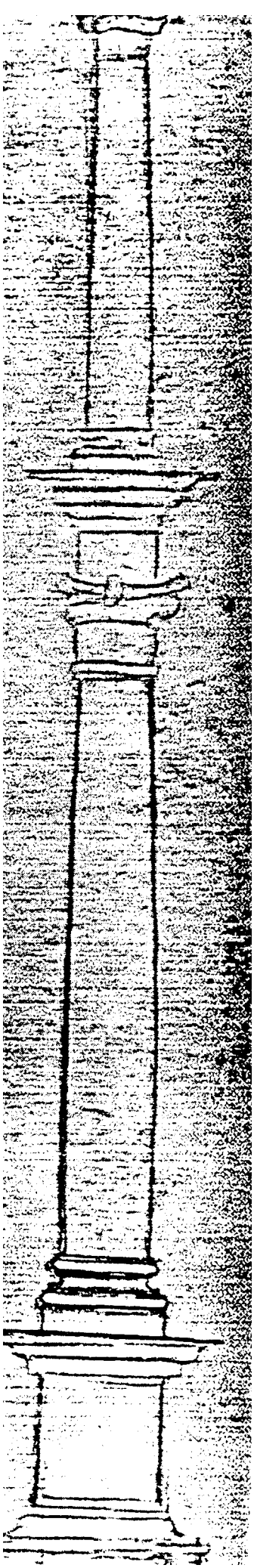

Denali Technical Overview

VERSION 1.0

MARCH 1993

KUBOTA PACIFIC COMPUTER INC.

Kubota

TABLE OF CONTENTS

1. INTRODUCTION

Denali is a high-performance graphics and imaging system with excellent cost/performance characteristics, which connects to a host computer through a highbandwidth data channel. Denali is very well suited for a range of demanding workstation application areas in 3D graphics and imaging, including mechanical CAD/CAE, imaging for medical, strategic, geophysical and remote sensing applications, molecular modeling, simulation, and engineering and scientific data visualization. Denali's advanced architecture is unique in combining hardware assistance both for graphics rendering and for image processing, making Denali a powerful solution for applications that require either or both graphics and imaging operations. In graphics rendering, Denali is capable of drawing more than a million lit, smooth-shaded, depth-buffered, 3D triangles per second. In volume rendering, Denali can perform twelve million trilinear interpolations per second.

The Denali architecture is based on a pipeline with very wide data paths and scalable parallelism at two different pipeline stages. This double scalability allows for tuning the configuration and balancing the pipeline according to the nature of the user's graphics and image processing requirements. Denali's implementation uses eight types of proprietary application-specific integrated circuits (ASICs) as well as a standard, off-the-shelf, high-performance floating-point engine and standard video output circuits.

The graphics rendering functions that Denali performs in hardware and firmware include geometry transformations, lighting calculations, Gouraud shading and depth cueing interpolations, depth buffer occlusion method (also called "Z-buffer"), anti-aliasing of lines and polygon outlines, 2D and 3D texture mapping (with several choices of texture sampling and filtering methods), environment mapping, stenciling, transparent surface rendering (both one-pass and multi-pass methods), alpha compositing, and several methods of volume data rendering. (See Section 3 for definitions of these and other graphics terms.) Denali also offers hardware support for fundamental image manipulation operations, such as zooming and scaling, image rotation, warping, and contrast enhancement.

Denali has a double 1280xlO24x24-bit frame buffer for double-buffered screen refresh. Off-screen frame buffer memory can have up to 576 bit-planes for use as depth buffers, stencil buffers, texture storage, and double-precision accumulation buffers to support full-screen anti-aliasing and other multi-pass advanced rendering techniques.

Denali provides a hardware cursor and further hardware support for functionality needed for interactive graphics in a window environment, namely, window-specific control of double buffering, overlays, and color table selection. Denali's Zero Time

Clear feature enhances double-buffered animation by eliminating the time cost for clearing the color and depth buffers for each new frame.

Initially, Denali is available with interfaces for two workstation families: Kubota Pacific's Kenai series and Digital Equipment Corporation's AXP series. Kubota Pacific supplies the low-level graphics system software needed to use Denali with each of these host platforms, as well as the standard, host-independent application programming interface libraries that provide access to the supported graphics functionality.

This Technical Overview describes the Denali architecture and implementation, and reviews the meaning and import of the graphics and imaging functions that it provides. Section 2 presents an introductory overview of the architecture and packaging. Section 3 focuses on the graphics functionality. Section 4 presents more detailed descriptions of Denali's subsystems. Section 5 treats Denali's applicability to image processing and volume rendering. Section 6 discusses the software architecture.

2. DENALI SYSTEM OVERVIEW

This section gives a brief overview of the Denali system, its architecture and its packaging. The section on architecture refers to a number of technical terms of computer graphics, flagged by italics. Definitions of these terms and some discussion of their import are contained in Sections 3 and 4.

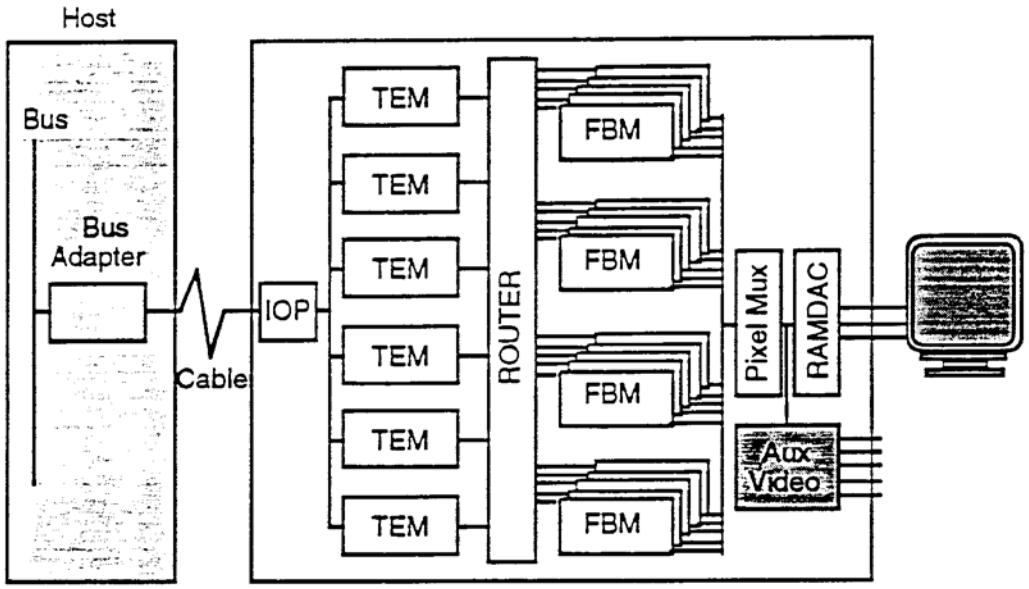

Figure 1 -- Denali Block Diagram

2.1. TOP-LEVEL ARCHITECTURE DESCRIPTION

Figure 1 is a block diagram showing Denali's organization as a pipeline with scalable parallel stages. Normally, in graphics rendering, data flows from left to right in this diagram. An important and notable characteristic of the system, (not explicitly shown in the diagram), is the use of very wide data paths and high bandwidth memory access throughout.

The first parallel stage comprises from one to six Transform Engine Modules (TEM). Each TEM is a daughter board plugging into the Denali mother board, containing a programmable Transform Engine (TE), several custom ASICs, and fast memory to support their functions. The TEMs implement the first stages of the *rendering pipeline*. They perform rendering computations that are applied on a per-primitive basis or a per-vertex basis, including the needed geometry transformations, and the *lighting* and *shading* computations. Finally, the TEMs perform *scan conversion*, or *rasterization* of the primitives. The TEMs typically work in parallel on different primitives making up a scene.

The output of the TEMs in rendering is a stream of pixel packets, each containing data pertaining to a specific screen pixel location. The contents of each pixel packet arise from sampling the primitive at a point corresponding to the pixel location, and typically involve interpolation of values specified at the primitive's vertices. The per-pixel data can include ***color, transparency, depth*** in the 3D scene of the corresponding point on the primitive (conventionally called Z), ***texture coordinates,*** and other parameters used in advanced rendering methods. Section 3.3 discusses the sampling process and the nature of the sample data in more detail.

Denali's second parallel stage comprises five, ten or twenty Frame Buffer Modules (FBM), which implement the last stages of the rendering pipeline. They perform the rendering computations that are applied on a per-pixel basis. They also contain the raster display memories, the off-screen raster memories (for ***double buffering, depth buffering, accumulation buffer,*** and ***texture*** storage), and per-pixel control memory used for control of window-specific ***double buffering, overlays,*** and the Zero Time Clear function. The FBMs work in parallel on different pixels.

Some of the per-pixel operations that the Frame Buffer Modules apply are comparison tests that control whether new pixel data is written to the output buffer, as in ***depth buffering*** and ***stenciling.*** Otherwise, per-pixel processing consists of accumulating the contributions of samples of possibly several different primitives that affect a pixel or the contributions of several samples of each primitive in multi-pass rendering methods. Such per-pixel computations arise in ***anti-aliasing,*** in ***texture mapping,*** in ***transparent*** surface rendering, and in several other advanced graphics methods. Section 3.3 expands on the FBM functionality in more detail and Section 4.4 gives more detail on its architecture.

The Router, located on the Denali motherboard, connects the TEM stage to the FBM stage. The stream of pixel packets that comes from the parallel TEMs has no particular ordering with respect to pixel location. On the other hand, the FBMs are pixel-location specific; each FBM contains a particular fixed (interleaved) set of pixel locations. The Router is a highly parallel cross-bar switch, which distributes the pixel packets coming from the TE:Ms to the correct FBMs. Section 4.3 gives more details on the Router.

At the beginning of the pipeline, the I/O Processor ASIC (IOP) receives a stream of graphics command packets from the host computer and distributes them among the TEMs. The Denali IOP communicates with a host-specific I/O processor, external to the Denali system. The host-specific IOP is typically a board which connects to the host's DMA channel (TURBOchannelTM for both the Titan II and AXP platforms). Section 4.1 discusses the IOP.

At the end of the pipeline, the pixel values that are stored in the FBM display buffers are used to refresh the CRT display. The Video Output Section provides the

display refresh functionality. It also includes the color look-up tables and overlay control. It is discussed in more detail in Section 4.5.

The system also has a Read Back mode, in which data flows in the opposite direction. Some multi-pass rendering methods, as well as image processing applications, use the Read Back mode to allow the TEs or the host to process data from the frame buffers.

2.2. PHYSICAL PACKAGING

Denali is implemented on a single large mother board measuring 14x16 inches, which has connectors to receive daughter boards carrying the two types of parallel subsystems--six connectors for Transform Engine Modules and twenty connectors for Frame Buffer Modules. Kubota Pacific offers several options for packaging of the Denali system. There is an independent enclosure including power supply, which can be placed on a desktop or mounted vertically with a supplied stand. Or, for some workstation models, the Denali board can be placed in the same enclosure as the host workstation system.

The rear panel has connectors for line power, the IOP connection, video output and stereo control signals. The Denali IOP connects to the host-specific IOP with a 50wire cable that can be up to 10 feet long. The mother board includes a connector to accommodate an auxiliary low-resolution video board, described in Section 4.5.

This section presents Denali's graphics rendering functionality. In the interest of clarity, some care is taken to review the meanings of the relevant graphical terms and concepts.

3.1. THE ROLE OF THE HOST

Rendering **is the process of producing a picture from a** *model.* **Modeling methods are usually specific to the application domain, but 3D models will all include some form of geometry specification describing 3D objects to be drawn on the 2D display surface. Graphics application programming interfaces (APIs) accept geometry specifications in terms of defined** *primitive* types. An **application must express its applicationspecific geometry in terms of the primitives defined by the API in use. Some APIs accept higher-level primitives, such as text characters, complex polygons with holes, or parametric curves and surfaces. Denali** and **most graphics hardware systems provide a shorter list of simpler primitives. The run-time support for an API must decompose the application's primitives into the Denali primitives before passing them on to the low-level controlling software, which in turn passes the graphics commands to Denali.**

The primitive types accepted by Denali are:

- **lists of points, defined by coordinates in a 3D model space, to be drawn as one or more pixels**
- **lists of markers, which are points to be drawn with defined marker symbols**
- **lists of line segments, each defined by a pair of end points, called vertices**
    - **polylines, which are chains of line segments joined end-to-end at common vertices**
- **filled rectangles**
- **circles and filled circles**
- **triangle lists and triangle strips**
- **quadrilateral lists and quadrilateral meshes**
- **sphere lists**

The polyline primitive and the several polygon-based primitives are defined by *vertices,* **and lists of** *edges,* **which are line segments that join adjacent vertices. An important benefit of including polylines, triangle strip and mesh, and quadrilateral**

mesh primitives in the Denali primitive set is that in these primitives each vertex is shared by two or more edges or polygons. Thus, Denali needs to perform some vertexspecific operations, such as the geometry transformations and lighting calculations, only once for each vertex of the primitive, rather than repeating them for each polygon to which the vertex belongs. (See Section 4.2)

In addition to primitives, applications specify *attributes* **that control how primitives are to be drawn. Attributes include line width, marker type, color, stipple and hatch patterns, surface reflectance properties, locations and properties of light sources, anti-aliasing control, geometry transformation parameters and view definition parameters. Most attributes are implemented as state variables of the rendering pipeline.**

The application software on the host (and the supporting graphics API) draws a picture by traversing an application object database and generating appropriate commands for the graphics subsystem. The graphics commands generated by the host include commands to draw primitives, to set attributes, and to otherwise control the graphics system's state. (With some APIs, it is the application program itself that performs the traversal of the object data, while for other APIs it is the API's runtime support software that performs the traversal of display lists or structures that the application has defined.)

Whatever the API and the nature of the application, the traversal of the object data base on the host generates a stream of Denali commands, which are passed via the host IOP to the Denali IOP, which distributes them to the Transform Engine Modules for interpretation by the Transform Engines (TE).

3.2. GEOMETRY PROCESSING AND THE TRANSFORM ENGINE

One of the necessary stages of rendering provided by the TEMs, and at the beginning of the rendering pipeline, is *geometry processing,* **which includes:**

- **the coordinate transformations that have to be applied to vertices in order to position the primitives with respect to each other, with respect to the light sources, and with respect to the eye position and viewing direction;**
- **the transformations needed to project the 3D vertices to the viewing plane and map them to integer screen coordinates;**
- **clipping, which is the elimination from the pipeline of the primitives and parts of primitives that lie outside of defined regions;**
- **lighting computations--specifically, the determination of the light reflected towards the viewer from the vertex position on the surface under specified illumination, according to attributed surface reflectance properties.**

The geometry processing, including lighting, is generally applied only to the vertices of the primitives. Clipping can result in redefining a primitive, eliminating some vertices and creating new ones, when a primitive crosses a clipping boundary. The coordinate transformations and viewing transformations reduce mostly to matrix multiplications, which are basically floating-point multiply-accumulate operations. Evaluation of the lighting function at each vertex also reduces mainly to floatingpoint multiply-accumulate operations. On the Denali, the Transform Engine processor provides all the geometry processing. Section 4.2 shows how this processor is especially well suited to the task.

3.3. SCAN CONVERSION AND THE LINEAR EVALUATOR ARRAY

After geometry processing, each vertex is represented by a pair of screen coordinates, an ROB color value, a depth value (Z), possibly a transparency coefficient, and a pair or triplet of texture coordinates in the case of 2D or 3D texture mapping, respectively. The next stage of the rendering pipeline consists of *scan conversion* or *rasterization,* **which determines for each primitive the screen pixel locations that it affects, and the corresponding values of the relevant variables for each affected pixel: color, depth, alpha, texture coordinates, and, for anti-aliased lines, the distance of the pixel from the line. In scan conversion, these per-pixel quantities may be determined for each pixel by interpolating from the vertices of the primitive to its interior points.** In **particular, in smooth** *shading* **the color of each pixel is determined by interpolating the vertex colors that are the result of the lighting model computation at the vertices, while in** *facet shading,* **every pixel in the primitive has the same color.**

For scan conversion in Denali, the primitives are decomposed into their simplest parts polylines are decomposed into vectors and polygon strip and mesh primitives are decomposed into triangles. Determining whether a given point lies within a triangle or a thick line, and determining the distance of the given point to a line or triangle edge, consist of evaluating a set of linear expressions that mathematically represent the edges of the primitive. Determining the values of the interpolated variables similarly reduces to evaluating some linear interpolation equations. In **Denali, the scan conversion is produced by the Linear Evaluator Array ASICs on the Transform Engine Modules, described in Section 4.2.**

The result of the scan conversion of each primitive is a stream of pixel packets, representing the pixels inside (or sometimes close to) the primitive. The packet for every pixel contains its pixel address and its color based on the object color attribute and the application of lighting and shading. Depending on the details of the rendering method in use, a pixel packet might also contain a 24-bit depth value, an 8or 16-bit alpha value, an eight-bit stencil mask, an eight-bit distance, and two or three texture coordinates, each of 24-bits. All these per-pixel data may be used later in the rendering pipeline for depth buffering, for transparency or other compositing operation, for stenciling, for texture mapping, or for anti-aliasing of vectors and polygon outlines. The Pixel Engine ASICs, which reside on the Frame Buffer

Modules, perform all of these downstream per-pixel operations. Section 4.4.3 describes the Pixel Engines in greater detail.

3.4. DEPTH BUFFER AND STENCIL

Depth buffering (also called *Z-buffering) is* the method most commonly used in workstation graphics to account for the effects of occlusion, which occurs in 3D graphics because opaque objects can obscure other objects (or parts of objects) from view. Depth buffering is also the most common instance of the requirement for downstream per-pixel processing, between the scan conversion stage and writing a pixel to the frame buffer. For depth buffering, the per-pixel processing consists of comparing depth values to control whether or not a pixel is written to the output buffer.

The depth buffer consists of some off-screen frame buffer memory, 24 bits/pixel in Denali. Before a rendering traversal begins, the depth buffer is initialized to a smallest possible value (which corresponds to the deepest positions in the 3D scene, i.e. the locations most distant from the viewer). Then, during traversal, as each pixel packet arrives from the scan conversion stage, its depth value is compared with the value in the depth buffer at the same pixel position, which is either the initial depth value or the depth value of a pixel in that location from some primitive encountered previously in the traversal. If the comparison shows that the new pixel comes from a point in the model scene that is nearer the viewer than the previous pixel, then the new pixel color value and its depth value are written to the output buffer and depth buffer respectively; otherwise the new pixel is discarded. Thus, we can be sure at the end of the traversal that each pixel will be colored according to the primitive that covers it and is nearest the viewer.

In Denali, depth buffer memory is allocated from DRAM memory on the Frame Buffer Modules. The per-pixel depth comparison is performed by the Pixel Engine ASIC on the FBM, discussed in Section 4.4.3. In addition, the PE includes a depth query feature that supports hierarchical application of depth testing, discussed further in Section 4.4.7.

Stenciling is another way of applying conditions to control whether pixels are written to the output buffer. The conditions that control the writing of pixels can be quite general, based on a defined eight-bits-per-pixel stencil mask in the off-screen frame buffer memory and an eight-bit stencil field contained in the pixel packet of every 3D pixel from the TEMs. The condition for writing a pixel can be based on a variety of logical and arithmetical combinations of the rendered pixel stencil value and the value for the same pixel location in the stencil buffer. In its simplest use, the stencil can provide a spatial mask of arbitrary shape, specified independently of the object data base.

3.5. TRANSPARENCY

The transparent surfaces simulated in graphics must reflect some light, else they would not be visible. However, transparent surfaces do not completely occlude the objects behind them. The transparency effect is simulated by computing the displayed pixel value as a blend of the color of the transparent surface and the color of the opaque surface behind it. Transparent surfaces are illuminated and scan-converted like any other surface to determine their pixel colors under the standard lighting model and texture application. However, each pixel coming from a transparent primitive will also contain a non-zero transparency coefficient in the pixel packet's alpha field. This alpha coefficient is used in blending the color value of the present pixel with color values contributed to the same pixel location by the primitives that lie behind the current primitive.

Denali supports three modes of rendering transparent surfaces. If the triangles can be sorted into a strict back-to-front ordering, then the scene with transparent surfaces can be rendered on a single pass, a single traversal of the object data base. In the two-pass mode, the objects are also sorted back to front as well possible, but the opaque objects are all drawn in a first pass and the transparent objects on the second pass. In the multi-pass method, the opaque geometry is drawn in the first pass, then transparent geometry is drawn on multiple passes, until the ordering of the geometry at each point is known. Of course, the per-pixel blending operations are all performed by the PEs on the FBMs. Multi-pass transparency requires the use of an accumulation buffer. (See Section 4.4.2)

3.6. ANTI-ALIASING OF LINES AND POLYGON OUTLINES

In raster graphics, *aliasing* occurs because of the finite pixel size, which limits the spatial frequency at which an image can be sampled. The visible manifestation of this sort of aliasing are "jaggies", the familiar staircase appearance of lines and polygon boundaries, which is evident even on raster displays of the highest resolution. Jaggies constitute a major limitation of raster image quality, and moreover, can be extremely distracting and annoying to the operator in some interactive applications.

***Anti-aliasing*, the cure for the jaggies, is a kind of filtering. It smoothes, softens, and blurs the appearances of the pixels on or near the aliased edge, to make the jagged stair steps less noticeable.** In **Denali, anti-aliasing of lines and polygon outlines makes use of the distance from a pixel sample point to the mathematical line or edge. It uses this pixel sample distance as an index into a table of filter coefficients, which are between zero and one and are used to multiply the interpolated pixel color before writing it to the output buffer or blending it with the current contents. The Linear Evaluator Array on the TEM computes the pixel sample distance in the scan conversion process. The Pixel Engine on the FBM performs the filter coefficient lookup and the scales the pixel's intensity by the filter coefficient. This kind of anti**

aliasing works better on lines that have some thickness. The **Denali hardware supports line widths of one, two, and four pixels; the antialiasing filter can be up to eight pixels wide.**

More generally, the problem of aliasing goes beyond jaggies on rendered lines. It is a general phenomenon of discrete sampling methods. Aliasing generally refers to artifacts that can arise in a sampled signal whenever the original signal has significant high-frequency periodic components, specifically, with frequencies higher than half the sampling frequency. These artifacts can appear as visually prominent lower-frequency periodic effects that are not at all present in the original signal or image. In particular, the textures commonly used for texture mapping in graphics (e.g. wood grain) frequently contain repetitive fine detail, and so are especially susceptible to such aliasing artifacts. The next section describes further antialiasing methods for textures that are supported by the Denali hardware.

3.7. TEXTURE MAPPING

Texture mapping is a method of adding fine-scale variations to surface color to simulate more realistic-appearing surfaces. It would be computationally intractable to produce such fine detail using the same geometric modeling methods as used for the gross surface shape. An application supplies *textures* **as pixmaps, which may come from scanning real images or from procedural generation. Denali also supports 3D textures, which may be considered as stacks of 2D textures, or time varying 2D textures.**

To map a texture to an area primitive, an application has to supply texture coordinates with each vertex of the primitive--a pair of texture coordinates for ordinary 2D textures and three texture coordinates for 3D textures. Texture coordinates are (possibly fractional) indices that select the row and column of the texture pixmap from which to fetch a color to apply to the point on the surface. The texture color for the point might replace the color defined by the other attributes and the lighting model, or it might modify that color as a multiplicative factor, or it might be blended with that color in some other way.

In the scan conversion stage of the pipeline, texture coordinates at vertices must be interpolated to give texture coordinates for each pixel. When the view is perspective (the usual case in realistic rendering), the interpolation of the texture coordinates must be non-linear, specifically rational, to obtain good results. That is, the perspective projection must be computed for each pixel; it does not suffice to compute the perspective projection only at the vertices and then linearly interpolate projected coordinates. Effectively, this requires dividing two or three linearly interpolated texture coordinates by a linearly interpolated common denominator at each pixel. On Denali, the LEAs perform the linear interpolations, and the PEs perform the division.

In general, as textures are mapped to the model surface by the texture mapping, and thence to the display screen through viewing transformations, the pixels of the texture are not put into one-to-one correspondence with the pixels on the screen. (For this reason, the elements of the texture are called texels, to avoid confounding them with the screen pixels). The rows and columns of the texture do not generally map to lines parallel with the rows and columns of the display raster. The texture map can stretch the texture, so that a typical texel spreads out over several screen pixels; or, the texture map can compress the texture, so that several texels map into a single screen pixel. Even the simplest planar texture map compresses the texture nonlinearly when projected in perspective. Generally, in a single texture map, the texture may be stretched in one direction and compressed in another direction, and the nature of the stretching and compressing can vary from point to point. Perspective projection induces a non-linear compression of the texture, which necessitates the rational interpolation of texture coordinates discussed above.

Therefore, displaying a texture-mapped surface requires resampling the given texture. Resampling a digital image always provides an opportunity for unwanted aliasing artifacts, especially when the image contains repetitive fine detail, as many interesting textures do. These artifacts can be especially distracting in animation, because textures can appear to crawl on moving surfaces in unnatural ways. Therefore, in order to achieve high quality texturing in rendered images, the graphics system must apply some sort of blending or filtering in the resampling of the texture pixmap. A fundamental problem of texture map filtering is that the appropriate filter kernel depends on the degree of compression of the texture when it is mapped to screen space, and, in general, this compression varies across the texture.

Denali offers several methods of texture sampling and filtering, which give a cost/quality tradeoff, and which are appropriate for a range of texture mapping applications:

- **point sampling -- for each screen pixel the sample is the value of the texel closest to the point in the texture plane given by the pixel's texture coordinates. Point sampling is the cheapest to compute, but is most subject to aliasing artifacts that limit image quality;**
- **bi-linear interpolation -- the texture sample color is a blend of the four texels around the sample point in the texture plane (the point defined by the texture coordinates); each texel's contribution to the blend is weighted inversely to its distance from the sample point, using a bi-linear function of the texture coordinate differences. (Tri-linear interpolation from the eight nearest volume texels is used in the case of 3D textures). This method is appropriate when the texture is stretched, so that a single texel covers a number of screen pixels.**
- **MIP mapping is a combined filtering and resampling method that is especially applicable when the texture is compressed, so that each screen pixel is influenced**

by a number of texels. A **MIP map is a compact structure for a storing a texture pre-filtered at multiple resolution levels: an original texture and pre-filtered versions of 1/2, 1/4, 1/8, ..., of the original resolution. Then, in evaluating the texture for each pixel, an appropriate resolution level is selected according to an estimate of the degree of texture compression at that pixel. Because of the powerof-two ratio for each resolution step, the texture coordinates for the resolution level can be economically obtained by logical shifting of the binary texture coordinates contained in the pixel packets. In the simplest version of MIPmapping, the pre-filtered sub-resolution texture is point sampled. Denali also supports refinements to MIP mapping that use bi-linear interpolation within the MIP map at each resolution level, as well as linear interpolation between the two MIP map resolution levels that bracket the exact compression ratio.**

On Denali, textures can be of many visual types: 32-bit RGBa, 64-bit RGBa, both with several different byte orderings. The fastest texture mapping obtains when the textures are stored locally in the FBMs, with a copy of the texture in each FBM. Local 2D textures can have size up to 512x512 and 3D textures up to 64x64x64. In practice, these are usually not serious limitations because the texture mapping objective is frequently well served by using the stored texture as a *tile* to be repeated periodically to fill a larger texture space. The PE's can evaluate texture coordinates modulo the tile size, to effect texture tiling automatically.

Larger textures can be stored non-locally, in host memory. Of course, application of non-local textures will be much slower because texture coordinates have to be read back to the host to fetch the texel values. Non-local textures can have size up to 4096x4096. This limitation is due to the precision of the LEA interpolators.

3.8. ENVIRONMENT MAPPING

Environment mapping is a method of using the mechanics of texture mapping to simulate specular reflection effects on a rendered surface. It is not geometrically accurate, but it is an inexpensive way to produce a variety of shiny-surface effects.

The application begins by generating a texture that represents the projection of the environment on a cube surrounding the object. In **the Denali hardware, the geometry processing in the TEs assigns to each vertex a reflection vector that depends on the viewing direction and the surface orientation at the vertex. Then the LEA's interpolate the reflection vector components to each pixel. The pixel processing stage in the PEs then uses the interpolated reflection vector components as indices into the environment texture.**

3.9. SHADOWS

Denali can render shadows in two or more passes--a first pass that ignores shadowing, and an additional pass for each light source, to account for the effects of shadowing of

that light source. On each light source pass, the Transform Engine performs the geometry calculation to determine a set of *shadow quads* for each light source. A shadow quad is a quadrilateral in 3D space, determined by the light source, a silhouette edge of the object, and a distant plane. Projecting the shadow quads to the image plane and rasterizing the projected shadow quads gives a pixel map that is used to modulate the intensity of the first-pass image, to simulate the effects of shadowing.

3.10. FULL SCREEN ANTIALIASING

The highest quality anti-aliased images can be obtained by repeatedly rendering a scene, varying the sub-pixel offsets for the pixel sample points, and compositing the contributions of the several passes, using weighting coefficients derived from an appropriate filter kernel. Denali hardware supports such sub-pixel sampling to an 11x11 sub-pixel resolution. These methods generally require Denali's doubleprecision accumulation buffer to maintain precision through repeated multiplyaccumulate operations. (See Section 4.4.2)

3.11. DOUBLE BUFFERING

Double buffering is a means of displaying only completely rendered images, avoiding displaying images while rendering is in progress. It is needed for good animation effects. Animation consists of displaying a sequence of images (called *frames)* in rapid succession, while some of the objects in the scene move or otherwise change from frame to frame. If the time required to render a frame is noticeable (frequently the case in practice), then animation without double buffering is unsatisfactory.

Double buffering makes use of two display buffers, which alternate roles as *refresh buffer* and *render buffer.* The refresh buffer supplies the data used in the video raster scan to refresh the screen image. While the rendering system is rendering a frame, it writes the output pixels to the render buffer, while the Video Output Section displays the image of the previous frame from the refresh buffer. When the image in the render buffer is completely rendered, then, through switching, the two buffers swap roles. The render buffer becomes the new refresh buffer in order to display the newly rendered frame, and the refresh buffer becomes the new render buffer to receive the rendering of the next frame. In this way, the viewer is presented only with completed frames, and never sees frames in the process of rendering.

Of course, before the rendering of the new frame begins, the new render buffer must be cleared of the previous image; that is, all of its pixels must be set to a background color, at least logically. It can take a significant amount of time to clear the buffer by actually writing the background color to all the pixels. Denali's Zero Time Clear` feature (ZTC), discussed below in Section 4.4.6, eliminates this time cost.

When a graphics display is used in a multi-tasking window environment, the enabling of double buffering, as well as the timing of the buffer role swapping, will naturally be window specific. Thus, the switching that selects the refresh buffer from the two physical buffers must occur on a per-pixel basis during the refresh cycle. In Denali, this window specific buffer switching is performed by the Data Path ASICs on the Frame Buffer Modules, discussed below in Section 4.4.4.

Even when the rendering time is very short, there is no benefit in swapping the buffer roles before the completion of the screen refresh cycle in which the new frame completes rendering. In Denali, the swapping of buffer roles for all double-buffered windows occurs only at vertical retrace time. This synchronization has implications for ZTC, as we shall see below.

3.12. **OVERLAYS**

An overlay is an image intended to be superimposed on a primary image in the display, usually covering only a fraction of the pixel locations. At most of the pixel positions the overlay is "clear" and the pixel has the color of the primary image; only the covered pixels have the overlay color. Overlays usually have relatively few bit planes and are typically used for grids, annotations, pop-up menus, and other graphical effects that are auxiliary to the primary image.

Hardware overlays are stored in a set of bit planes distinct from the primary refresh buffer and are implemented by per-pixel switching. That is, where the overlay color is "clear" the pixel data from the refresh buffer is used, otherwise the overlay color is used. Thus, overlays can be turned on or off and moved about the screen nondestructively, without disturbing the data in the primary image buffers. In Denali, the switching between primary image and overlay color occurs in the RAMDACs` in the Video Output Section. But since overlays are also window specific and can also be double-buffered, the Data Path chip on the FBMs is also involved in overlay control. Sections 4.4 and 4.5 give more details on the overlay function.

3.13. **STEREO**

Denali provides a stereo display by partitioning the display buffer scan lines into two sets, one for each eye, and displaying the two partitions alternately, each at 72 Hz. The stereo effect is realized by viewing the screen through glasses whose lenses are liquid crystal shutters that open alternately in synchronization with the refresh cycle. An infrared source mounted atop the monitor provides the synchronization signal detected by the glasses.

There are two stereo modes: one has 1280x512 resolution, and therefore non-square pixels, and the other has 640x512 resolution and square pixels. Both stereo modes can be double-buffered.

4. DENALI SUBSYSTEMS

4.1. IOPs

The Denali IOP ASIC is the central interface between the host computer and the Transform Engine Modules. It defines a Denali-specific DMA interface, consisting of a bi-directional parallel bus and a unidirectional serial bus through which Denali status information can be passed back to the host. The IOP connects to a hostspecific IOP via a 50-wire cable. The host-specific IOP is a board that plugs into and provides the interface to the high-bandwidth I/O channel of the host. Initially, hostspecific IOPs exist for Alpha/TURBOchannel hosts and for the Titan II.

On its other side, the Denali IOP communicates with the one to six Transform Engine Modules. Data passes between the IOP and the TEMs via two 32-bit buses, one shared by TEMs 0,2,4, the other shared by *TEMs* 1,3,5. **The IOP supports separate control signal lines to each of the six** *TEM* slots. **This radial control topology simplifies the bus protocol and reduces the overhead of IOP-TEM transactions. Thus, the IOP can transfer unique information simultaneously to two TEMs, and can broadcast simultaneously to all six.**

As described above in Section 3.1, in graphics rendering, software running on the host traverses the object data base and produces a stream of graphics commands, including primitives, attributes, and other commands to be interpreted by the Denali system. Denali commands are packaged in packets of one or more 32-bit words. They are passed from the host, through the host IOP to the Denali IOP. Some of the graphics commands, such as output primitives, are processed by any one TEM, so that potentially several TEMs are working simultaneously in parallel on different primitives. Some of the graphics commands that set rendering state, such as primitive attributes, are to be broadcast to all the TEMs.

Each TEM contains an input FIFO of programmable size, as described in the next section. The IOP-to-TEM buses include signal lines by which each TEM can signal whether its FIFO contains enough free space to receive a command. The IOP sends the commands that are intended for any one TEM on a round-robin basis to TEMs whose FIFOs have sufficient space. The maximum possible data transfer rate, host to TEMs via the IOP is 100 MB/s.

4.2. TRANSFORM ENGINE MODULES

Figure 2 depicts a block diagram of the Transform Engine Module. The heart of each TEM is the Transform Engine (TE), a programmable microprocessor whose functions include interpretation of the graphics commands passed from the host, geometry transformations, clipping, lighting computations, and other per-primitive or per

vertex operations. Generally, the TE performs all the floating point processing that occurs in the Denali. The TE breaks larger primitives (e.g. triangle strips) into smaller primitives (e.g. triangles) that can be consumed by the specialized scan conversion circuits on board, which are described below.

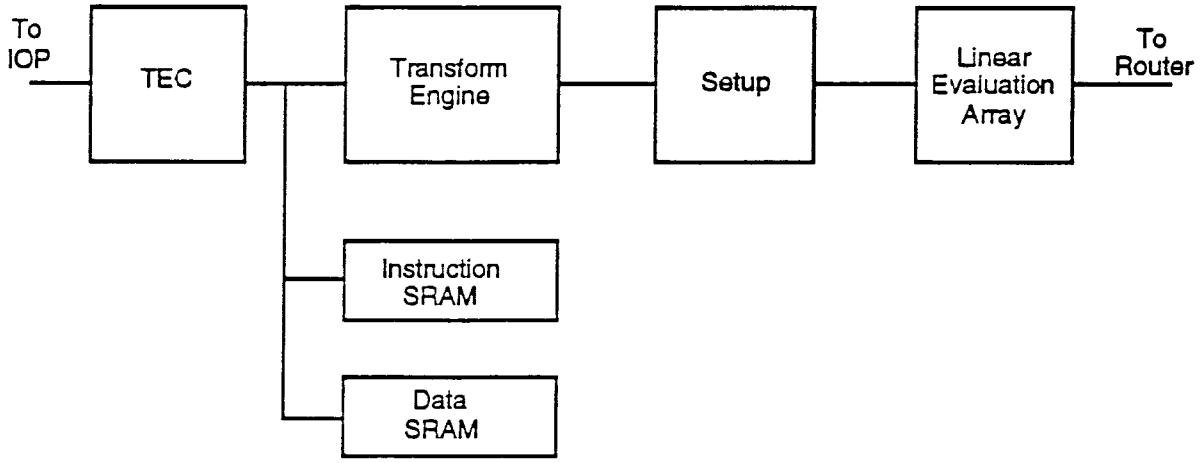

Figure 2 -- Transform Engine Module Block Diagram

The TE also executes a sphere rendering algorithm that includes both geometry processing and scan conversion. The algorithm implements both diffuse and specular reflection, computes depth values for each pixel, and correctly accounts for the effects of perspective foreshortening.

TEp has a special role in controlling the Data Path ASICs on the FBMs, the Pixel MUX ASIC in the Video Output Section, and the RAMDACs. All these elements, plus the IOP, share an auxiliary bus, called the RAMDAC bus, by which TEp can set the control registers and load the internal tables of these elements. The TEs can program the Pixel Engines on the FBMs by sending Non-Pixel Data pixel packets.

The Denali Transform Engine is implemented by a general purpose, highperformance 32-bit microprocessor, the Am29050. This processor features a high degree of on-board pipelining and parallelism. The integer unit, address bus, and floating point unit are all fully pipelined. The integer unit and floating point unit operate in parallel, and the floating point unit provides simultaneous addition and multiplication. The Am29050 has 192 general registers and a 1-KB Branch Target Cache`, which ensure that instructions and operands are almost always available when needed. It has separate data and instruction buses, which support burst-mode memory access, so it can achieve transfer rates to memory up to 160 MB/s. Peak integer performance is 32 MIPS and peak floating-point performance for multiplyaccumulate operations is 80 MFLOPS. On Denali, the Am29050 is used exclusively in a physical addressing mode. A Denali can have up to six Transform Engines, one on each Transform Engine Module.

The Transform Engine Modules each contain one MB of fast SRAM, half a megabyte each for instructions and data. The input FIFO that receives graphics commands from the host resides in Data SRAM. The code that the TE executes is all in Instruction SRAM, downloaded at system initialization time.

The TE and its supporting SRAM are augmented by three custom ASICs on the TEM board: the Transform Engine Controller ASIC (TEC), the Linear Evaluator Array ASIC (LEA), and the Draw Setup ASIC (SETUP).

The Linear Evaluator Array (LEA) is the scan conversion engine. The LEA samples each given primitive to determine which screen pixels the primitive affects and to determine the pixel values that the primitive contributes to each affected pixel location. LEA primitives are points, triangles (filled), and lines (with width). Polygon outlines can be rendered like lines. The pixel values computed by the LEA include color and, potentially, depth, alpha coefficient, texture coordinates, and the distance functions used in antialiasing. Thus, the LEA is a linear equation engine and a linear interpolation engine. Each LEA can evaluate up to ten geometry and attribute linear functions in parallel. There are five geometry evaluators: four edge evaluators for resolving in/out pixels and one depth interpolator. Five attribute evaluators can interpolate color, alpha, and texture coordinates. To support texture mapping, the color contributions of diffuse and specular reflection may be interpolated separately.

The LEA begins at a vertex of a given primitive (triangle or vector) and executes a scan algorithm that evaluates each pixel with respect to the edges of the primitive and computes its interpolated pixel values. The result of the edge evaluation on each pixel determines the next pixel to be tested in the scanning process, which will usually be one of the pixels adjacent to the current pixel on the same scan line or an adjacent scan line. The scanning algorithm is efficient in that it touches the minimum number of pixels necessary to render the primitive, at least for primitives that do not cross clipping boundaries. The output of the LEAs are pixel packets to be sent to the Frame Buffer Modules via the Router.

Registers on the LEA hold the coefficients of the linear equations that represent the edges that define the primitive. These coefficients have to be computed for each primitive from the list of transformed vertex coordinates. The Draw Setup ASICs function is to compute the linear coefficients from the defining vertices and write them to the LEA registers to set up the LEA for each primitive. It contains a 32-bit multiplier and arithmetic-logic unit.

The Draw Setup chip and the LEA use fixed-point arithmetic. Vertex coordinates are carried to 16-bit precision, with four bits of sub-pixel resolution. Depth value is carried to 24-bit precision. Texture coordinates have 24 bits with 11 bits of sub-texel resolution. Color and transparency values may have eight or 16 bits per channel. The double-precision pixel values are needed for the multi-pass rendering methods.

The LEA and Draw Setup chip contribute substantial processing power to the TEM functionality, fully comparable with the computing power of the TE itself.

The Transform Engine Controller (TEC) controls the several data paths connecting the memory and computational elements of the TEM. The TEC also manages communication between its TEM and the IOP. Graphics commands coming from the host via the IOP are received by the TEC and stored in an input FIFO in the TEMs data SRAM. The TEC contains registers that hold the pointers that define the FIFO's state. The FIFO length is programmable.

4.3. ROUTER

The Router is a wide cross-bar switch that sends pixel packets from the TEMs to the correct FBMs. Each pixel packet coming from a TEM consists of one to five 48-bits words. Each TEM has its own 48-bit bi-directional bus to the Router, plus a five-bit unidirectional bus that signals the destination FBM. There are ten parallel 24-bit bidirectional buses connecting the Router with the FBMs. So the Router can communicate simultaneously with six TEMs and ten FBMs. In a 20-FBM system, each FBM bus is shared by two FBMs. The Router can also duplicate data from a TEM and broadcast it to all the FBMs.

The Router has a bit-sliced architecture, implemented with six copies of a custom ASIC. Each Router chip has inputs to receive eight bits from each of the six 48-bit TEM buses, four bits from the low order 24 and four bits from the high order 24. Each Router chip has outputs for four bits of each of the ten 24-bit FBM buses. In every clock cycle, each chip performs identical operations, so the Router is considered as a single unit. The Router splits each 48-bit word received from the TEM and sends it to the destination FBM as two 24-bit words (the upper half followed by the lower half).

4.4. FRAME BUFFER MODULES

As its name implies, each Frame Buffer Module (FBM) contains frame buffer memory, which includes the memory that holds the image that is used to refresh the display, as well as considerable off-screen frame memory to support the functions described in Section 3. Each FBM contains multiple buffers for a fixed interleaved subset of the screen pixel locations. A **Denali system can contain five, ten or twenty FBMs, and so each contains one-fifth, one-tenth, or one-twentieth of the total frame buffer memory.**

Besides the frame buffer memory, each FBM contains two important custom ASICs:

- **the Pixel Engine ASIC (PE), which receives the pixel packets from the TEMs via the Router, performs all the per-pixel computations, and finally writes pixels to the frame buffers, and**

- the Data Path ASIC (DP), which provides the interface between the FBM and the Video Output Section, serially reads out pixel data from the display buffer synchronously with the screen refresh cycle, and performs the switching needed for window-specific double buffering, Zero Time Clear, and hardware cursor functions.

Sections 4.4.1 and 4.4.2 give more detail on the organization of the FBM buffers; Section 4.4.3 gives more detail on the Pixel Engines, and Section 4.4.4 describes the operation of the Data Path ASIC.

4.4.1. FBM *INTERLEAVE*

In terms of pixel locations, each FBM contains the buffer memory for an interleaved portion of the screen. The benefit of the interleaved frame buffer memory organization is increased opportunity for parallel computing. The pixel packets that arrive from the TEMs in close time proximity are likely to have come from the same small primitive, and so are likely to be clustered in a small area of the screen. The interleaved memory organization allows simultaneous access to spatially proximate pixels. The several FBMs work in parallel, each on pixel locations in its own interleave.

The interleaving in the x-direction is five-fold. In the y-direction the interleave depends on the configuration--one-fold, two-fold, or four-fold, for the five-, ten-, and twenty-FBM configurations respectively. Specifically, in a five-FBM system, FBM_O includes pixels 0 to 3, 20 to 23, 30 to 33, etc., of every scan line; FBM, includes pixels 4 to 7, 24 to 27, etc., of every scan line. And so on for FBM_2 to FBM_4. In a ten-FBM configuration, each FBM contains pixels only on every other scan line. The even numbered FBMs cover the even scan lines and the odd FBMs cover the odd scan lines. In a twenty-FBM configuration, each FBM contains pixels only on every fourth scan line.

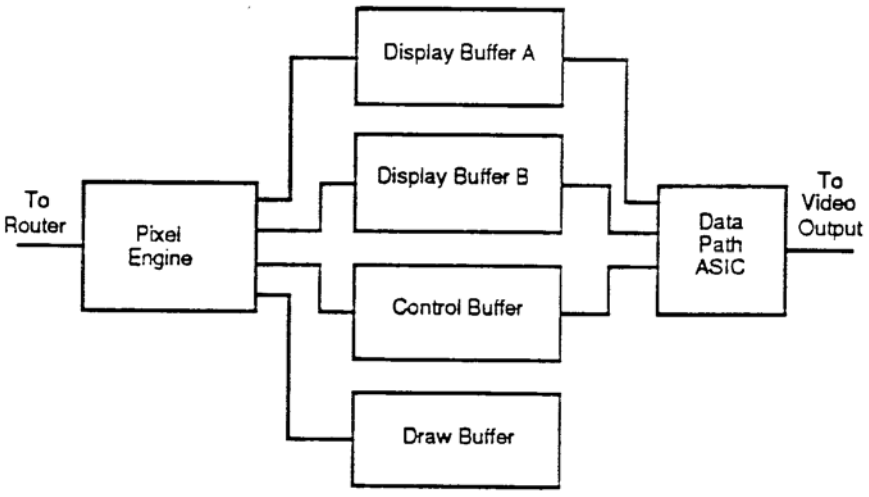

Figure 3 -- Frame Buffer Module Block Diagram

The reason that the interleaves each contain runs of four consecutive pixels on a scan line is to take advantage of the page-mode access to the RAMS, in which several consecutive locations can be accessed on successive clock cycles, without having to repeat a column address cycle for each access.

4.4.2. FBM MEMORY BUFFERS

Figure 3 shows a block diagram of the FBM. The complete frame buffer memory is divided into three parts: Display Buffer, Control Buffer, and Draw Buffer. Display Buffer includes both the refresh buffers that hold the image data used to refresh the display and the render buffers to which images are rendered in double buffering. (See Section 3.11). A Denali system contains at least two complete 1280x1024x24 bit RGB frame buffers, designated Display Buffer A and Display Buffer B. For double buffering, these two buffers provide the refresh buffer and render buffer, in a windowspecific way. That is, on any given screen refresh cycle, some pixels are refreshed from Display Buffer A and some pixels are refreshed from Display Buffer B, depending on the active windows in which they lie. The switching is performed by the Data Path ASIC, described in Section 4.4.4. The Display Buffer is implemented in video RAM (VRAM), a sort of dual-ported memory that provides random access for drawing a picture and simultaneous serial read-out for refreshing the display.

The Control Buffer has 1280x1024x16 bits. Six bit-planes constitute a pixel-specific Window ID. Thus, up to 64 simultaneous windows can be accommodated by the hardware window switching scheme. Five of the Control Buffer bits are for overlay control and five are for ZTC control, as discussed further below. The Control Buffer is also implemented in VRAM, and, in order to support the ZTC feature, part of the Control Buffer is implemented in a special three-port VRAM which provides a serial write access simultaneously with the serial read out.

Finally, the frame buffer memory includes a large Draw Buffer, comprising from two to sixteen full 1280x1024x32-bit buffers. These planes can be used as 24-bit depth buffer, eight-bit stencil buffer, 32-bit RGBa pixmaps, or 64-bit RGBa accumulation buffers, and for storage of textures and filter coefficient tables. The double-precision accumulation buffer is needed in multi-pass rendering methods in which each pixel's color is determined by combining the effects of a number of different sampling operations.

Each FBM contains the memory for the parts of the Display, Control and Draw Buffers in the FBM's own interleave. In a five-FBM system, each FBM must contain 1.5 MB of Display Buffer memory, in order to provide a full double-display buffer, and 0.5 MB of Control Buffer memory. In the ten-FBM and 20-FBM configurations, each FBM contains 0.75 MB of Display Buffer memory and 0.25 MB of Control Buffer Memory. In a twenty FBM system, there are four full 24-bit Display Buffers. The

extra Display Buffer memory can be used together with the corresponding Control Buffer memory, as an extension to the Draw Buffers.

Each FBM can have 2 MB or 4 MB of Draw Buffer memory, which is implemented in DRAM.

4.4.3. PIXEL ENGINE

There is one Pixel Engine ASIC (PE) per Frame Buffer Module. The PEs act as controllers for the frame buffer memory, each responsible for the interleave on its FBM. They receive the pixel packets from the TEMs via the Router. Each PE contains an on-board FIFO sufficient for holding up to 20 pixel packets, and each is connected to the Router via a 24-bit bi-directional data bus (that it may share with one or three other PEs, depending on the configuration) and two handshake lines.

The messages from the TEMs to the PEs consist of packets of one to five 48-bit words. Some one-word message types are Non-Pixel Data, used to program the PEs by setting their control registers. The remaining types of packets are Pixel Data. See Section 3.3 for a discussion of the different sorts of information that pixel packets can contain.

A **PE contains four 16-bit multiply-accumulators, four 32-bit logic units, internal lookup tables, and pointers for accessing external lookup tables. The lookup tables typically contain coefficients to use in scaling and mapping of pixel values, such as filter coefficients to be used in antialiasing.**

PEs can remap the incoming pixel color values according to lookup tables in FBM memory. (However, this is not the normal means of providing pseudo color mapping on Denali; normal color maps are contained in the RAMDAC, as discussed below in Section 4.5). Then, the PE can modify an arriving pixel value according to other data contained in the pixel packet. For example, in the case of anti-aliased lines and polygon outlines, the pixel packets contain a parameter measuring the distance of the pixel from the line; this distance parameter is used as index into a table of filter coefficients, which are used as multipliers of the pixel color values. Further, the PEs can read pixel RGB data from the Draw Buffer or RGBa data from the Display Buffer, to be combined with the arriving pixel data in various ways, supporting transparency, anti-aliasing, and several kinds of multi-pass rendering methods, as described in Section 3. In texture mapping, the PEs perform the perspective division of texture coordinates, mentioned in Section 3.7, before using the texture coordinates to look up the texture value. Finally, the PEs perform write masking based on depth comparison and stencil comparison, reading from the depth buffer or stencil buffer in the process.

The PEs also have occasion to write to the Control Buffer. They write the overlay data, and can apply the full range of blend operations, depth comparison, stencil comparison, and Window ID comparison conditions to the overlay write. Further, each time a PE writes to a rendered pixel, it sets the appropriate ZTC bit to flag the pixel as a foreground pixel. (See Section 4.4.6).

Generally the PE accesses frame buffer memory randomly in response to arriving pixel packets. Its high-bandwidth connection to the memory buffers is a significant factor in its performance. The PE can access the Display Buffer, the Control Buffer and the Draw Buffer simultaneously, transferring up to 72 bits of pixel data per memory cycle. For page-mode access, the peak transfer rate between one PE and the frame buffer memory is 60 MB/s. Thus the peak transfer rate of all PEs to memory can be as large as 1200 MB/s in a 20-FBM system.

4.4.4 DATA PATH ASIC

The Data Path ASIC (DP) on each FBM provides the interface between the Display Buffer memory and the Video Output Section. It performs the essential switching that implements window-specific double buffering, the hardware cursor, and Denali's Zero Time Clear feature (ZTC), described below. The DPs receive the pixel clock signal from the Pixel MUX chip set on the motherboard, and each DP generates an appropriately phased clock for the serial read out from its own frame buffer. On each pixel clock tick, each DP passes to the Video Output Section all the information needed to draw one pixel on the screen.

Each DP has separate 24-bit input ports for Display Buffers A and B, and a 16-bit port for the Control Buffer. The DP reads out simultaneously the color values from both Display Buffers and the control information from the Control Buffer, serially in phase with the video refresh scan cycle. For each pixel in the scan, the DP passes a single 24-bit color value and three bits for color map selection on to the Video Output Section. The transmitted color value may come from either of the two Display Buffers, or may come from internal tables in the case that the pixel lies under the cursor or the case that it is a background pixel when ZTC is in effect.

Each DP contains a Window Attribute Memory (WAM), containing 32 bits of attribute information for each of the 64 possible windows. One of these window attribute bits tells whether Display Buffer A or B is the refresh buffer for the corresponding window. This information, together with the Window ID in the control buffer, tells the DP which of the two Display Buffers is the proper source for the present pixel on the present refresh cycle. Of the remaining WAM bits, one controls overlay double buffering, three select the RAMDAC color table, 24 contain the window background color for use by the ZTC feature, and three control the window specific ZTC function, as described below in 4.4.6.

The DP also passes further control information through to the Video Output Section. For example, the RAMDAC provides five independent color lookup tables for pseudo color maps. Each window entry in **the WAM contains three bits that select the color table in the RAMDAC; the DP passes the color table selection bits to the RAMDAC in a window-specific way. The DP also passes the overlay select data to the RAMDACs. The five overlay bits can be used as a single-buffered five-plane overlay or a double-buffered two-plane overlay, in a window-specific way.**

The DP chips are all on the RAMDAC control bus. By this path, TE_O can set the Window Access Memory, cursor control memory, and other internal registers.

4.4.5. CURSOR

Each DP contains a pair of registers that hold the screen position of the cursor, and a 32x32x2-bit cursor shape pixmap. Therefore, the DP is able to detect when a pixel lies under the cursor position, and in this case the DP sends on to the Video Output Section a cursor color, selected from three-color internal color map, instead of a color from the Display Buffers. The cursor position stored in all the DPs is updated by the host software on each video retrace cycle, according to the movement of the pointer input device. All the DPs contain the same cursor information.

4.4.6. ZERO TIME CLEAR

In double-buffered rendering, before beginning the rendering of each frame, the render buffer must be cleared; that is, all of the pixels must be assigned the background color. When depth buffering is in effect, then the depth buffer must also be initialized before rendering by giving all pixels the minimum depth value. The time required to initialize these buffers directly by writing the background values to each pixel can be significant in relation to the frame rendering time, especially if the window is large. Denali's Zero Time Clear feature (ZTC) eliminates the time that would be needed to initialize the render buffer and the depth buffer. Under ZTC, background color and depth values are never actually written to the render buffer and depth buffer. Rather, flags in the Control Buffer distinguish background pixels from foreground pixels. These Control Buffer flags are initialized for each rendered frame with zero time cost, during a single screen refresh cycle. Because of parallel access to the three buffers, the PE can set the flag values for the foreground pixels during frame rendering and the DP can test them during screen refresh, both with zero time cost.

Five bits in the Control Buffer are used for the ZTC flags. Entries in the Window Attribute Memory within the DP ASICs provide for using ZTC on a window-specific basis. For each window, this entry specifies whether ZTC is enabled, which Control Buffer bits control the ZTC function for each rendered frame, and the color to be used for the background pixels.

The DP also passes further control information through to the Video Output Section. For example, the RAMDAC provides five independent color lookup tables for pseudo color maps. Each window entry in **the WAM contains three bits that select the color table in the RAMDAC; the DP passes the color table selection bits to the RAMDAC in a window-specific way. The DP also passes the overlay select data to the RAMDACs. The five overlay bits can be used as a single-buffered five-plane overlay or a double-buffered two-plane overlay, in a window-specific way.**

The DP chips are all on the RAMDAC control bus. By this path, TE_0 can set the Window Access Memory, cursor control memory, and other internal registers.

4.4.5. CURSOR

Each DP contains a pair of registers that hold the screen position of the cursor, and a 32x32x2-bit cursor shape pixmap. Therefore, the DP is able to detect when a pixel lies under the cursor position, and in this case the DP sends on to the Video Output Section a cursor color, selected from three-color internal color map, instead of a color from the Display Buffers. The cursor position stored in all the DPs is updated by the host software on each video retrace cycle, according to the movement of the pointer input device. All the DPs contain the same cursor information.

4.4.6. *ZERO TIME CLEAR*

In double-buffered rendering, before beginning the rendering of each frame, the render buffer must be cleared; that is, all of the pixels must be assigned the background color. When depth buffering is in effect, then the depth buffer must also be initialized before rendering by giving all pixels the minimum depth value. The time required to initialize these buffers directly by writing the background values to each pixel can be significant in relation to the frame rendering time, especially if the window is large. Denali's Zero Time Clear feature (ZTC) eliminates the time that would be needed to initialize the render buffer and the depth buffer. Under ZTC, background color and depth values are never actually written to the render buffer and depth buffer. Rather, flags in the Control Buffer distinguish background pixels from foreground pixels. These Control Buffer flags are initialized for each rendered frame with zero time cost, during a single screen refresh cycle. Because of parallel access to the three buffers, the PE can set the flag values for the foreground pixels during frame rendering and the DP can test them during screen refresh, both with zero time cost.

Five bits in the Control Buffer are used for the ZTC flags. Entries in the Window Attribute Memory within the DP ASICs provide for using ZTC on a window-specific basis. For each window, this entry specifies whether ZTC is enabled, which Control Buffer bits control the ZTC function for each rendered frame, and the color to be used for the background pixels.

Kubota Pacific offers an optional auxiliary video board that provides output in both the NTSC and PAL broadcast video formats. The auxiliary video board is a daughter board which plugs into a connector provided on the Denali motherboard. It puts out video signals that capture an appropriately sized window of the high-resolution screen (640x480 for NTSC and 768x576 for PAL). At any time this board is either in NTSC mode or PAL mode, and for each of the broadcast formats, it provides simultaneous outputs for three-wire RGB, two-wire S-video, and one-wire composite video. The video timing can be genlocked to an external signal. Use of the auxiliary video board does not impact the operation of the normal high-resolution display. A control port is available for interfacing with animation controllers.

S. DENALI IMAGING AND VOLUME RENDERING

Denali provides a powerful and flexible architecture for performing various image processing and volume rendering operations, by using parallel algorithms implemented on the Transform Engines, on the Pixel Engines, or on both working together in a pipeline. The very high-bandwidth data paths connecting the several processing elements provide an additional unique benefit for imaging applications. This section describes basic image processing and volume rendering functions that can benefit from acceleration by Denali hardware and firmware.

5.1. IMAGE PROCESSING AND GRAPHICS

The graphics rendering processing described above in Sections 3 and 4 begins with geometric models to produce pictures in the form of digital images (pixel data). Image processing, on the other hand, begins with digital images as its initial data. Frequently, the objective of image processing is to produce new images, which will be in some sense preferable to the initial images (image enhancement), but it sometimes also has the goal of extracting high-level information from given images (image understanding or image classification).

Historically, image processing and geometry-based graphics have been distinct activities. But more recently, the two disciplines are tending to converge. One reason for this convergence is that the newer high-quality rendering features, such as antialiasing, texture mapping, and transparency, are essentially image processing operations. Therefore, Denali's hardware provisions for quality rendering also naturally provide substantial support for image processing.

Since image data sets are inherently very large, the performance of every image processing operation depends strongly on the speed with which images and subimages can be copied from one part of image memory to another, including both the host system memory and the various frame buffers of the imaging system (put-image or block transfer). Denali provides very fast put-image because of the high-bandwidth data paths connecting its pipeline stages and connecting it to the host.

5.2. CINE LOOPS AND IMAGE PANNING

Cane loops are animation effects created by displaying a sequence of related images stored in memory, in rapid succession, usually in a repeating cycle. The typical use of rind loops is to present temporal sequences of images. Panning **applies to the situation where part of a stored image is displayed in a window that is smaller than the whole image, and the displayed portion is varied by copying different parts of the image to the memory mapped to the window. Interactive control and smooth motion are frequent requirements for panning applications.**

Denali's superior put-image performance greatly enhance its cine loop and panning functionality. Moreover, the presence of processing elements (TEs and PEs) on the data path allow interesting variations on the standard image transfer operations, such as combining them with the contrast stretching and image scaling operations, described below.

5.3. CONTRAST ENHANCEMENT AND HISTOGRAM OPERATIONS

After simple image transfer, the simplest image processing operations are those that involve recomputing the pixel values without having to combine the values of different pixels. **Such per-pixel operations usually have the goal of enhancing the image, helping the viewer to see better the information content of the image data by providing a better match between the displayed pixel intensities and the perceptual discrimination characteristics of human vision.**

The most common simple pixel operations are linear transformations of pixel values and remapping pixel values by table lookup. For example, *contrast stretching is* **used on an image whose pixel values are concentrated in a small portion of the available range of pixel values, to spread the pixel values over a wider range, thus converting imperceptible intensity gradations into perceptible variations.**

The same kind of linear transformations of pixel values are required to solve the problem of mapping images acquired with ten or twelve bits per pixel to eight-bit images.

In Denali, both the Transform Engines and the Pixel Engines are capable of performing these pixel-value transformations as the image data flows through the pipeline. The possibility of applying the transformations at two different pipeline stages entails flexibility for combining them with other image processing operations. Denali's capability for 16-bit pixel arithmetic further enhances its applicability to these operations.

A *histogram* **of a set of sampled values is simply an enumeration of the number of sample values falling into each of a set of intervals that covers the entire range of values. Some pixel value transformation methods involve evaluation of histograms or other statistics on all the pixel values, in order to determine the lookup tables for remapping them. On Denali, the TEs can evaluate the histograms as image data flows through the TEMs, and then download the resulting lookup tables to the FBMs. Then the PEs can apply the specific contrast enhancement manipulation to the image data.**

5.4. IMAGE SCALING, ROTATION, AND WARPING

The most costly image processing operations are those that involve resampling the image data. These include the simple geometric transformations, such as image

scaling (i.e. zooming and shrinking), image rotation, and linear and non-linear image warping. Applying these geometric transformations to images is somewhat more difficult than applying the corresponding operations to geometric model data before rendering. As discussed above in Section 3.7, image resampling always involves difficulties with aliasing artifacts, which require filtering and blending operations for good results. However, the basic image processing operations of image scaling, image rotation and image warping can all be regarded as special cases of texture mapping. Therefore, Denali's machinery for resampling, bilinear interpolation, filtering, and MIP mapping, all discussed in Section 3.7, are equally applicable to the image transformation operations.

5.5. CONVOLUTIONS

Convolution is a filtering operation that involves replacing each pixel value by a weighted average of itself and a set of its neighbors. It can be applied as an antialiasing measure for the kind of image resampling operations discussed in Sections 5.3 and 3.7. It also has applications in certain image enhancement operations, such as edge enhancement, blurring and sharpening.

The size and shape of the averaging neighborhood and the exact weighting function constitute the filter *kernel.* **Different cost/quality tradeoffs result from different filter kernels. Denali can provide high-speed implementation of convolution filters with user-specified kernels using the TEs and PEs together in a pipeline.**

5.6. IMAGE OVERLAYS

Denali's provisions for hardware overlays, described in Sections 3.12 and 4.4, are applicable to imaging applications as well as graphics applications. In particular, the overlay mechanism provides a convenient means of displaying graphical and textual information along with image data, panning and rotating the graphics together with, or independently of, the image data. These features are important in electronic light tables and remote sensing applications.

5.7. VOLUME RENDERING

Digital images consist of sets of sample data on uniform planar grids of sample points (pixels). The power of image processing methods in medical, strategic, geophysical, and remote sensing applications derives from the fact that any set of sample data on a regular planar grid may be interpreted and visualized as image data. Volume data generalizes this idea to three dimensions. A *volume* **data set consists of data elements, called voxels, associated with a regular grid of sample points in three-dimensional space. Because the world is three-dimensional, most of the application domains that use conventional imaging methods also naturally produce volume data sets as stacks of 2D image sets.**

Volume rendering refers to methods for producing 2D images from volume data sets to afford the user insight into the content of the volume data. Volume rendering methods are extremely demanding because of the very large size of volume data sets with usable spatial resolution. Volume rendering on Denali benefits from the large frame buffer memory and the parallel processing available both on the Frame Buffer Modules and the Transform Engine Modules. In these methods, the volume data set is partitioned into subvolumes that can fit on a single FBM. All of these methods involve some form of resampling the volume data, and thus are amenable to the existing machinery on the FBMs for 3D texture mapping. Resampled values can be determined either by the nearest neighboring voxel to the sample point or by trilinear interpolation from the eight nearest voxels that box a sample point.

KPC has developed low-level software and firmware to implement several volume rendering methods--multiplanar reformatting, maximum intensity projection, and isosurface rendering, all described below.

Multi-planar reformatting consists simply of displaying 2D images derived from the volume data set by passing plane sections through it. The derivation of the image on any plane section is a problem of image resampling. An arbitrary grid of sample points on the section plane is selected, and the pixel value for each sample point is determined either from the nearest voxel or from tri-linear interpolation from the eight voxels that bracket the sample point. Giving the user interactive control of the position and orientation of the section plane in multi-planar reformatting can help him explore the volume data set and gain understanding of its contents.

The remaining volume rendering methods implemented on Denali all make use of casting parallel rays into the volume, one ray for each pixel in the view plane, and determining each pixel's color according to samples of the volume data taken along the corresponding ray. In *maximum* intensity *projection* each pixel is assigned a value corresponding to the maximum of the voxel values sampled along the ray. This technique is effective when the voxel data has sharp maxima in interesting features with geometrically fine structure. An example of this situation is provided by blood vessels in medical magnetic resonance data.

In general, an isosurface of a 3D data distribution consists of the points on which the data has a specified constant value. Science and engineering has long used isosurfaces to visualize 3D data distributions (also called *fields*). Surface rendering methods applied to volume data sets seek to display estimated isosurfaces determined through sampling and interpolation of the volume data. The Denali implementation of surface rendering casts parallel rays into the volume for each pixel in the view plane, and stores in a depth buffer the depth values where the voxel values sampled along each ray first cross a specified threshold. Then the pixels are colored according to a simple depth-gradient shading algorithm applied to the buffered depth values, that shows the relief of the surface. One can display interesting surfaces by choosing

the threshold value to correspond to appropriate interfaces, such as the transition from soft tissue to bone in medical imaging data.

Denali's volume rendering methods can be used together with the graphics rendering techniques described in Sections 3 and 4 to produce images combining voxel data and geometric models. The depth-buffering mechanism provides the means of blending the rendering from the two kinds of sources, and also allows combining of two volume rendering methods, such as multiplanar reformatting and surface rendering.

All of the just-described volume rendering methods are most effective when they can be computed in interactive time, so that a user may explore a volume data set by interactively rotating, zooming, and adjusting threshold values and sampling frequencies. Denali achieves interactive volume rendering through the parallel processing by the Pixel Engines, working on volume data sets distributed over the Frame Buffer Modules.

6. SOFTWARE ARCHITECTURE

Kubota Pacific Computer supplies the host software needed to access the functionality of the Denali system and to manage its resources. The supplied software includes both the high-level application programming interfaces, which give users access to the Denali graphics functionality, and the supporting low-level software elements, which directly control the hardware but are not normally visible to the user. The lower level software shields the higher levels from the details specific to the hardware. The high level interfaces conform to industry standards, either official or *de facto.*

This section describes the Denali software architecture as it is designed and planned for full development, but without regard to the schedule for the release of any particular item. Ask Kubota Pacific sales representatives for availability and release dates and announcements of further functionality.

The low-level software base, denoted VLL, is broad in function; it includes the device driver function, basic window management and resource management functions, 2D drawing, accelerated 3D graphics, and imaging functions. User access via VLL is not necessary, since virtually all the Denali hardware functionality and performance levels are accessible through supported higher-level interfaces.

The higher-level interfaces split into two groups: network-transparent interfaces and peer interfaces. The network-transparent interfaces are all based on extensions of the X Window System, described below. They allow Denali to be used as an output device for applications running on remote hosts. The peer interfaces are based on XPeer, KPC's unique peer extension to the X Window system, also discussed below.

6.1. THE X WINDOW SYSTEM

Normal user access to the Denali display is through the X Window System, and certain extensions. X **is an industry standard, network-transparent window system based on a client-server model. It is developed and maintained by Massachusetts Institute of Technology with the support of a consortium of comprising virtually all major workstation vendors, including Kubota Pacific.**

The X Server controls the interactive user workstation, consisting of the display surface and several input devices. Several simultaneous client processes, running on the local host or remote hosts, may share the display surface, each owning one or more windows. The client processes communicate with the server through the X protocol, which is a convention for communicating window management functions and simple 2D drawing semantics. In a networking environment, the X protocol rests on the

lower-level basic network protocol in use (such as TCP/IP). **When the client and**

server reside in the same host, the X protocol can be supported by a shared memory transport mechanism.

Denali's X Server is a well-tuned, high-performance implementation of X11R5, the current standard version of the X system. It includes the standard MBX multibuffering extension, implemented to make use of Denali's hardware features that support window-specific double buffering and the Zero Time Clear feature. Further supported extensions are XInput, which handles general interactive input devices, Xtrap, which is a tool for debugging, SHM for shared memory transport, and Shape, which provides non-rectangular windows. Denali's hardware overlay is supported as an X visual type.

The lowest-level interface to the X Server that application programs normally use is Xlib, a library of about 400 routines for basic window management, 2D drawing, and input handling functions. Also included with the X11R5 system is the Xtk tool kit, which is a core function library for constructing the elements of a window-based user interface: buttons, dials, scroll bars, dialog boxes, and so on. In addition, the Alphabased hosts run the OSF/1 operating system, which includes Motif, a system for constructing user interfaces in a common style.

6.2. XPEER EXTENSION

A **unique feature of the Denali X Server implementation is XPeer, an extension to accommodate** *peer interfaces.* **A peer process can access the Denali hardware directly, avoiding some overhead and data copy stages inherent in the X Protocol, while still respecting the window management functions of the X server. A peer programming interface can provide greater performance for the applications that link to it, especially for applications that have to send large data sets to the graphics subsystem. Peer interfaces give up network transparency, so are appropriate only for applications that run on the local host.**

A **peer process has privileges whose use could possibly interfere with the activities of other processes sharing the display. Hence, the peer extension is not intended for direct use by application programs, but rather only to support high-level programming interfaces. Several of the application programming libraries discussed below are (will be) implemented as peer interfaces.**

6.3. DISPLAY POSTSCRIPT

Display PostScript is an application programming interface licensed from Adobe Systems, based on the PostScript page description language. It emphasizes 2D drawing and a rich set of text fonts. For Denali it is implemented as standard extension to the X Window system, and so offers network transparent access.

6.4. PEX

PEX is an X consortium standard extension to the X protocol for communicating 3D graphics semantics across the X network interface to control 3D graphics acceleration hardware. A PEX implementation must extend the X Server to process the PEX commands and pass them along to the low-level device-specific software levels.

The 3D graphics functionality supported by standard PEX semantics is very similar to the functionality addressed by the PHIGS standard, including the PHIGS PLUS extensions. (See next section). While PHIGS is the prototypical API for accessing PEX functionality, other APIs are available for PEX, including PEXIib, discussed below in Section 6.6. PEX offers control of the graphics hardware at a lower level than the PHIGS semantics calls for, and allows for both immediate mode and structure mode graphics programming styles. PEX structure mode supports serverside structure storage, which can yield substantial performance gains for applications that traverse structures repeatedly with only few changes, as in animation.

The Denali implementation of PEX includes all of the standard version 5.1, with certain extensions to allow access to the Denali features that transcend the PHIGS standard. It is derived from the PEX-SI sample implementation of PEX, and is tuned for high performance on the Denali system.

6.4. PHIGS

PHIGS is the ANSI/ISO standard 3D graphics application programming interface. (The term "PHIGS" is now generally understood to include the "PHIGS PLUS" extensions adopted in 1992.) PHIGS is a relatively high-level interface, oriented towards a structure-mode (or display-list) style of graphics programming. That is, in standard PHIGS, applications build editable, hierarchical structures of graphics commands, and the supporting run time software performs the structure traversal to render pictures. The PHIGS rendering model includes lighting and shading functionality, as well as high-level primitives, such as sets of indexed polygons and rational B-spline curves and surfaces. But it does not address some of the advanced rendering features of Denali, such as antialiasing, transparency, texture mapping, and shadows.

Kubota Pacific supplies DEC PHIGS, the PHIGS implementation developed by Digital Equipment Corporation. DEC PHIGS communicates with the Denali hardware through the PEX extension to the X Window protocol, described above. In particular, PHIGS structures may be stored either on the client side, where they are traversed by the PHIGS run-time support, or on the server side, where they are traversed by the PEX server. DEC PHIGS also has some extensions to the PHIGS standard, including an immediate mode (in which graphics commands are sent

immediately to the hardware without being retained in structures), transparency, and anti-aliased line drawing.

6.5. PEXlib

PEXlib is an X Consortium standard programming interface to PEX functionality. Thus, PEXlib includes virtually all the semantics of PHIGS. In addition, it offers some access at a lower level with finer control of the hardware. Another distinction between PEXlib and PHIGS is in how they fit into the X Window System. PHIGS was designed before the

advent of window-based workstation user environments, and the PHIGS standard does not speak to how PHIGS is to work with an ambient window system. Thus, the way that PHIGS interacts with X varies substantially among different PHIGS implementations. PEX, on the other hand, is a standard extension to X, and PEXlib extends Xlib in a seamless and natural way.

6.6. NPGL AND OPENGL

GL, for "Graphics Library", is the graphics programming interface provided by Silicon Graphics, Inc. for their workstations. Although proprietary, GL has attained status as a *de facto* **standard because of the number of applications that have been written to it. In many ways, GL can be considered a lower-level interface than PHIGS; it offers fewer favors for the programmer, and it allows a finer degree of control of the graphics system. NPGL is a portable version of GL 4.0 developed by Nth Graphics,Inc. Kubota Pacific has worked with Nth Graphics, to port NPGL to Denali, as a high-performance peer interface.**

OpenGL is a new API, derived from, similar to, but not compatible with GL 4.0. Silicon Graphics has licensed OpenGL to a number of vendors, and an industry Technical Review Board controls its development. Kubota Pacific includes support for OpenGL in the planned Denali software architecture.

6.7. AVS

AVS (after "Application Visualization System") is a package for visualization and display of geometric modeling data, multi-dimensional field data, and image data. It is licensed from Advanced Visual Systems, Inc. It has gained wide use in the scientific computing community since 1988. AVS includes a graphical interface for defining the display of given data sets by constructing data-flow diagrams linking processing modules. Further, it is extensible in that users can add new modules. Kubota Pacific participates in a consortium that supports a center for exchange of AVS modules, based at the University of North Carolina.

Kubota Pacific supplies an implementation of AVS that is optimized to make use of the graphics and imaging support of Denali hardware.