are series of tests the pixels undergo before they are rendered. Each of these pixels carries information about its color intensity, shade and light. For each pixel the pixel shader is executed and only after this the model is rendered on the screen. The architecture of programmable pixel shader is as shown in Fig. 3.

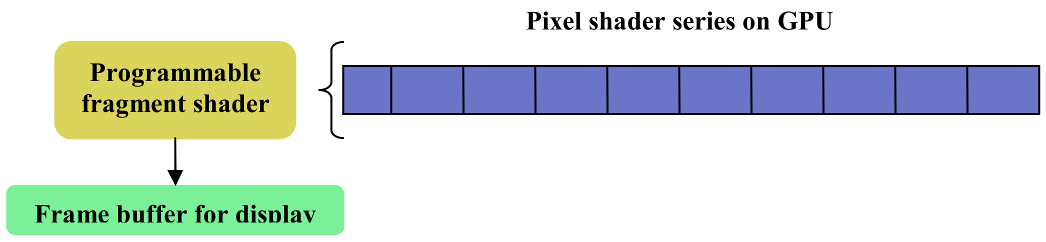

Fig. 3: Pixel shader hardware design.

Programmable pixel shaders are a set of independent shader processors working in parallel. Like vertices stored in the vertex buffer, pixels are stored in a texture and retrieved when needed. A 2D texture is a rectangular image of the order mXn stored in the video memory. To avoid reading exact dimensions of the texture, graphic cards access normalized floating point texture coordinates [0, 1] X [0, 1]. Pixel shader reads texture samples and combines them with the other input to form the final color value. The final color of the pixel is passed to the final stage of the rendering pipeline, like stencil and depth testing. Applying texture to the model influences the color of the pixel. The final color of the pixel will be the blending function of the model and the texture. A pixel shader essentially replaces this portion of the graphics pipeline that determines the final color value for a pixel. The vertex shader calculates the per-vertex light and color values, but the job done by the vertex shader is minimal. This is the reason why the output of a vertex shader provides the inputs for a pixel shader. The positions of the pixel (vertex), color value, normal of the pixel, texture co-ordinate are possible input values to the pixel shader and possible outputs are per-pixel lighting value, depth of the field. For example from [10], if the output of vertex shader is two colors and three texture coordinates, then the input to the pixel shader will be two colors and three texture coordinate values per pixel. The table highlights the features and description of pixel shaders currently available from [16]:

Pixel Shader Feature	**Shader 2.0**	**Shader 3.0**	**Shader 4.0**	**Description**
Shader length	96	More than 65535	Unlimited	Allow more complex shading
Dynamic branching	No	Yes	Yes	Saves performance by skipping complex shading on irrelevant pixels
Shader anti-aliasing	Not supported	Built in derivative instructions	Built in derivative instructions	Allowing shader frequencies which eliminate jagged lines in the image
Back face register	No	Yes	Yes	Lighting and transformation on back faces of the geometries in a single pass
Interpolated color format	8 bit integer	32 bit floating point	4096 bit floating point	Indicates allowable range of lighting & color values. Higher the value more number of colors
Multiple render targets	Optional	4 required	32	Allows advanced lighting algorithms to save filtering and vertex work.
Texture coordinate count	8	10	32	More pixel inputs allow more realistic rendering

Tab. 2: Pixel shader features.

A summary of the inputs, outputs, and traditional functionality provided by a pixel shader is shown below:
Input:

Interpolated vertex color
Interpolated vertex normal
Filtered texture co-ordinates
Initial depth value

Output:

Final pixel or fragment color
Final pixel or fragment depth

Traditional functionality:

- Applying color values to each pixel by blending texture values and scene
- Performing light calculations on each pixel or fragment

2.3 Shaders Applied to CAD

Triangle meshes are important surface approximations used in many situations. Often complex triangle mesh models consisting of large numbers of triangles have to be processed. Hence, the parallel processing provided by the GPU and its native ability to work with triangle meshes makes a marriage between GPU and triangle mesh processing algorithms attractive. Martin Marinov, Mario Botsch, Leif Kobbelt in [13] addressed the application of GPU as an efficient technique for evaluating multi-resolution deformations of high-resolution triangle meshes.

Shaders can be applied in CAD applications to improve the realism of the scene. Generation of the shadows is one of the ways to make the scene more real. Soft Shadows for the boundaries for a dynamic scene was proposed by [12] Peter Tamas Kovacs and Gyorgy Antal. The algorithm is based on determining the areas of the world that are in shadow using shadow volumes. The main idea is to use the contents of the stencil buffer as a shadow mask, but blur it before using. In order to obtain soft shadows the scene has to jitter. The algorithm uses stencil buffer and it cannot be blurred directly. In its last step, pixel shader is combined with the ambient image, the diffuse-specular image and the blurred shadow mask. Even though shadow mapping is not of interest in CAD, the concepts used here can be applied for depth test for occluding geometric parts in an engineering sense such as an assembly.

Generating virtual assemblies of individual parts using CAD software can benefit from collision detection functionality to aide in mating the part geometries. In most of CAD packages supporting assemblies, situations where parts overlap inappropriately are not prevented. Collision detection between the mating parts would avoid these problems. Alexander Greb, Michael Guthe proposed [8, 15] GPU-based collision detection method for deformable parameterized surfaces that can easily be combined with the aforementioned approaches.

Parallel sorting of the data on the stream processors like vertex shader and pixel shader is of high interest. In rendering large set vertices of geometry will fall in this case. The complexity of the algorithm for “n” values utilizing p processors is O ((n log n)/p). Greb and Zachmann [6] proposed an algorithm for stream processing based on adaptive bi-tonic sorting method. When implemented on GPU, time taken to obtain the results is faster when compared to the routine sorting techniques GPU follow. The data is set into the GPU stream is either a 1D or 2D or 3D set of textures. Discussion on these data set ranges, sent in to the processors, is also given.

In CAD applications, it is often required to solve for parameters by using various linear algebra techniques. For example, the points of intersection of the geometries require solving set of equations formed out of the vertices. Naga K Govindaraju *et. al.* [4] implemented LU decomposition and Jin Hyuk Jung in [11] implemented the Cholesky decomposition technique on the GPU to solve a large system of linear equations. Both these implementations are essential for CAD applications such as finite element analysis, computational fluid dynamics, and least squares approximation.

A GPU finite element implementation to solve the linear heat equation has been demonstrated by Rumpf and Strzodka. Their algorithm utilizes the shader blending functions and texture environment functions to not only perform algebraic operations on the vectors which are represented as images but also optimize operations by reducing the number of rendering passes. They achieved more than 300 MOP/s in a single iteration of the Jacobi solver.

3. DEPTH BUFFERS

3.1 The Purpose of a Depth Buffer

A depth buffer, also called the z-buffer is an integral part of the frame buffer, which is used to accelerate 3D rendering in graphics. It does not contain depth image data but rather depth information about a particular pixel. It is primarily used for hidden surface removal also called z culling. It stores the distance from the eye to each visible point in the image. When a pixel is to be rendered, its depth value is first compared with the pixel already in that position in the frame buffer. The pixel is drawn only if its depth value is less than the existing pixel. Thus, a depth buffer plays an instrumental role in avoiding draw calls for objects occluded by other objects having nearer depth values. Apart from z culling, the depth buffer is also useful in other graphics applications. In depth map shadows, the depth buffer is used to store the z-coordinates of all vertices in the image when rendered from the light source. These values are compared with the ones obtained when the image is rendered from the viewer's eye to determine which vertices are to be rendered dark. The depth buffer is also used to obtain effects such as viewing the internal parts of object rendered also called a "cutout". This is achieved by drawing a cutting plane by disabling drawing to the color buffer, and then setting the drawing to the back buffer again.

3.2 Depth Buffer Applied to CAD

The depth buffer can be utilized in many CAD related applications. Almost all CAD packages involve 3D viewing of surfaces or assemblies. The depth buffer can be used for hiding surfaces which are not required to be viewed thus avoiding unnecessary render calls.

Programmable graphics hardware accelerated algorithms have been developed to determine the castability of geometric parts and assist with part redesign by McMains, Khardekar and Burton [14]. These algorithms are capable of identifying and displaying undercuts and minimum and insufficient draft angles graphically. In this approach, they utilize the occlusion query functionality (depth buffer) of the graphics card to determine whether or not a part is castable in a given direction when the object is looked upon from the mold removal direction. This involves rendering the part geometry and storing the distance to the visible part facets also called "up-facets" from the eye-point in the depth buffer. The geometry is then re-rendered with depth test enabled. In this pass, only those portions of the up-facets that were invisible earlier are rendered. Thus, if any pixels are rendered in the second pass, the object is not castable in that direction. The implementation of this algorithm on a NVIDIA Quadro FX 3000 series GPU was shown to test the castability of parts with over 20,000 faces in less than a millisecond per direction. The algorithm is further extended to highlight the non-castable features of the part to allow the designer to make the necessary changes. The depth buffer obtained from the first rendering pass is transferred to a depth texture and the orthogonal viewing matrix for the current camera position is saved. This procedure is repeated from the opposite casting direction thus allowing the designer to rotate the object and examine the undercuts in real time.

A graphics hardware–assisted approach to 5-axis surface machining has been developed by Gray *et. al.* [5] to build upon a tool positioning strategy called the Rolling Ball Method. In this method, a ball of varying radius is rolled along the tool path such that a small portion of the surface of the ball is used to approximate a small portion of the surface being machined by the cutting tool. The radius of the rolling ball is computed by checking a grid of points in the shadow of the cutting tool on the work piece surface. The hidden surface removal property of the depth buffer is used to expedite this checking process by clipping the part of work piece that is not in the shadow of the tool. This approach was shown to eliminate the need for parameterization of the surface to be machined, and thus allow the machining of multiple patch-triangulated surfaces.

Dokken *et. al.* [3] presented an idea of utilizing the GPU as an accelerator for CAD-type intersection algorithms. They propose the use of the fragment processor for subdividing surfaces to create a tight hull containing the surface, and a tight hull containing the normal field of the surface. The depth buffer is used as a tool for testing for overlap of the extent of hulls containing the surfaces. For two surfaces to intersect in a closed loop, their normal fields have to overlap. The result is that using the depth buffer to determine that the hulls containing the normal field of surfaces do not intersect will rule out the possibility of closed intersection curves.

A mechanistic model based on an adaptive and local depth buffer to calculate milling forces when machining a part on a multi-axis milling machine has been developed by Roth *et. al.*[18, 19]. Their paper presents a novel method of calculating chip geometry and volume of material removed during machining in order to determine the cutting forces.

The terms “adaptive” and “local” depth buffers are used as the depth buffer is changed to be constantly aligned with the tool axis and sized to the tool instead of the work piece respectively. Previous tool positions are rendered to the scene and the depth buffer is saved. The current tool position is then rendered and the depth buffer is saved again to obtain the in-process chip geometry, which is the difference between the two states of the depth buffer. However, this model is limited to only flat end mills and inefficient as most of the depth buffer holds previous tool positions as it is sized to cover the tool. The implementation has been modified to overcome the limitations of the model. The algorithm is updated to handle more complex tool shapes. The depth buffer is sized to the cutter teeth, thereby improving the memory requirements resulting in efficient usage.

4. OTHER FEATURES

NURBS based surface representation with the triangular meshes has an advantage of being flexible and efficient. Botsch, Bommes and Vogel [1] proposed a simple and generic method for computing the distance of a given polygonal mesh to the reference surface, based on a linear approximation of its signed distance field. This method used modern GPUs to perform up to 3M triangle checks per second, enabling real-time distance evaluations even for complex geometries. The paper discusses accurate high-quality distance visualization of dynamically changing meshes at a rate of 15M triangles per second. The method describes the implementation of a generic distance check to test whether the triangle is within a tolerance volume around reference surface using GPU. The data set is loaded onto a 3D texture with a special color to distance values greater than the prescribed tolerance, render the candidate triangle and detect whether this color appears in the frame buffer. As soon as the pixel is rendered, using alpha tests all transparent pixels are discarded.

Due to the fine sampling of trim boundaries, trimming complicates the tessellation. Pabst [17] and his team presented the concept of extended graphics pipeline that allows the rendering of complex primitive such as parametric and implicit surfaces. Their pipeline adds an intersection stage between the Rasterisation stage and the fragment program. The intersection stage reconstructs corresponding ray from the viewpoint for each fragment generated from the Rasterisation unit and computes the intersection with the surface contained in the bounding volume of the object to be displayed. The intersection point and the normal are passed into the fragment program. This integration into the graphics pipeline combines its high efficiency with the advantages of ray casting. They address the direct real-time rendering of trimmed NURBS surfaces. They use the Newton iteration for the intersection test and Iterative Bezier Clipping for exact trimming of the NURBS surfaces.

Michael Guthe *et al* [7] developed a GPU-based trimming and tessellation method for NURBS and T-Spline surfaces which approximated NURBS with a bi-cubic hierarchy of Bezier patches on the CPU with certain geometric error of approximation and then tessellated these on the GPU. Their novel approach to solving the trimming problem is a GPU-based algorithm that allows representing the trimming region by an appropriate black and white texture of sufficient resolution. For each texel, the color determines if it is inside or outside the trimmed region. They show that one-bit masking texture is used for this purpose. The trim texture resolution is calculated additionally to the grid resolution. Since this approach only took the geometric error of approximation into account, the various illumination artifacts introduced by the chosen bi-cubic approximation and the subsequent tessellation were neglected.

To address this issue Guthe *et al* [9] devised a method, which takes into account normal error of approximation and also sets up a new error measure to calculate the required grid resolution for the bilinear approximation. The method is called “Appearance Preserving” as it involves preserving the normal of the approximating as well as the original surface by preserving the first derivative of each iso-parametric curve. The error of approximation is mapped from first derivative space to object space using a Taylor expansion and the maximum derivation deviation error is assumed to occur at a point where the second derivative of the curve has its maximum. They demonstrated that this method of preserving the normal allows GPU-based NURBS trimming and tessellation with guaranteed visual fidelity.

5. THE FUTURE OF THE GPU

Increased interests in games using 3D graphics coupled with ever increasing use of GPUs in non-graphic applications have escalated the need for higher computational capabilities. Fig. 4 indicates the trends in development of GPUs from August 2005 till date [23]. It is observed that there is a rapid increase in the Memory (MB) and Memory bandwidth (GB/s) every six months. The latest NVIDIA graphics card, the 8800 GTX, provides their new CUDA (Compute Unified Design Architecture) technology that allows the card to be programmed in ‘C’ like threads. It is embedded with 128 individual stream processors running at 1.35 GHz. The memory bandwidth is a staggering 88.4

GB/s, which accounts for the spike in the plot. NVIDIA's major competitor, ATI, will likely follow with similar technology in the months ahead. Moore's Law seems to apply to the GPU but at a rate much higher than that for the CPU. If these trends continue, the performance advantage for applications that utilize the computational capability of the GPU will continue to grow.

Computer Aided Design is one such application that would benefit greatly from these developmental trends of the GPU. CAD systems are used to create detailed geometric models which serve as a starting point for diverse analysis tools such as computational fluid dynamics, stress analysis, geophysical data exploration, and computational electromagnetics or acoustics and manufacturing processes such as numerical-control machining, injection molding and casting. The problems in CAD systems currently faced by the research community are the mathematical issues concerned with the computation, representation and manipulation of complex geometries, which hamper efficient solution procedures. Many successful attempts have been made to utilize the GPU in addressing these CAD issues [14], but with the current trend, CAD researchers have numerous opportunities to device methods to harness the raw computational power and capabilities of the GPU to obtain fast and accurate CAD solutions.

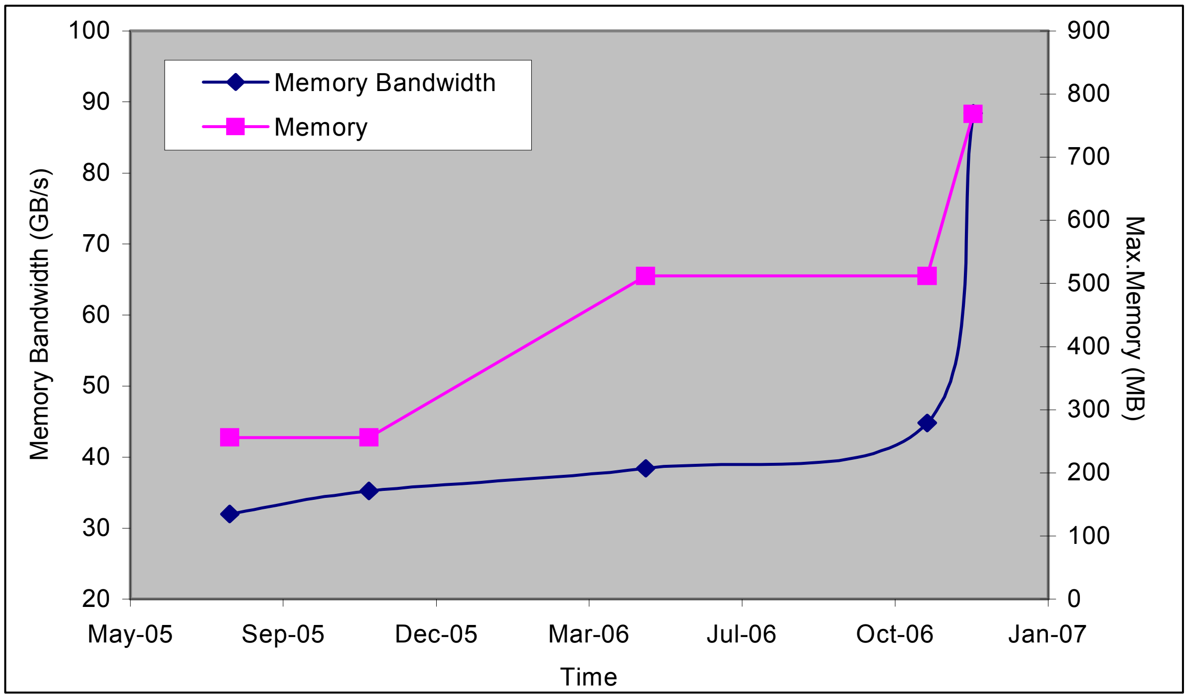

Fig. 4: Trends in GPU processing – data from NVIDIA and ATI websites.

6. HOW TO GET STARTED

There are two Application Programming Interfaces (APIs) that can be used to program a graphics card: OpenGL and Direct3D. Though the frameworks of these APIs are similar, OpenGL is cross-platform while Direct3D is limited to a Windows platform. However, either API is suitable for GPU research.

Microsoft, NVIDIA and ATI (AMD) corporations have developed the DirectX, NVIDIA and ATI (Radeon) Software Developer's Kits (SDK), which are freely available and provide a useful starting point to GPU programming. ATI has a shader development environment called the RenderMonkey Tool suite and an SDK to support it. These SDKs can be downloaded and installed from the links given below

http://msdn.microsoft.com/directx

http://developer.nvidia.com/object/sdk_home.html

http://ati.amd.com/developer/radeonSDK.html

http://ati.amd.com/developer/rendermonkey/index.html

http://ati.amd.com/developer/rendermonkey/sdk.html

The NVIDIA SDK provides a suite of tools for content creation, performance analysis and general software development using both OpenGL as well as DirectX. It includes a large number of code samples, effects, white papers and more to help the user take advantage of the latest technology. The code samples include 3D graphics samples, video and image processing samples and implementations of General Purpose computation on the GPU (GPGPU).

The ATI SDK provides developers with an insight of getting the most of latest ATI products using DirectX and OpenGL APIs. It is a compilation of new and previously posted information from developers. It consists of a number of sample applets/codes that are updated often. The RenderMonkey Tool suite provides a rich shader development environment which facilitates the collaborative creation of real-time shader effects. It contains a full-featured shader editor integrated with a shader compiler and preview window to provide immediate visual feedback. It incorporates a number of features to aid in shader debugging and optimization. The SDK for RenderMonkey allows the developer to create custom plug-ins to solve un-anticipated problems and various other features including importing/exporting application's data scripts directly to/from RenderMonkey, creating GUI editors and creating custom geometry and texture loaders.

OpenGL is a simple committee developed API, which, at its most basic level is a specification that describes a set of functions and the behaviors that they must perform. Hardware vendors create libraries of functions to match the OpenGL specification. OpenGL's basic operation is to accept primitives such as points, lines and polygons and convert them to pixels. The specification is currently being overseen by the OpenGL Architecture Review Board (ARB) which consists of a set of companies of vested interest in creating a consistent and widely available API. The official OpenGL is http://www.opengl.org/.

The website not only contains information about using the latest developments in the field of graphics, developer tools and other features but also provides a good starting point for beginners by providing useful links to a large number of OpenGL code samples and tutorials. Developing an application involves

- *Downloading the following:*
  OpenGL utility toolkit, GLUT- a window system independent toolkit for writing OpenGL programs and Interface toolkits GLX- a toolkit used on Unix OpenGL implementation, GLU- OpenGL utility library consisting of a set of functions to create texture mipmaps from a base image, map coordinates between object and screen space and draw quadric surfaces and DRI- a software architecture for coordinating the Linux kernel, X window system, 3D graphics hardware and an OpenGL based rendering engine.
- Visiting http://www.opengl.org/code/category/C22, which includes samples covering fundamental topics such as modeling, lighting, depth buffering and texture mapping which are useful for anyone just starting out with OpenGL.

The DirectX SDK [2]is a self-explanatory tool containing numerous features such as programming guide, graphics tools including the shader compiler and debugger, tutorials and samples and reference. The following steps should be followed to get started with Direct3d

- Install the latest Microsoft .NET Framework runtime that includes the .NET runtime, a set of dynamic link libraries required for code applications. Download and install the latest DirectX SDK from the link provided.
- The samples provided in the SDK must be built using the Microsoft Visual Studio .NET 2003. Individual sample folders contain a variety of applications specifically designed for managed code (written in high level programming languages such as C++, C#, J # and a few others) along with a .NET 2003 project and solution.
- The tutorials include sample applications that give step-by-step procedures of creating a complete working application starting from creating a device to using transformations, texture maps and meshes.

The samples in the SDK contain simple codes exhibiting the various features and effects that can be achieved using shaders, meshes and lighting.

Shader Debugger:
Microsoft Visual Studio.NET supports debugging assembly-level and high-level language vertex and pixel shaders and effects by using the DirectX extension also called the Shader Debugger. The shader debugger allows performing certain operations within the Integrated Development Environment (IDE),

- Debug vertex and pixel shaders.

- Debug High-Level Shader Language (HLSL) and assembly shaders.
- Look at the contents of variables, registers and device state.
- Use the IDE to run the code, step through the code a single line at a time, or break where necessary.
- Set breakpoints on lines or on a pixel area.
- Perform expression evaluation in a watch window.
- View the assembly code generated by the HLSL compiler.
- View loaded texture(s) on the Direct3D device.
- View render target(s) as pixel shaders write to them.

See syntax coloring for recognized file types including effect (.fx) files, assembly vertex shader files (.vsh) and assembly pixel shader files (.psh).

7. REFERENCES

[1] Botsch, M.; Bommes, D.; Vogel, C.; Kobbelt L.: GPU Based Tolerance Volumes for Mesh Processing, Computer Graphics and Applications, Proceedings. 12th Pacific Conference, 2004, 237-243

[2] DirectX-SDK Apr-2006 documentation, © Microsoft Corporation.

[3] Dokken, T; Hagen, T.R.; Hjelmervik, J.M.: The GPU has a high performance computational resource, Proceedings of the 21st Spring Conference on Computer Graphics, 2005, 21-26.

[4] Galoppo, N.; Govindaraju N.K.; Henson, M; Manocha D.: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, Supercomputing, Proceedings of the ACM/IEEE 2005.

[5] Gray, P. J.; Ismail, F.; Bedi, S.: Graphics-assisted Rolling Ball Method for 5-axis Surface Machining, Computer-Aided Design, 36, 2004, 653-663.

[6] Greb, A.; Zachmann, G.: GPU-ABiSort: Optimal Parallel Sorting On Stream Architecture, Parallel and Distributed Processing Symposium, IPDPS 2006 20th International, 2006.

[7] Guthe, M.; Balazs, A.; Klein, R.: GPU based trimming of NURBS and T-Spline surfaces, ACM Transactions on Graphics, 24(3), 2005, 1016-1023

[8] Guthe, M.; Greb, A.; Klein, R.: GPU based Collision Detection for Deformable Parameterized Surfaces, EUROGRAPHICS, 25(3), and 2006, 497-506.

[9] Guthe, M.; Balazs, A.; Klein R.: GPU-based appearance preserving Trimmed NURBS rendering, Journal of the WSCG, 14, 2006.

[10] Introduction to 3D game programming with DirectX 9.0, Wordware Publishing, Inc.; Plano, Texas, 2003

[11] Jung, J. H.: Cholesky Decomposition and Linear Programming on GPU, Scholarly Paper Directed by Dianne P. O'Leary, Department of Computer Science, University of Maryland, 2006.

[12] Kovacs, P. T.; Antal, G.: Soft Edge Stencil Shadow in CAD applications, Third Hungarian Conference on Computer Graphics and Geometry, Budapest, 2005.

[13] Marinov, M.; Botsch, M.; Kobbelt, L.: GPU-Based Multiresolution Deformation Using Approximate Normal Field Reconstruction, Computer Graphics Group, RWTH Aachen, Germany.

[14] McMains, S.; Kardekar, R.; Burton, G.: Finding Feasible Mold Parting Directions using Graphics Hardware, Computer-Aided Design, 2006. 38(4), 327-341.

[15] Meseth, J.; Guthe, M.; Klein, R.: Interactive Fragment Tracing, Visual Computer, 21, 2005, 591-600.

[16] NVIDIA GPU Programming Guide, Version 2.4.0, © 2005 NVIDIA Corporation.

[17] Pabst, H.; Springer, J.P.; Schollmeyer, A.; Lenhardt, R.; Lessig, C.; Froehlich, B.: Ray casting of trimmed NURBS surfaces on the GPU, IEEE symposium on Interactive Ray Tracing, 2006

[18] Roth, D.; Ismail, F.; Bedi, S.: Mechanistic modeling of the milling process using an adaptive depth buffer, Computer-Aided Design, 35, 2003, 1287–1303.

[19] Roth, D.; Ismail, F.; Bedi, S.: Mechanistic modeling of the milling process using complex tool geometry, The International Journal of Advanced Manufacturing Technology, 25, 2005, 140-144.

[20] Technical Brief – NVIDIA nfiniteFX Engine: Vertex and Pixel shaders, © 2005 NVIDIA Corporation

Websites for reference:

[21] www.gamedev.net

[22] www.toymaker.com

[23] www.wikipedia.com