Three-dimensional computer graphics architecture

Tulika Mitra* and Tzi-cker Chiueh

Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA

Three-dimensional (3D) computer graphics hardware has emerged as an integral part of mainstream desktop PC systems. The aim of this paper is to describe the 3D graphics architecture at a level accessible to the general computational science community. We start with the generic 3D graphics rendering algorithm, the computational requirements of each of its steps, and the basic architectural features of 3D graphics processors. Then we survey the architectural features that have been implemented in or proposed for state-of-the-art graphics processors at the processor and system levels to enable faster and higher-quality 3D graphics rendering. Finally, we describe a taxonomy of parallel 3D rendering algorithms that accelerate the performance of 3D graphics using parallel processing.

1. Introduction

UNTIL recently, real-time three-dimensional (3D) computer graphics was available only in very high-end machines from Silicon Graphics Inc. In the last few years however, the PC industry has seen an unprecedented growth of cost-effective 3D graphics accelerators. Because a significant amount of industrial research effort has been invested in powerful 3D graphics cards, it is predicted that the performance of these accelerators will surpass the performance of SGI machines by the year 2001 (ref. 1). 3D graphics applications place a stringent demand on the processing power and on the data transfer bandwidth of the memory subsystem and interconnecting buses. The growing importance of 3D graphics applications has motivated CPU vendors to add new instructions to the existing instruction set architecture, and to develop higher-bandwidth memory and system buses. In fact, the data-intensive nature of 3D applications has been one of the primary motivations behind the introduction of advanced Dynamic Random Access Memory (DRAM) architectures for host memory, and the local memory on graphics cards.

In this article, we start with the basic steps required to render a polygon-based 3D graphics model and their associated and bandwidth requirements. Then we examine the major design issues in generating photo-realistic images on desktop machines in real time, and the architectural innovations that attempt to address these problems. Finally, we present a taxonomy of parallel rendering algorithms, which uses parallel processing hardware to render extremely complicated 3D graphics models.

2. 3D graphics pipeline

Polygon-based 3D graphic rendering is the process of converting the geometric description of a 3D model (or a virtual world) to a photo-realistic two-dimensional image (a 2D array of picture elements or pixels) that can be displayed on a computer monitor. Each pixel represents a colour value consisting of red, green, and blue (RGB) components. The sequence of steps involved in this conversion forms the 3D graphics pipeline, each stage of which can be implemented either in hardware or software.

The input to the 3D graphics pipeline is a virtual world created by application programmers. This world/scene consists of a mathematical representation of a set of objects, their positions relative to each other, an optional set of light sources, together with a viewpoint that provides a camera angle into the virtual world. Objects or primitives are typically represented by a set of triangles for ease of implementation. The description of the 3D model is passed to the 3D graphics engine through a standard Application Programmer Interface (API) such as OpenGL[2] or Direct 3D[3]. The 3D graphics pipeline itself consists of two distinct stages: **geometric transformation** and **rasterization**. The geometric transformation stage maps triangles from a 3D coordinate system (object space) to a 2D coordinate system (image space) by performing a series of transformations. The computation in this stage is mostly floating-point intensive, involving linear algebraic operations such as matrix multiplication and dot products. The rasterization stage converts transformed triangles into pixel values to be shown on the computer screen. This stage involves mostly integer arithmetic, such as simple additions and comparisons. An excellent reference to the 3D graphics pipeline can be found in Foley *et al.*[4].

2.1 *Geometric transformation*

At the input of the geometric transformation stage, each triangle consists of three vertex coordinates, vertex normals and other attributes such as colour. For ease of

*For correspondence. (e-mail: mitra@cs.sunysb.edu) manipulation, vertices are represented in homogeneous coordinates, which are quadruples of the form $\{x, y, z, w\}$, where in most cases w is 1. (The tuple $\{x/w, y/w, z/w\}$ is the Cartesian coordinate of the homogeneous point.) The geometric transformation stage applies a sequence of operations on the vertices of the triangle. Figure 1 shows the geometric transformation part of a typical 3D graphics pipeline which consists of the following stages:

2.1.1 *Model and viewing transformation*: Modelling transformation positions primitives with respect to each other, and the viewing transformation orients the resulting set of primitives to the user viewpoint. These two transformations can be combined into a single multiplication of the homogeneous vertex coordinate by a 4 × 4 matrix, which is implemented as 16 floating point multiplications and 12 floating point additions. Lighting calculation, in addition, requires the transformation of the vertex normal by a 3 × 3 inverse transformation matrix, which costs 9 floating point multiplications and 6 floating point additions.

2.1.2 *Lighting*: This stage evaluates the colour of the vertices given the direction of light, the vertex position, and the surface-normal vector and material characteristics of an object's surface. We will consider here only the most popular shading model, called *Gouraud shading*, which interpolates the colour of the three vertices across the surface. Evaluating the colour of a vertex requires a variable amount of computation depending on the number of light sources and the material properties. We assume the simplest case of a single light at infinite distance, and the material with only ambient and diffuse coefficients. This lighting model calculates the following equation for each R, G, B component:

$$C_{diffuse} \times C_{light} \times (N \cdot L) + A_{reflection} \times A_{light},$$

where C_{light} and $C_{diffuse}$ are the light source intensity and diffuse reflection coefficient; A_{light} and $A_{reflection}$ are the ambient light intensity and ambient light coefficient; $(N \cdot L)$ is the dot product of surface-normal vector and the direction of light vector. $(N \cdot L)$ is calculated only once. However, the rest of the equation should be calculated independently for R, G and B components for each vertex. This requires a total of $(3 + 3 \times 3 = 12)$ multiplications and $(2 + 3 \times 1 = 5)$ additions per vertex.

2.1.3 *Projection transformation*: This transformation projects objects onto the screen. There are two types of projections: (1) *orthographic* projection, which keeps the original size of 3D objects and hence is useful for architectural and computer-aided design; (2) *perspective* projection, which produces more realistic images by making distant objects appear smaller. Each of these transformations again involves a 4 × 4 matrix multiplication. However, as most entries in these matrices are zero, a careful implementation requires only 6 multiplications and 3 additions.

2.1.4 *Clipping*: The application programmer defines a 3D *viewing frustum* such that only the primitives within the frustum are projected onto the screen. This step removes the objects that are outside the viewable area. The algorithm requires one floating point comparison per view-boundary plane, and thus 6 comparisons per vertex. If a triangle is partially clipped, then the algorithm should calculate the position of the new vertices at the intersection of the triangle edge and the view-boundary plane. The number of such operations performed depends on the actual number of triangles that cross the view-boundary planes, which varies from one viewpoint to another. Hence, we will not take this cost into account for our computation requirement calculation.

2.1.5 *Perspective division*: If perspective transformation is applied on a homogeneous vertex, then the w value no longer remains equal to 1. This stage divides x, y, z by w to convert the vertex to Cartesian coordinates.

2.1.6 *Viewport mapping*: This step performs the final scaling and translation to map the vertices from the projected coordinate system to the actual viewport on the computer screen. Each vertex component is scaled by an independent scale factor and offset by an independent offset, i.e. 3 floating point multiplications and 3 floating point additions.

The total computation requirement to perform geometry transformation per vertex is then 46 multiplications, 29 additions, 3 divisions, and 6 comparisons. Modern processors can execute floating point addition, subtraction, comparison, and multiplication operations quite fast using pipelined execution units. Floating point division operation however, is not usually pipelined, and can take as high as 50 floating point addition operations' worth of time. The total floating point operation requirement for a single vertex transformation is then around 130. Today

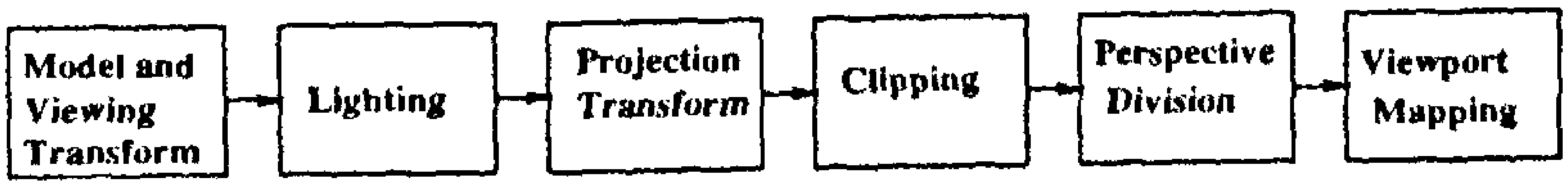

Figure 1. *Geometry transformation stage of a 3D graphics pipeline.*

even a modest scene requires around 1 million vertex transformations per second to achieve a rate of 30 frames per second. This would translate to 130 MFlops (million floating point operations) per second. Today's PCs have sufficient floating point computation power and therefore typically perform the geometric transformation stage in the main CPU.

2.2 *Rasterization*

The rasterization stage comprises two steps. The **scan conversion** step decomposes a triangle into a set of pixels, and calculates the attributes of each pixel, such as colour, depth, alpha, and texture coordinates. The **pixel processing** step performs **texture mapping**, **depth test** and **alpha blending** for individual pixels. Figure 2 shows the rasterization stage of the graphics pipeline.

There are two distinct mechanisms that are quite popular for the scan conversion step: **linear interpolation** algorithm and **linear edge function** algorithm. In linear interpolation-based algorithms[4], the **triangle set-up** step first computes the slopes, with respect to the X-axis, for all the attributes along each edge of the triangle. Next, the **edge processing** step iterates along the edges and computes the two end points of a horizontal pixel segment, called a **span**. Finally, the **span processing** step iterates along each span and computes the attributes for each pixel on the span through linear interpolation (Figure 3).

In linear edge function-based algorithms[5], each edge of the triangle is defined by a linear edge function. The triangle is scan converted by evaluating, at each pixel's centre, the function for all edges, and processing only those pixels that are inside all the edges. The attributes are also computed from the linear functions. Typically, the traversal of a triangle proceeds down from a starting point, and moves outward from the centre line[6]. The centre line shifts to the left or right, until it steps outside of the triangle at any point of time (Figure 4 *a*). To achieve parallelism, the triangle may be traversed one **pixelstamp** at a time, rather than pixel by pixel[6]. A pixelstamp is an array of pixels of dimension $X \times Y$. Evaluation of edge functions for all the pixels within a pixelstamp could start in parallel, and only qualified pixels are sent to the pixel processing stage. Triangle traversal visits all pixelstamps that are completely or partially inside the triangle (Figure 4 *b*).

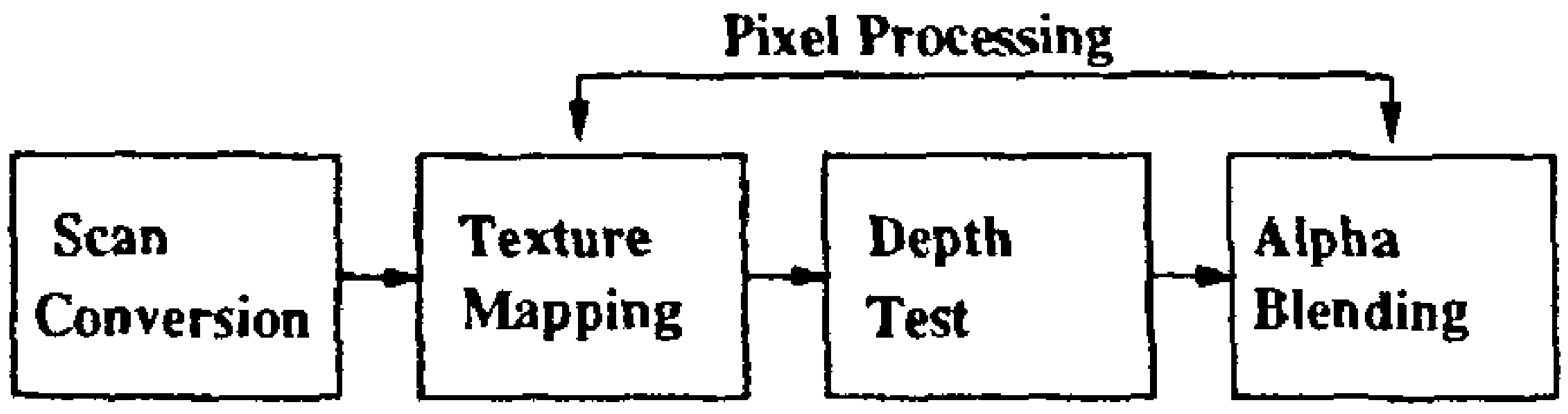

Figure 2. Rasterization stage of a 3D graphics pipeline.

The rasterization stage also includes texture mapping, which is a crucial and widely used technique that wraps a 2D texture image on the surface of a 3D object to emulate the visual effects of complex 3D geometric details, such as wooden surface, tiled wall, etc. Each vertex of a texture-mapped triangle comes with a texture coordinate that defines the part of the texture map to be applied (refer to Figure 5). These texture coordinates are interpolated across the triangle surface via scan conversion. The most popular texture mapping implementation is based on **mip-mapping**[7] (Figure 6), which pre-calculates multiple reduced-resolution versions of a texture image. Each resolution level corresponds to a particular depth. Coarser (finer) resolution levels are used for farther (closer) objects. For a 3D object at a given depth, the mip-mapping algorithm chooses a pair of adjacent resolution levels of the texture image, and performs weighted filtering of 8 texels (texture pixel) from these two resolution levels. This **tri-linear filtering** eliminates visual discontinuities when different mip-map levels are applied on the same object.

Before a pixel is written to the frame buffer, the rendering engine needs to check whether that triangle is actually visible at that pixel, i.e. no other triangle overlaps that pixel making it invisible. This is known as hidden surface removal for opaque objects. The number of overlapping triangles for a pixel is called the **depth complexity** of the pixel. The majority of graphics accelerators achieve hidden surface removal using a **depth/Z buffer,**

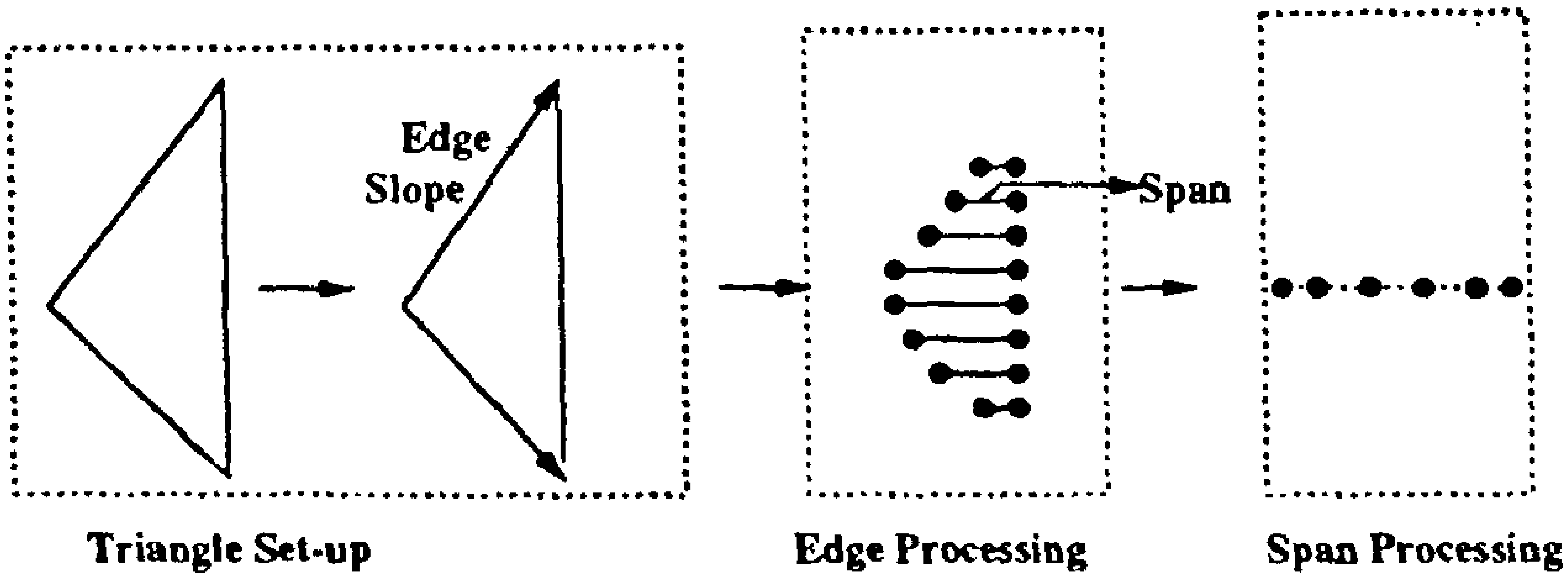

Figure 3. Scan conversion of a triangle using linear interpolation algorithm.

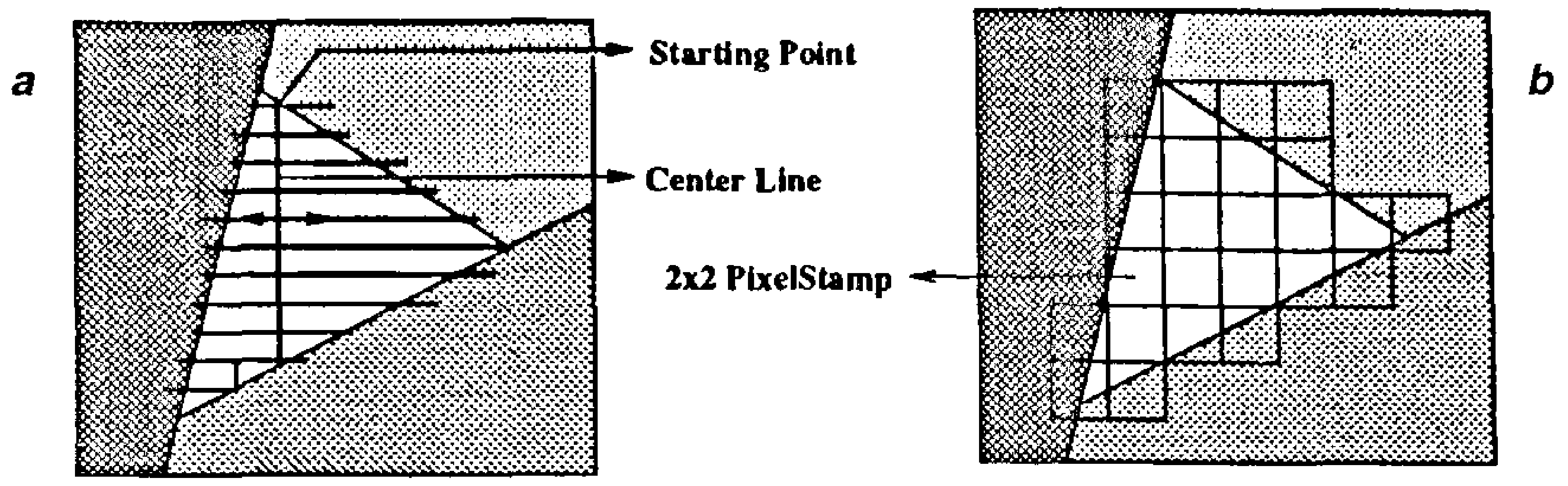

Figure 4. Scan conversion of a triangle using linear edge function.

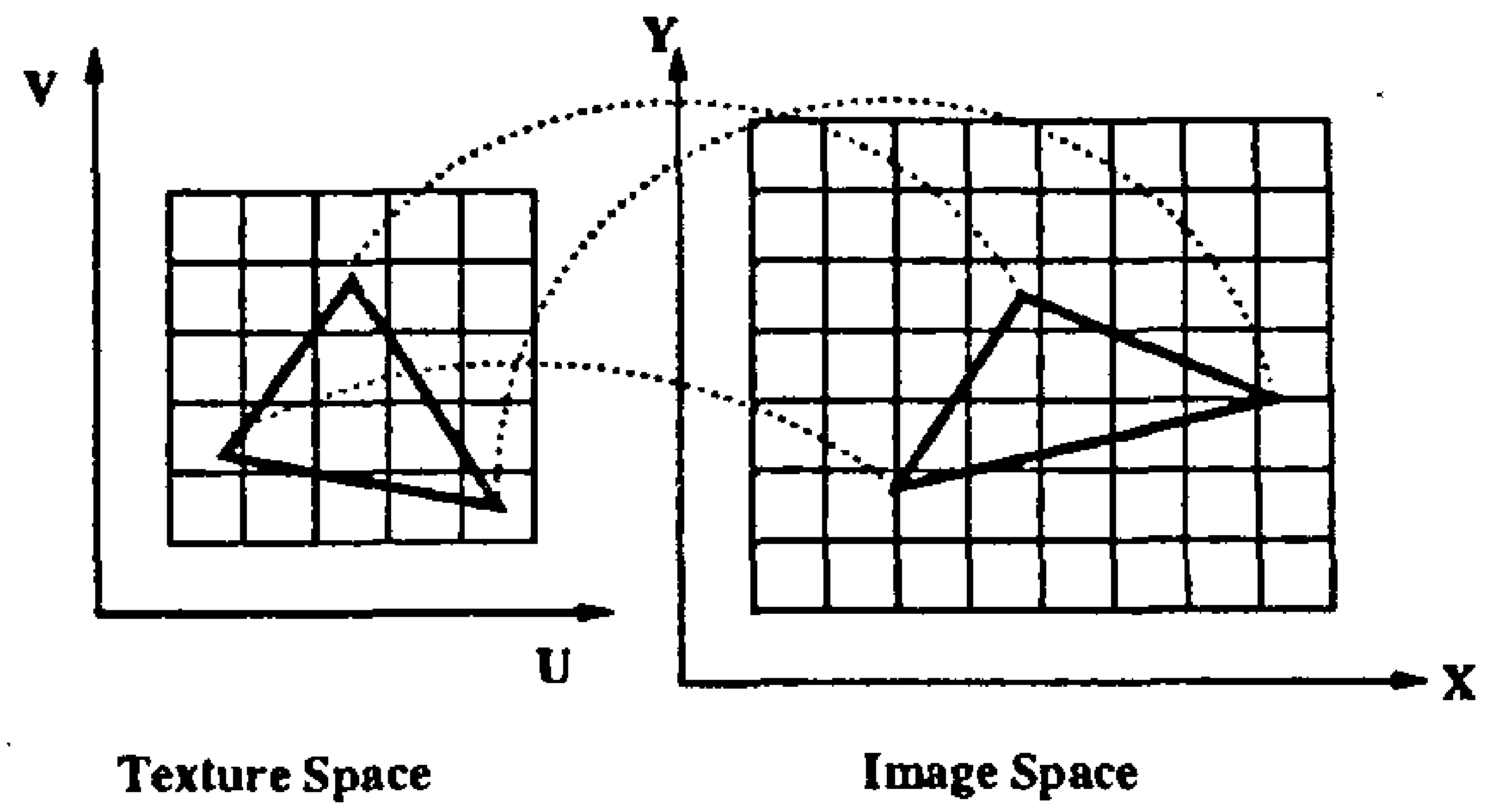

Figure 5. Texture mapping of a triangle. X, Y represent the coordinates of the triangle in image space. U, V represent the coordinates of the triangle in texture space and are known as the texture coordinates.

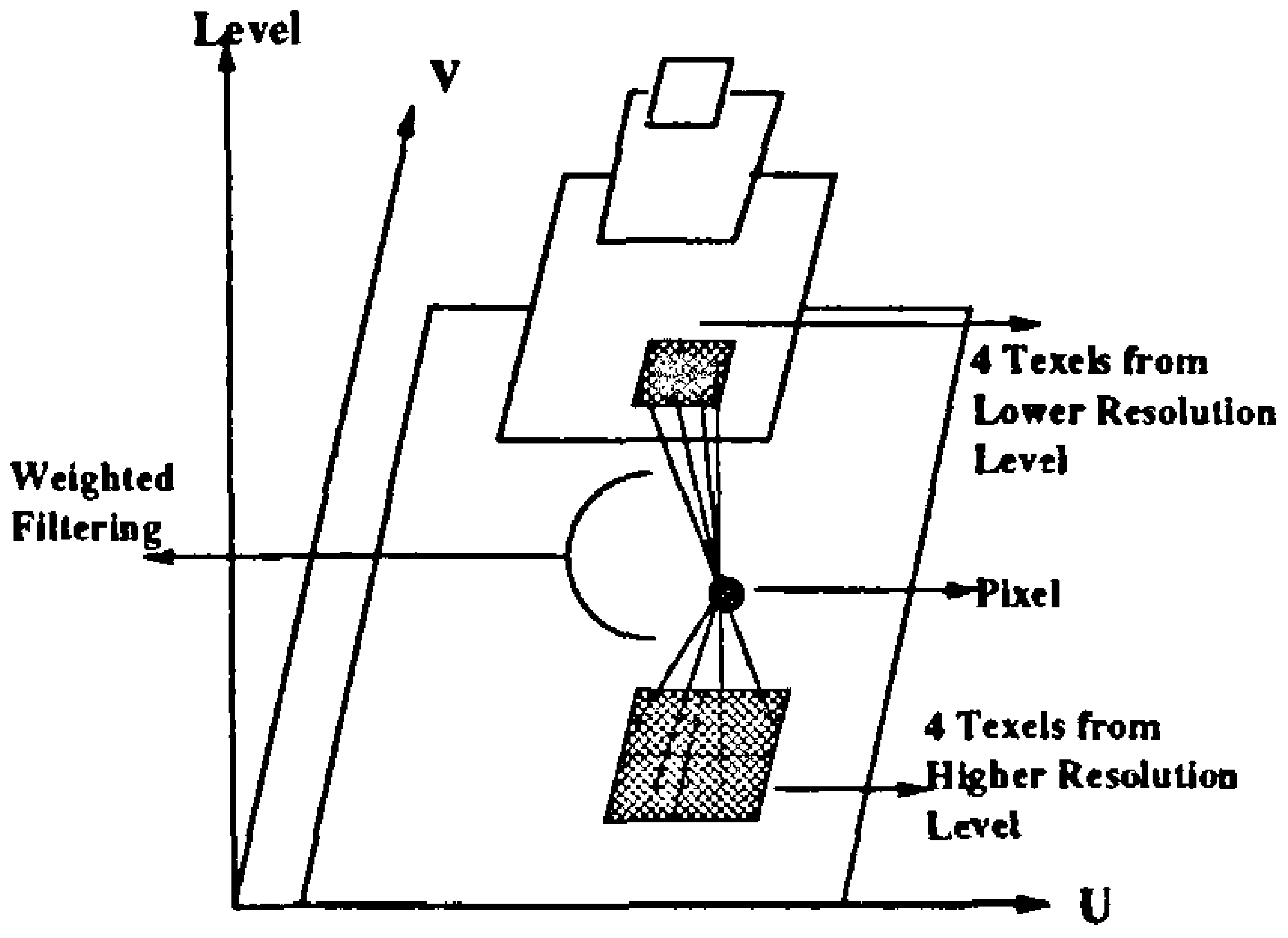
Level
V
4 Texels from Lower Resolution Level
Weighted Filtering
Pixel
4 Texels from Higher Resolution Level
U

Figure 6. Mip-map and tri-linear filtering. Each level in the mip-map represents a reduced resolution texture image from the previous mip-map. Tri-linear filtering performs weighted filtering of 8 texels, 4 from the lower resolution level and 4 from the higher resolution level.

which is an array with the same dimension as the frame buffer. After a triangle is scan-converted into a set of pixels, each pixel goes through the depth test. This test compares the depth value of the current pixel against the depth value of the pixel at the corresponding X–Y coordinate of the frame buffer. If the new value is smaller, the current pixel is closer to the viewpoint than the old pixel and therefore the depth and colour value of the current pixel replace the old values. Otherwise, the new pixel values are discarded. For transparent objects, the colour of old and new pixels is composited according to their transparency, or alpha value. This composition is known as **alpha blending** and requires another buffer for storing the alpha value called the **alpha buffer**.

The rasterization stage is quite compute and memory intensive. Let us consider a frame buffer with resolution 1280 × 1024 and average depth complexity for a scene of about 3. Assuming 32-bit pixel and 30 frames/sec, the frame buffer read bandwidth requirement will be 1280 × 1024 × 3 × 30 × 4 = 472 MB/sec. Similarly the rendering engine would require Z-buffer read bandwidth of 472 MB/sec assuming 32-bit Z-buffer. If a pixel passes the depth test, both the colour and depth information have to be written back. Assuming 50% of the pixels pass depth test, the bandwidth requirements are 236 MB/sec for frame buffer write and 225 MB/sec for Z-buffer write. Finally, each pixel requires 8 texels to perform tri-linear filtering. Even assuming an aggressive texture cache that stores the recently accessed texels, around 2.5 texel access per pixel are required, which translates to 1280 × 1024 × 3 × 30 × 2.5 × 2 = 590 MB/sec of texture memory bandwidth (assuming each texel is 16-bit).

Figures 7 and 8 show the triangle and pixel processing requirement per frame for Viewperf benchmarks[8], and Figure 9 shows the texture bandwidth requirement for some sample applications. A more detailed compute and bandwidth requirement of the rendering stages for different real-world applications can be found in refs 9–11.

Theoretically, the entire 3D graphics pipeline can be implemented in software. The geometry transformation stage is extremely floating point intensive, which was beyond the capability of general purpose processors even a few years ago. Today however, with processors having peak performance of around 400 MFlops/sec, the host CPU is capable of handling the load. The pixel-related rasterization operations, on the other hand, require tremendous memory bandwidth to process around 100 million pixels/sec. It is imperative that a separate hardware accelerator be dedicated to rasterization. Hence, two distinct classes of graphics architectures have been implemented: (1) combined geometry processor and rasterizer, the prime examples being RealityEngine[12] and InfiniteReality[13] from SGI; (2) host CPU-based geometry processing and dedicated hardware accelerator for rasterization. Almost all of today's low-end 3D graphics accelerators belong to the second class. In this case, the transformed geometry (vertex position, colour, and texture coordinates), as well as the texture images are transferred over a high-speed system bus such as PCI (Peripheral Components Interface) to the rasterization hardware accelerator. A major design issue for rasterization-only graphics accelerators is how to use the system bus bandwidth efficiently.

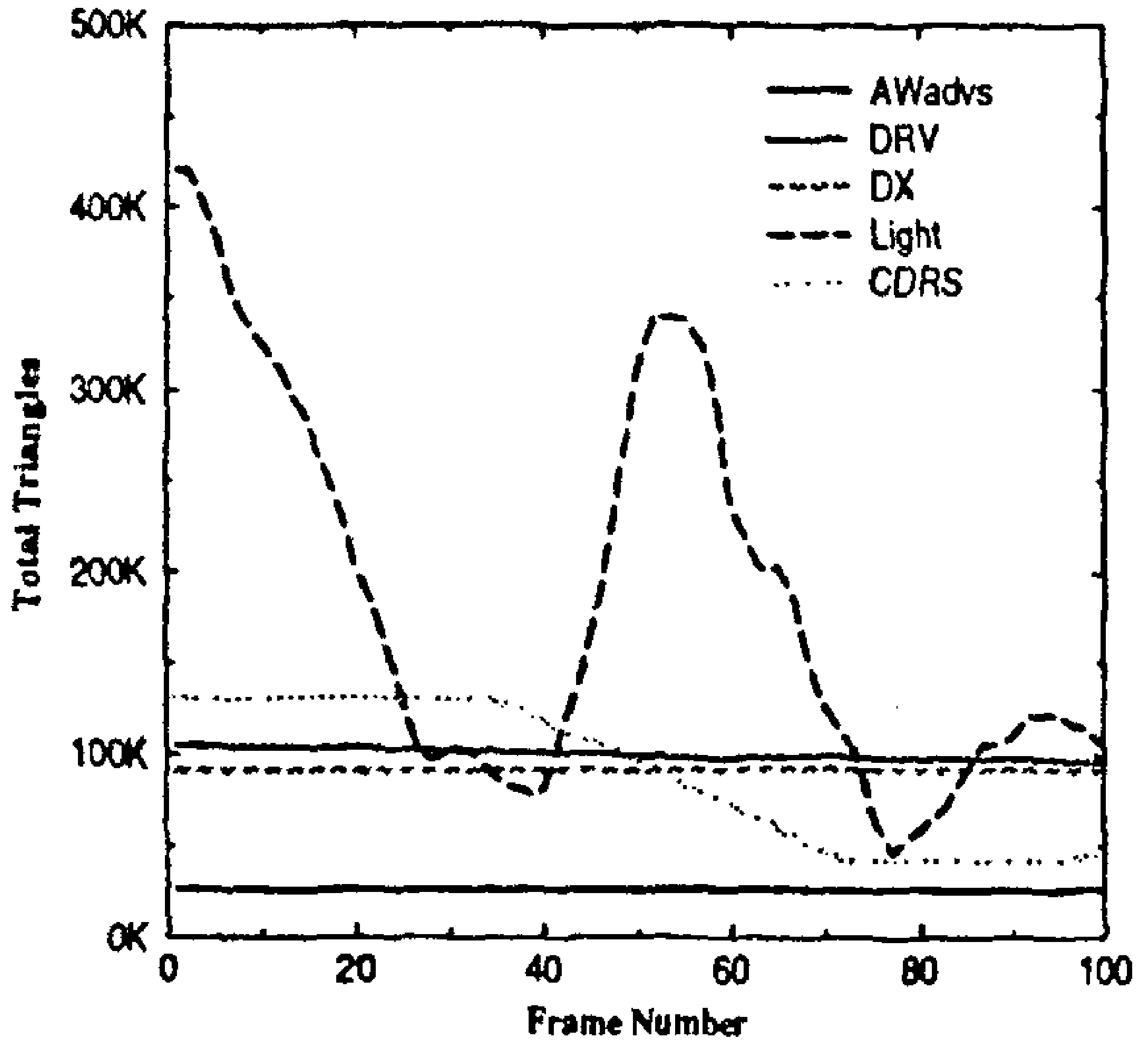

Figure 7. Total number of triangles processed by the rasterization engine at different frames or viewangles for various 3D applications.

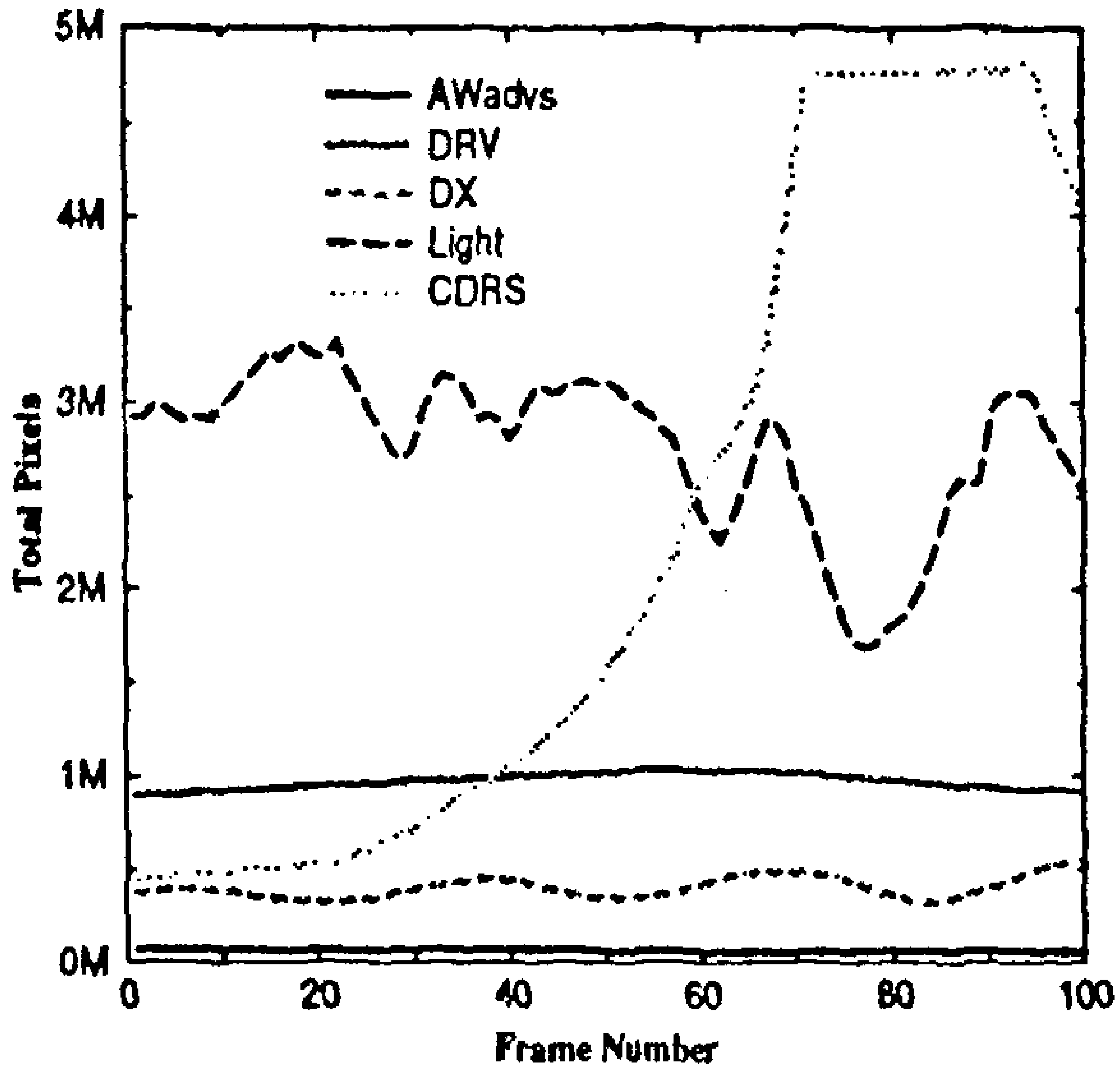

Figure 8. Total number of pixels processed by the rasterization engine at different frames or viewangles for various 3D applications.

3. Architectural innovations

To scale up the performance of the generic 3D graphics architecture described in the previous section, the following architectural issues need to be resolved:

- Although in theory state-of-the-art processors seem to have sufficient raw floating-point computation power to support geometric transformation at interactive frame rates, in practice the CPUs are lagging behind the rasterization performance of the 3D graphics cards. Therefore higher floating-point performance is essential to achieve faster frame rates with better rendering quality.
- The data transfer bandwidth between the CPU, which performs geometric transformation, and the 3D graphics card, which performs rasterization, plays a crucial role in the extent to which the entire 3D graphics pipeline can be sped up. The heavy use of texture map in modern 3D applications further exacerbates the bandwidth problem.
- The memory access performance in the scan conversion process has a dominating impact on the overall rasterization performance. Improving the rasterization algorithm's data access locality is pivotal to the graphics card's performance.

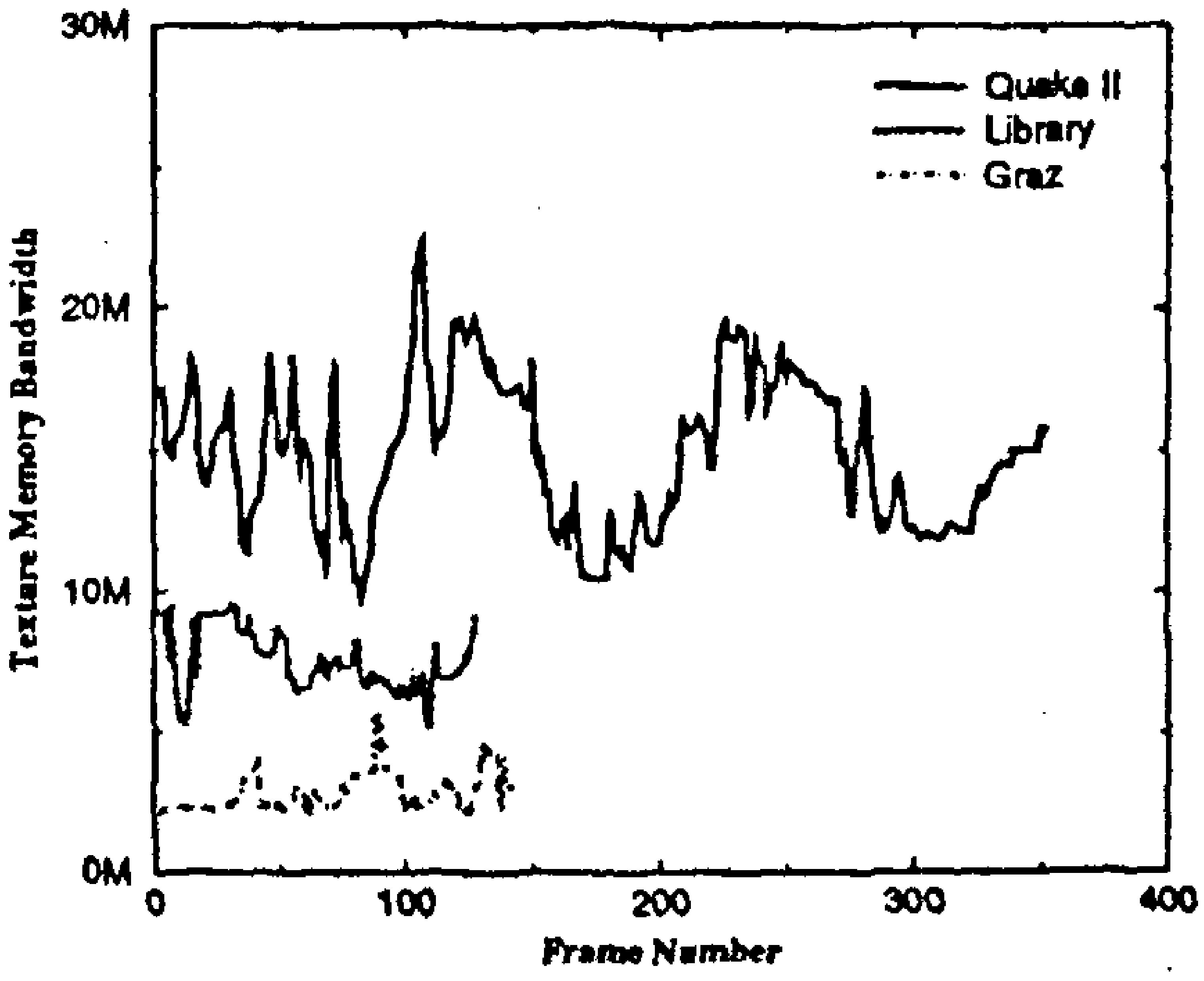

Figure 9. Total texture memory bandwidth in MBytes for different frames.

The following subsections describe architectural techniques that have been proposed and implemented to address these issues.

3.1 *Streaming SIMD extensions to instruction set*

Many current microprocessors have added Single Instruction Multiple Data (SIMD) type instructions to accelerate integer processing for media applications such as audio, video and image processing. This includes Intel's Multimedia Extensions (MMX), HP's Multimedia Architectural Extensions (MAX-2), Sun Microsystem's Visual Instruction Set (VIS), etc. However, the geometry processing stage of the 3D graphics pipeline is based on floating point data types. To exploit the parallelism in the geometry processing stage, Intel, AMD and others have recently added floating point SIMD instructions[14,15] to the instruction set. The main idea behind these extensions is that the geometry processing requires 32-bit floating point data types, whereas the floating point paths (registers and ALUs) are 64-bit in width in most modern processors. Because vertex processing is inherently parallelizable, SIMD instructions allow two vertex-processing operations to be performed simultaneously using a single floating-point instruction, with each vertex using half of the 64-bit data path. Yang, Sano and Lebeck[16] showed that SIMD instructions can improve the geometry transformation performance by 20 to 30%.

3.2 *Accelerated graphics port*

Figure 10 shows a high-level view of the components of a PC desktop system. It consists of the processor, main memory, the north bridge, PCI-based devices and various interconnecting buses. The north bridge has the memory controller and provides connections among different system components. The main processor fetches the 3D model from main memory, performs geometry transformation, and writes it back to the main memory. The graphics accelerator sitting on the PCI bus uses DMA (Direct Memory Access) to retrieve that data from the main memory and then performs rasterization. One major bottleneck of PC-based systems is the transfer bandwidth over the PCI bus, which connects the system memory to the local memory of the graphics accelerator[1]. The CPU needs to transfer geometry data, graphics commands as well as texture data to the graphics accelerator. Typically, the geometry information associated with a vertex is about 32 bytes[1], including the vertex coordinates, colour, and texture coordinates, i.e. 32 MB/sec for 1 million vertices. This information crosses the processor bus two time (once for reading and once for writing in the geometry transformation stage), the PCI bus once (transferring data to the graphics card), and the memory bus three times (in all the above cases). In addition, a large amount of texture data need to be transferred over the PCI bus as well. The peak PCI bandwidth of 32-bit, 33-MHz PCI bus is 132 MB/sec, which is still not quite sufficient. To solve this problem, Intel introduced a new bus specification, called Accelerated Graphics Port (AGP)[17]. AGP connects the graphics accelerator exclusively to the main memory subsystem (refer to Figure 11). AGP has four main advantages over PCI:

1. Reduction of load on PCI: The primary advantage of AGP is that it eliminates the graphics-related bandwidth requirement from the PCI bus by transferring data from the main memory to the graphics card over a dedicated bus.
2. Higher peak bandwidth: AGP 2X (32 bit data path at 66 MHz) transfers data on both edges of the clock, thereby achieving a peak bandwidth of 528 MB/sec. AGP 4X has a bandwidth of 1GB/sec.
3. Higher sustainable bandwidth: AGP supports *pipelining* of requests, i.e. overlapping of access time of request n with the issue of requests $n + 1$, $n + 2$ and so on. It also does *sidebanding* which provides extra address lines to issue new requests while the main data/address lines are transferring the data corresponding to previous requests. These two features makes it more likely for AGP to achieve a *sustained* bandwidth that is much closer to its peak bandwidth.
4. Direct memory execute: The amount of local memory present in the graphics accelerator is limited. However, to obtain more realistic images, applications use more and more high resolution textures, all of which cannot fit into the local memory. Hence, the graphics driver needs to perform texture memory management that keeps track of

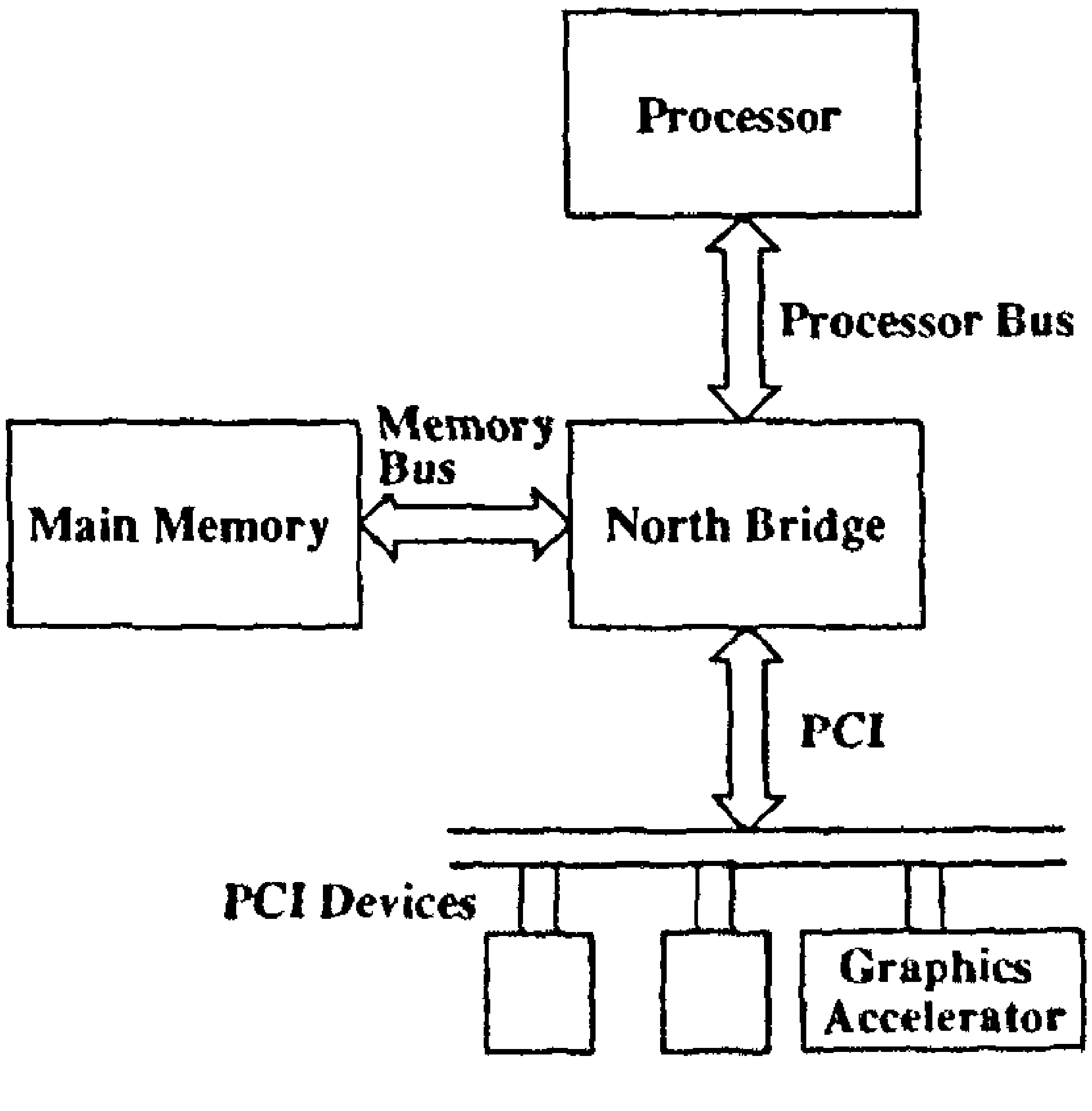

Figure 10. High-level view of the components of a PCI-based graphics subsystem.

the textures present in the local memory and downloads the required textures before they are used. This can introduce significant latency as the rendering engine waits for the complete mip-map of the texture image to be downloaded over the PCI/AGP bus. AGP provided a new feature called direct memory execute (DIME) that allows the graphics accelerator to directly address main system memory over the AGP bus. A translation table in the AGP controller, similar to the virtual to physical address translation table in the CPU, allows non-consecutive memory pages to appear as a single contiguous address space to the accelerator. This way the graphics accelerator can cache the heavily used textures in the local memory, and access the comparatively little used ones directly from the system memory.

3.3 *Bucket rendering*

Traditional rendering requires random access to the entire frame buffer, and it is not very cost-effective to provide a large high-bandwidth frame buffer. An interesting architectural idea that addresses this problem is bucket rendering. Bucket rendering is a technique where the screen-space is partitioned into tiles (also called chunks), and all the primitives of the scene are sorted into buckets, where each bucket contains the primitives that intersect with the corresponding tile. This architecture renders the scene one tile/bucket at a time, thereby reusing the Z-buffer, alpha-buffer as well as other necessary buffers for storing the results of intermediate rendering. At the end, all the tiles are collected together to form the final image.

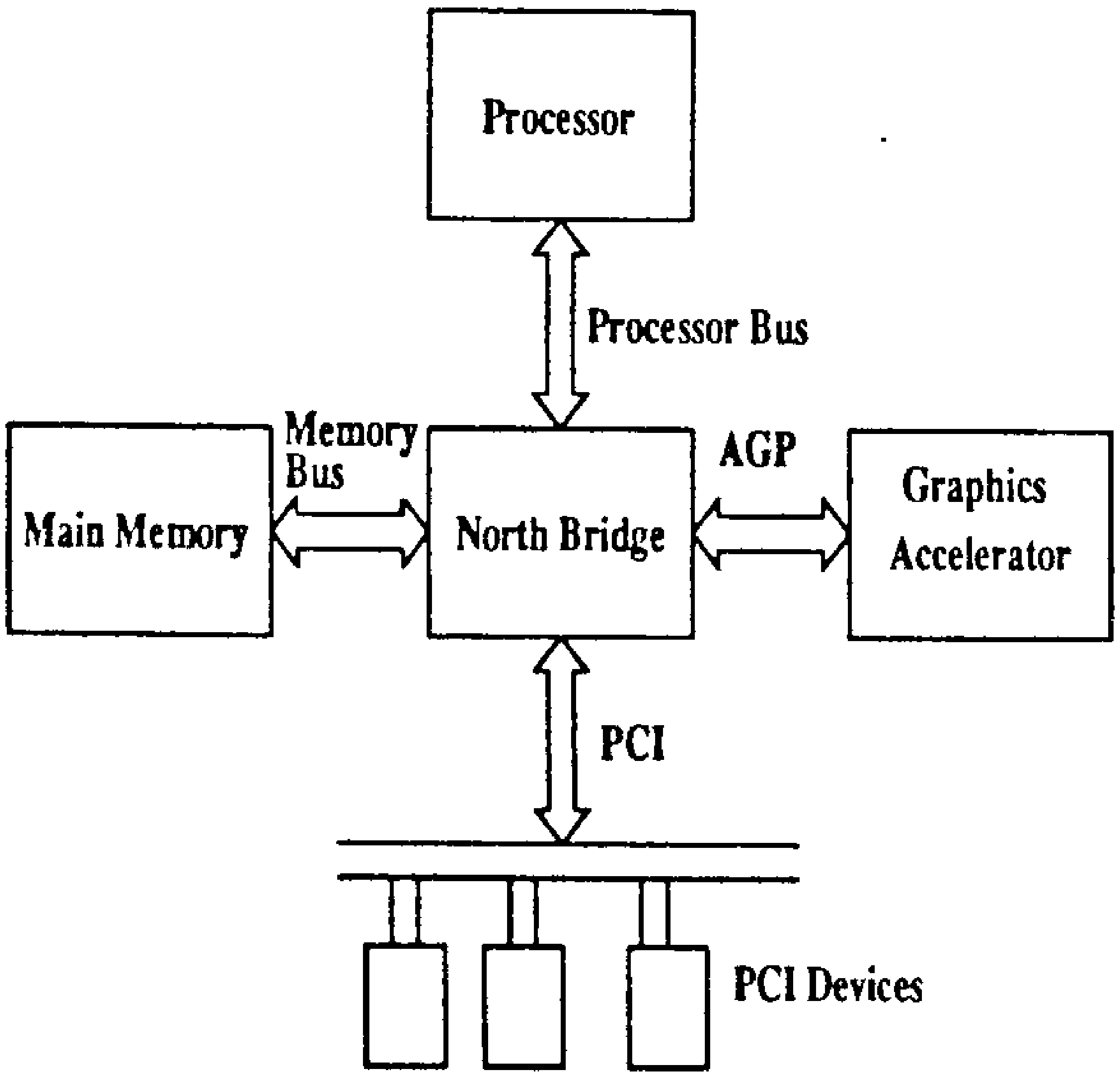

Figure 11. High-level view of the components of an AGP-based graphics subsystem.

Bucket rendering has been implemented in Pixel-Planes 5 (ref. 18), PixelFlow[19], Talisman[20] and finally commercially available PowerVR from NEC/VideoLogic. The main advantages of bucket rendering are the following:

- Since only one tile worth of rasterization buffer is required as opposed to a full-screen buffer, it is possible to use more bits per buffer entry to support more advanced rendering techniques such as oversampling or anti-aliasing, which rasterizes each pixel at a higher resolution and then down-samples the result to the required resolution.
- Titled architecture matches very well with the emerging embedded DRAM process that can provide small on-chip memory and high memory access bandwidth.

The main disadvantages of this architecture are

1. It requires an additional pipeline stage to sort triangles into buckets, thus increasing the total rendering latency.
2. Redundant work is performed because large primitives may overlap with multiple tiles.

3.4 *Composited image layers: Talisman*

Microsoft introduced Talisman architecture in 1996, that comprised several independent ideas. However, the key distinguishing feature of Talisman is composited image layer rendering[20] that exploits the frame-to-frame coherence for the first time. In traditional architecture, all the primitives are rendered in each frame even though there is a great deal of coherence between consecutive frames. Instead, Talisman renders each primitive on a separate image surface. All the image surfaces are then composited together to form the final image. In the next frame, the image for a primitive is transformed in the screen-space, given the transformation matrices in the object-space. If the error introduced by image-space transformation is below a threshold, the transformed image can be used as the final result of rendering. This architecture relies on the fact that image-space transformation is much less expensive compared to object-space transformation, and image layer composition can be performed more efficiently. The main disadvantage of this architecture is the complexity and gate count, and the incompatibility problem with traditional APIs like OpenGL. As a result, no commercial attempt has been made so far to implement Talisman architecture.

4. Parallel architecture

The 3D graphics pipeline is computation intensive, but is quite amenable to parallel implementation both in the object space as well as in the image space. Exploiting the graphics pipeline's parallelism can significantly reduce the total polygon rendering time. A considerable amount of research effort has been invested so far to design and implement various efficient parallel polygon rendering engines. In this section, we briefly describe different classes of parallelization techniques. Because the fundamental issue in 3D rendering is sorting the geometric primitives with respect to a given viewpoint, the parallelization strategies for polygon rendering can be classified as **sort-first**, **sort-middle** and **sort-last** depending on where the sorting operation is performed[19], which are illustrated in Figure 12.

In the sort-first strategy[21], the image space is partitioned into regions and each processor is responsible for all the rendering calculations (both geometry and rasterization) in the region to which it is assigned. The screen space bounding box of each 3D primitive is calculated by performing just enough transformations. Every 3D primitive is then distributed to the processors that are responsible for the image regions with which the bounding box overlaps. One primitive can be sent to multiple processors. From this point on, the set of 3D primitives in each processor goes through geometric transformation and rasterization completely independent of primitives in other processors. Finally, the image regions from the processors are simply combined together to form the final rendered image. The sort-first architecture has received the least attention so far because of the load imbalance problem in transformation and rasterization stage. However, as Mueller[21] pointed out, the sort-first architecture can easily exploit the frame-to-frame coherence and he has proposed a new adaptive algorithm to achieve better load balancing.

In the sort-middle strategy[22], the image space is again partitioned and each processor is responsible for one image region. 3D primitives are first transformed and then distributed to different processors based on the transformed X and Y coordinates of the primitives. Again a primitive is sent to multiple processors if it crosses the image region boundaries. After distribution, each processor performs rasterization on the transformed primitives independent of one another to produce a sub-image for the associated image region. The sub-images are then combined to form the final projection image. Sort-middle seems to be the most natural architecture and has been implemented both in hardware and software. Both InfiniteReality[13] and RealityEngine Graphics[12] have been implemented using sort-middle strategy. The main disadvantages of the sort-middle architecture are the load imbalance in the rasterization stage and the communication cost due to redistribution of primitives after transformation. Crockett and Orloff[22] proposed a static scan-line-based scheme for image space partitioning. Whitman[23] suggested adaptive load-balancing schemes for sort-middle architecture, while Ellsworth[24] took advantage of frame-to-frame coherence to achieve better load-balancing.

The sort-last strategy partitions the 3D input model in the beginning of the rendering pipeline without taking into account the viewpoint or object coordinates, performs geometric transformation and rasterization on each partition independently to produce a partial image, and finally composites the partial images according to the depth value of each image pixel. Because of its simplicity, the sort-last approach has been implemented in several systems, including PixelFlow[19] from University of North Carolina, which uses a high-speed combining network to composite sub-images. The performance of the sort-last strategy depends critically on the composition stage. Various methods have been proposed so far to perform the composition. The simplest method is to send the sub-images to a single compositing processor[19]. Other schemes proposed are binary tree composition[25], binary-swap composition[26,27] and parallel pipeline composition[28]. Mitra and Chiueh[29] showed that all previously proposed sub-image compositing methods can be unified in a single framework.

In general, in sort-last, a processor sends all the pixels of the relevant image space to another processor. This is known as **sort-last-full** technique[30]. Cox and Hanrahan[31] pointed out that it is sufficient to send only the 'active' pixels of the image space which is termed as **sort-last-sparse**. The trade-off between the two methods is the communication overhead versus extra processing required to encode the 'active' pixels.

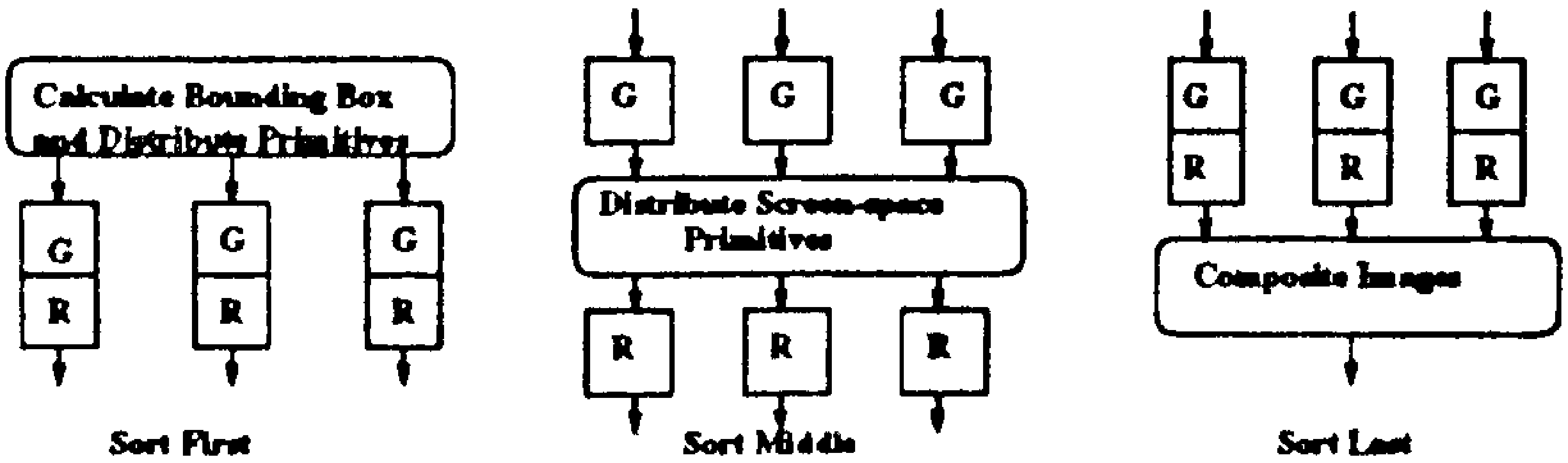
Calculate Bounding Box and Distribute Primitives
G
R
G
R
G
R
Sort First
G
G
G
Distribute Screen-space Primitives
R
R
R
Sort Middle
G
R
G
R
G
R
Composite Images
Sort Last

Figure 12. Sort-first, sort-middle, and sort-last parallel rendering architectures. The main difference between the architectures is where the distribution/sorting of primitives take place. G represents the geometric transformation engine and R represents the rasterization engine.

Until recently, all the parallel rendering engines were implemented either as dedicated ASIC, such as RealityEngine and InfiniteReality, or were implemented on massively parallel message passing or distributed shared memory machines such as Intel Paragon. Currently however, advances in the processor and graphics accelerator technology, as well as the emergence of gigabit local network technology, such as Myrinet, have made it possible to implement high performance 3D graphics engines on a cluster of workstations each of which is equipped with a low-cost 3D graphics card[32,33]. The basic parallelization strategies will remain the same for these architectures. However, the loosely coupled network topology may require different kinds of load balancing and composition algorithms.

5. Conclusion

A unique characteristic of 3D graphic applications is that there is no end to the addition of new features to the standard graphics pipeline. Unlike microprocessors, 3D graphics requires both advances in performance, i.e. more triangles and more pixels per second as well as new and improved techniques that deliver more realistic image and cinematic effects. Engineering and scientific 3D applications such as Computer Aided Design (CAD) and Computational Fluid Dynamics (CFD) applications as well as entertainment applications such as computer games and animated movies, all require higher-quality rendered images at a faster rate, thus placing an increasing demand on the triangle and pixel rate. Therefore, we expect that 3D graphics architecture will remain a challenging field in the foreseeable future and thus has abundant room for further algorithmic and architectural innovation.

1. Kirk, D., in Proc. of 13th ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, http://www.merl.com/hwws98/presentations/ kirk/index.html, Keynote address, 1998, pp. 11–13.
2. Neider, J., Davis, T. and Woo, M., *Open GL Programming Guide*, Addison-Wesley, 1993.
3. Microsoft Corporation, http://www.microsoft.com/directx/overview/d3d/default.asp, 1996.
4. Foley, J. D., vanDam, A., Feiner, S. K., Hughes, J. F. and Phillips, R. L., *Computer Graphics: Principles and Practice*, Addison-Wesley 1990, 2nd edn.
5. Fuchs, H., *et al.*, in Proc. of the 12th Annual ACM Conference on Computer Graphics (SIGGRAPH), 1985.
6. Pineda, J., in Proc. of the 15th Annual ACM Conference on Computer Graphics (SIGGRAPH), 1988, pp. 17–20.
7. Williams, L., in Proc. of the 10th Annual ACM Conference on Computer Graphics (SIGGRAPH), 1983.
8. OpenGL Performance Characterization Group, http://www.spec.org/gpc/opc.static/opcview.htm.
9. Dunwoody, J. C. and Linton, M. A., in Proc. of the ACM Symposium on Interactive 3D Graphics, 1990, pp. 155–163.
10. Chiueh, T. and Lin, W., in Proc. of the 12th ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1997, pp. 17–24.
11. Mitra, T. and Chiueh, T., in Proc. of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO), 1999, pp. 62–71.
12. Akeley, K., in Proc. of the 20th ACM Annual Conference on Computer Graphics (SIGGRAPH), 1993, pp. 109–116.
13. Montrym, J. S., Baum, D. R., Dignam, D. L. and Migdal, C. J., in Proc. of the 24th Annual ACM Conference on Computer Graphics (SIGGRAPH), 1997, pp. 293–302.
14. Intel Corporation, http://developer.intel.com/design/PentiumIII/manuals/, 1999.
15. Advanced Micro Devices, Inc., http://www.amd.com/products/cpg/3dnow/inside.html.
16. Yang, C., Sano, B. and Lebeck, A. R., in Proc. of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998, pp. 14–24.
17. Intel Corporation, http://www.intel.com/technology/agp/agp_index.htm, 1998.
18. Fuchs, H., *et al.* in Proc. of the 16th Annual ACM Conference on Computer Graphics (SIGGRAPH), 1989, pp. 79–88.
19. Molnar, S., Eyles, J. and Poulton, J., in Proc. of the 19th Annual ACM Conference on Computer Graphics (SIGGRAPH), 1992, pp. 231–240.
20. Torborg, J. and Kajiya, J. T., in Proc. of the 23rd Annual ACM Conference on Computer Graphics (SIGGRAPH), 1996, pp. 353–363.
21. Mueller, C., in Proc of the ACM Symposium on Interactive 3D Graphics, 1995, pp. 75–84.
22. Crockett, T. W. and Orloff, T., *IEEE Parallel Distributed Tech: Sys. Appl.*, 1994, **2**, 17–28.
23. Whitman, S., *IEEE Comput. Graphics Appl.*, 1994, **14**, 41–48.
24. Ellsworth, D., *IEEE Comput. Graphics Appl.*, 1994, **14**, 33–40.
25. Shaw, C., Green, M. and Schaeffer, J., *Advance in Computer Graphics Hardware, III*, 1991.
26. Ma, K., Painter, J. S., Hansen, C. D. and Krogh, M. F., *IEEE Comput. Graphics Appl.*, 1994, **14**, 59–68.
27. Karia, R. J., in Proc. of IEEE Scalable High Performance Computing Conference, 1994, pp. 252–258.
28. Lee, T., Raghavendra, C. S. and Nicholas, J. B., *IEEE Trans. Vis. Comput. Graphics*, 1996, **2**, 202–217.
29. Mitra, T. and Chiueh, T., in Proc. of the 6th IEEE International Conference on Parallel and Distributed System, 1998.
30. Molnar, S., Cox, M., Ellsworth, D. and Fuchs, H., *IEEE Comput. Graphics Appl.*, 1994, **14**, 23–32.
31. Cox, M. and Hanrahan, P., *IEEE Parallel Distributed Technology: Syst. Appl.*, 1994, **2**.
32. Samanta, R. and others, in Proc. of the 14th ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1999, pp. 107–116.
33. Experimental Computer Systems Lab, Department of Computer Science, SUNY at Stony Brook, http://www.ecsl.cs.sunysb.edu/sunder.html.