High Quality Rendering Using the Talisman Architecture

Anthony C. Barkans

Microsoft

One Microsoft Way

Redmond, WA. 98052

tbarkans@microsoft.com

Abstract

Currently graphics devices that offer both high performance and high quality interactive rendering have been priced at a level that places them out of the reach of the broad number of users that constitutes the mass-market. Because of the cost constraints placed on graphics devices designed for the mass-market, they often trade off image quality in order to get reasonable rendering rates with minimum use of hardware. This approach is not leading to a rapid adoption of true 3D graphics technology for the broadest number of users.

The goal of the Talisman initiative is to make 3D graphics truly ubiquitous. This requires that both high performance and high quality interactive rendering be made available at mass-market price points. This means that trading off image quality, as a means to obtain high performance rendering is unacceptable.

In this paper it will be shown that high quality rendering is a natural extension of the high-performance rendering architecture embodied in Talisman.

Introduction

Talisman [1] is the code name for a Microsoft technology initiative in research and deployment of advanced multimedia and 3D technologies for the PC industry. In essence the Talisman initiative is an effort to bring PC multimedia to the next level. The elements needed for the multimedia experience includes both high performance and high quality rendering. The goal is to provide technology to the PC industry to allow truly compelling content to be created. For this to occur, hardware must be produced that allows very complex scenes with antialiasing, and high quality texture mapping, that can be updated at the refresh rate of the monitor.

The fundamental issue that must be addressed in order to achieve high performance rendering is bandwidth. In traditional architectures this is addressed in two ways. First the evolution of silicon processing technology has allowed designs with greater clock speeds to be used. Simply re-implementing existing architectures in a newer generation of logic will increase the bandwidth of the system. The second way traditional systems address the bandwidth issue is to design wider buses for the graphics memory system. This approach is often used on high-end graphics devices. The technique increases the memory bandwidth, but can increase the system cost to the point where this approach is not useful for mass-market graphics devices.

It is important to note that over the years several unique architectures [2, 3, 4] have been proposed that address the memory bandwidth problem in various ways. However, these innovative architectures have typically been targeted for implementation in high-end devices rather than for mass-market devices

Addressing the Bandwidth problem

Talisman addresses the bandwidth issue with three techniques:

- Capture bandwidth on chip where it is affordable.
- Selective rendering.
- Aggressive use of compression.

Capturing bandwidth on chip:

Each image (represented by a ***DirectDraw surface***) is ***chunked*** into smaller pieces prior to rendering. In the reference design [1] each chunk was 32 X 32 pixels. Chunking is accomplished in the **DirectX** driver by subdividing the DirectDraw surface into chunk sized regions (i.e. 32 X 32 pixels). Then the triangles within the DirectDraw surface are sorted so that each triangle is processed with every chunk it touches. It is important to note that chunking is completely hidden from the application. However there is some overhead associated with the chunking. The big advantage is that chunking allows on-chip color buffers, on-chip Z-buffers, and an on-chip antialiasing engine. Having these units on-chip makes it possible to replace the external bandwidth requirements with on chip data accesses where the bandwidth is more affordable. It is interesting to note that other than the fact the memory is on chip, and it supports antialiasing, the Talisman rendering engine performs like a typical rendering engine.

Over the years, the cost per bit of DRAM has fallen dramatically (about 40% per year). Unfortunately this does not solve the DRAM bandwidth problem. Although the cost per bit of memory has dropped dramatically, the price of DRAM bandwidth has been improving at only about 12% per year. On the other hand the price performance of logic devices has been improving at about 42% per year. There are two implications of this; first, designs primarily based on exploiting the bandwidth to external DRAM's have note realized the full potential that I.C. technology can offer. Second, designs that continue to rely on bandwidth to external memory for performance will not keep pace with designs based on exploiting on chip bandwidth.

The chart below illustrates how quickly the relative performance of these two approaches is diverging. There are significant benefits to capturing bandwidth on-chip today. These advantages will become even more pronounced as time goes on and gate performance and density continue to grow dramatically, while bandwidth to external memory continues to grow relatively slowly.

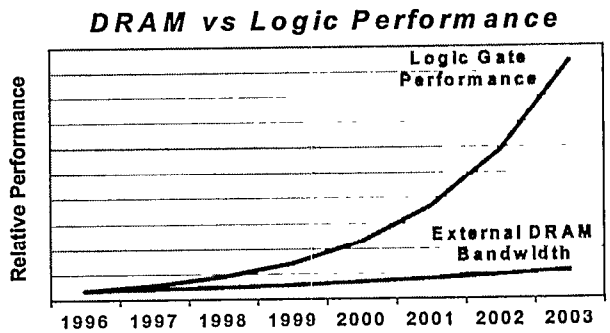


Figure 1: On chip vs. External Bandwidth

Selective rendering:

In smoothly moving interactive sequences most of the image is either the same or nearly the same from frame to frame. In a game, the characters may move around, but the background remains nearly the same between two frames. In fact, in the short time it takes to display two frames the main action characters may not have moved much. Even when the camera is moving, many of the pixels will have moved just a short distance from their last location.

In traditional graphics each frame is thrown away and the next one built from the beginning. It should be noted that there have been some very innovative people that have

seen that this is a waste of rendering resources and have written software that limits the regions that need to be re-rendered for new frames. However, hardware to exploit this image coherence does not exist today.

The graph below shows that for a scene from a typical interactive animation sequence that a relatively small percentage of the frame is new data.

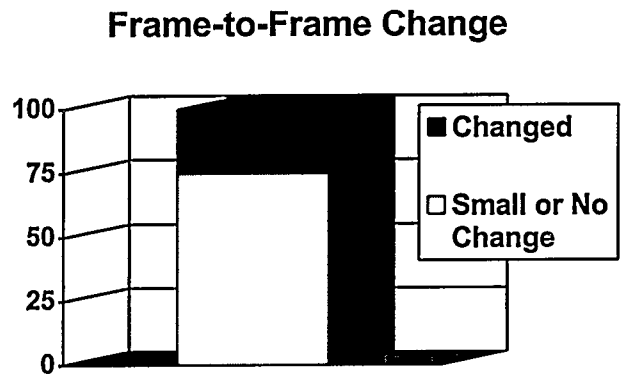


Figure 2: Frame to Frame Image Coherence

In order to use selective rendering a scene is broken into objects. Each object is then rendered as its own ***DirectDraw surface*** (called a sprite when stored in the graphics hardware system memory). There may be many of these surfaces in a scene. Each surface would typically contain objects that are spatially separated from other objects in the scene. For example: an airplane may be rendered to one surface and a mountain into another surface. As the airplane flies past the mountain, the mountain may never need to be re-rendered. There are two possibilities for handling the airplane. In one case the airplane may be spinning around and so the display surface that it uses may require re-rendering. In the other case the airplane may have such subtle changes that it could effectively be pushed across the sky. Applying an affine transformation to the display surface of the airplane does this pushing.

The general idea is to reuse as many display surfaces as possible for each frame. By reusing previously rendered parts of the scene, the rendering engine is freed to render more objects, or to apply higher quality rendering to the existing objects.

Aggressive use of compression:

Data transfers between chips require bandwidth. A major usage of this bandwidth in a traditional system involves reading and writing color and depth information to the frame buffer. Another major bandwidth use is transferring texture data from texture memory into the rendering engine. Compressed data requires less inter-chip bandwidth. In Talisman, most inter-chip data is passed in a compressed format. This can include both texture data and rendered images. This compressed data not only reduces bandwidth requirements; it also reduces storage requirements.

The Talisman Rendering Pipeline:

The details of the reference design can be found in the Siggraph paper in reference 1. Instead of repeating that information here, another way of looking at Talisman will be given. This view is a conceptual model of the Talisman graphics pipeline. There are seven main stages. These are:

1. The *application program*. The Talisman unique part of this stage is that the image is broken into display surfaces. Each surface contains an object that does not penetrate another object. These surfaces could be as complex as an entire frame or a single object within the frame. As an example, an entire object such as an airplane could occupy a unique surface. The other extreme is that independently moveable parts of an object, such as the flap on the

wing of an airplane, may be placed in its own surface.

2. The *geometry and setup* stage processes one surface at a time. The processing includes transformation, lighting, chunking, clipping, and polygon rasterization setup.
3. The *rendering engine* processes a chunk at a time. This is the stage where pixel level operations occur. These operations include color and depth interpolation, texture filtering, and antialiasing. There may be a data compression unit at the back end of the rendering block.
4. *Sprite data* for each of the display surfaces is stored, in compressed format, in a RAM array. This is envisioned to consist of off the shelf memory devices. Note that the sprite data includes newly rendered chunks from the rendering engine, along with older data that was rendered for a previous frame and can be reused in the current frame. Also note that in addition to the chunked data, compressed texture data is stored here as well.
5. The per-frame operation starts in the *sprite-image-processing* block. As each of the compressed chunks of image data is needed for display, it is pulled into the sprite-image-processing block. The data is first decompressed. Next the affine transformation is applied and an image filtering step is performed. Note that this filtering step is in addition to the texture filtering that is done earlier in the pipeline. The filtering step at this stage is required only to improve the quality of the affine transformation.
6. The data from the various display surfaces are assembled in the *compositing logic*. At this point the overlapping surfaces are resolved at each pixel. The data is then sent out in scan line order as typical RGB data.
7. The *display* is a typical device, such as a CRT or LCD.

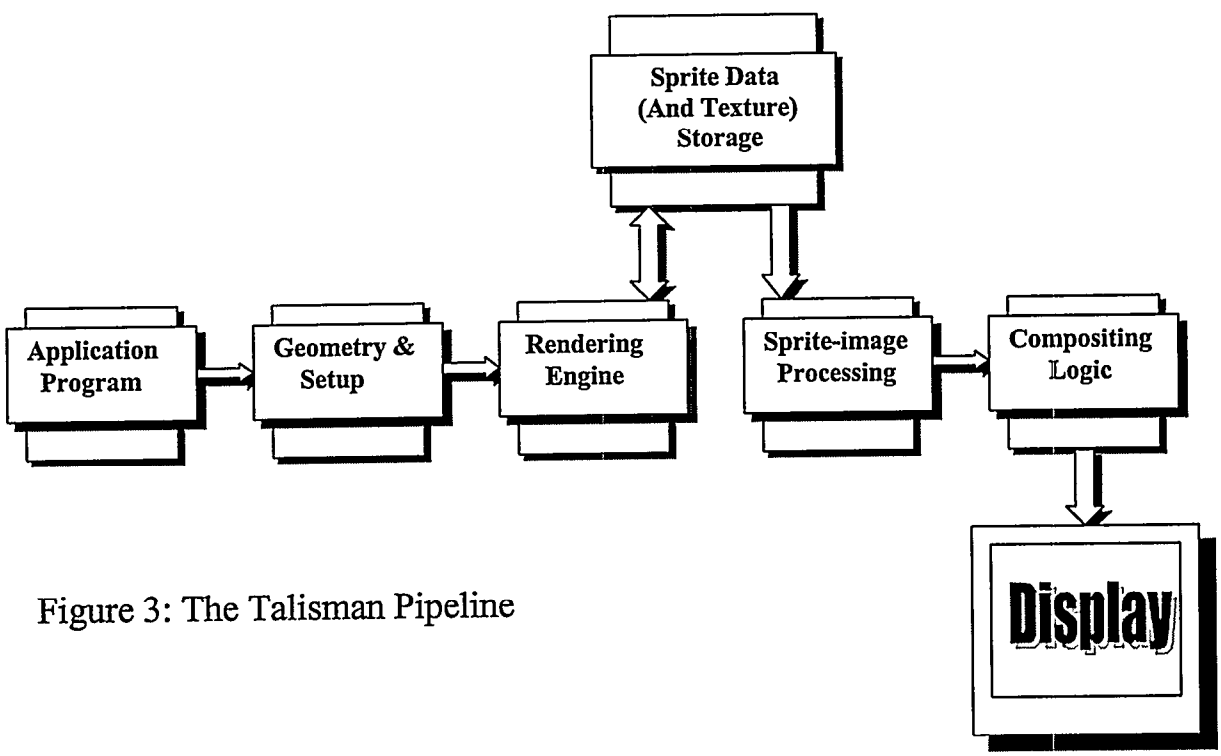


Figure 3: The Talisman Pipeline

Using the Talisman Pipeline to Produce High Quality Interactive Images:

The requirements of interactive image generation place a large demand on the rendering hardware. However, the architectures used in current mass-market graphics devices are severely limited in the amount of work they can perform to produce any single frame. Because of this limitation most hardware uses fast approximations [5] to render images. For example: typically polygons are sampled once per pixel as opposed to being sampled at multiple locations. The use of this approximation results in the well-known aliasing artifacts. In addition if texture filtering is performed, it is often done using a filter that does not account for the space-variant, elliptically shaped projection of the pixel back into the texture map. The use of these texture filtering approximations, such as the bilinear or trilinear method, results in noticeable blurring in the image. In addition there are other image quality features, such as reflections, complex shadowing and motion blur, that are so far beyond the capabilities of today's mass-market graphics hardware, that they are not approximated in real time.

Antialiasing

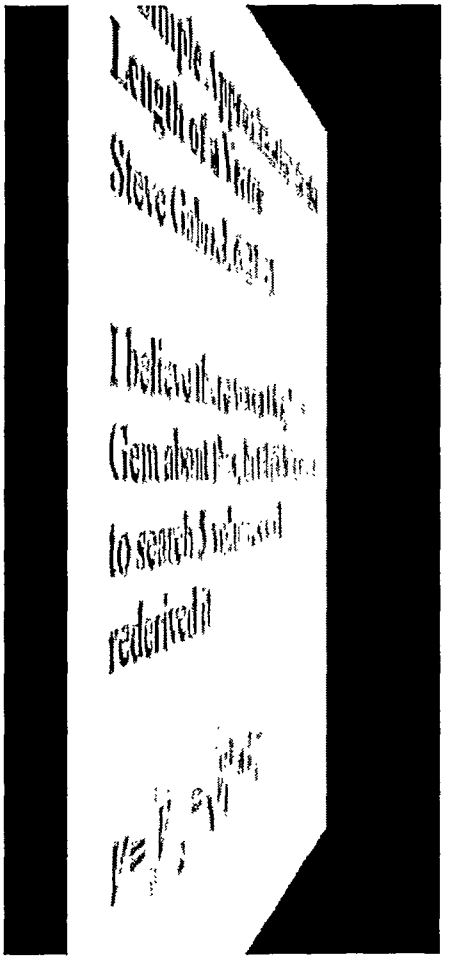
Perhaps the most noticeable artifacts in most interactive systems are the aliasing artifacts. This issue has been addressed in some of today's high-end graphics devices. The highest quality interactive antialiasing hardware available today is based on sampling each pixel within a polygon at several sub-pixel locations [6]. At each of these sub-pixel sample locations the color and "Z" information is found. The rendering works in the normal fashion, only with sub-pixel accuracy. After the frame is rendered the color samples from each sub-sample location are added together and divided by the number of sub-pixel sample locations. This produces the final color at each pixel. For example a polygon with an edge passing half way through a pixel that is sampled at 16 sub-pixel locations may contribute 8 of the sub-samples used to construct the final color. In this example the final pixel color will be a mix of one half the polygon color and one half the background color. The effect on the screen is a smooth antialiased edge. The problem with this method is one of bandwidth. Using this method with a traditional architecture, you need up to 16 times the frame buffer storage and it requires up to 16 times the number of accesses to the frame buffer and depth buffer memory. In high-end machines the cost of this extra memory and controller logic is reflected in the high cost of the machine.

The Talisman architecture uses a technique called ***chunking*** to reduce the amount of memory and off-chip bandwidth required to perform antialiasing. In talisman an image is rendered in small chunk regions, one at a time. This allows the entire region, including the color buffer, depth buffer, and antialiasing fragment buffers to be contained on chip. The primary benefit of this approach comes about because very fast and wide buses can be used to connect the on chip data path with the on chip memory. By capturing the bandwidth on chip, the bandwidth requirements for high quality antialiasing can be met.

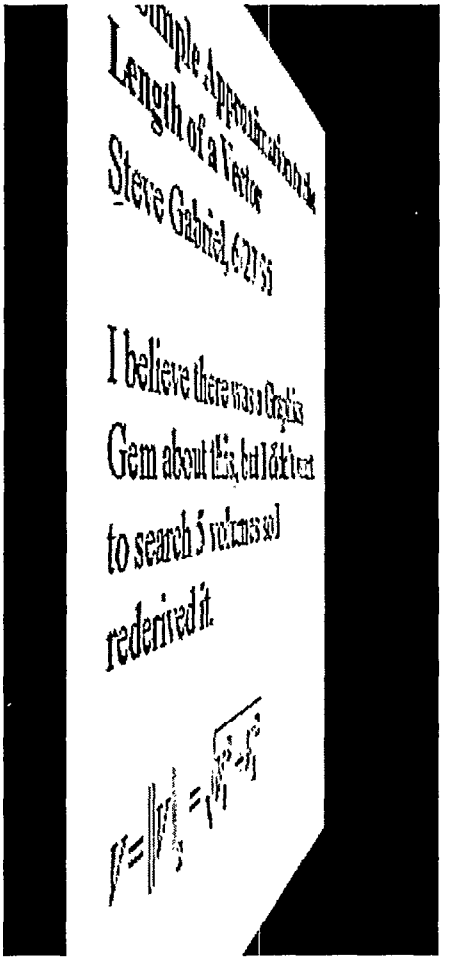
Since the driver bins all of the triangles for each chunk, the rendering engine can be thought of as containing a chunk of a typical frame buffer. The important point is that the rendering is done on chip and thus requires no external bandwidth. This means that various antialiasing techniques could benefit from the Talisman architecture. Thus various Talisman implementations may use different antialiasing algorithms. For example the reference design uses a variation of the Carpenter A-buffer algorithm [7] with the coverage mask idea similar to the one described by Schilling [8]. In addition the multi-sample algorithm could also benefit from the on chip memory accesses.

Anisotropic Texture Filtering:

Traditional texture filtering methods (bilinear and trilinear) produce blurred images when an object is tilted with respect to the view plane. This is because they rely on a round filter kernel. Unfortunately as an image is tilted the projection of the pixel on to the texture is no longer round, but elliptical in shape. Talisman accounts for this change in shape by computing the appropriate filter kernel while rendering. The result is that images produced using anisotropic filtering contain more detail and appear sharper than images produced using the other, more traditional methods. An example scene is shown below. The image on the left side of the page is rendered using traditional trilinear texture filtering. The image on the right hand side is shown rendered using anisotropic texture filtering.



Traditional Trilinear Filtering



Talisman Anisotropic Filtering

Figure 4: Visual Result When Applying Different Filtering Techniques to an Image

Anisotropic Filtering Algorithm:

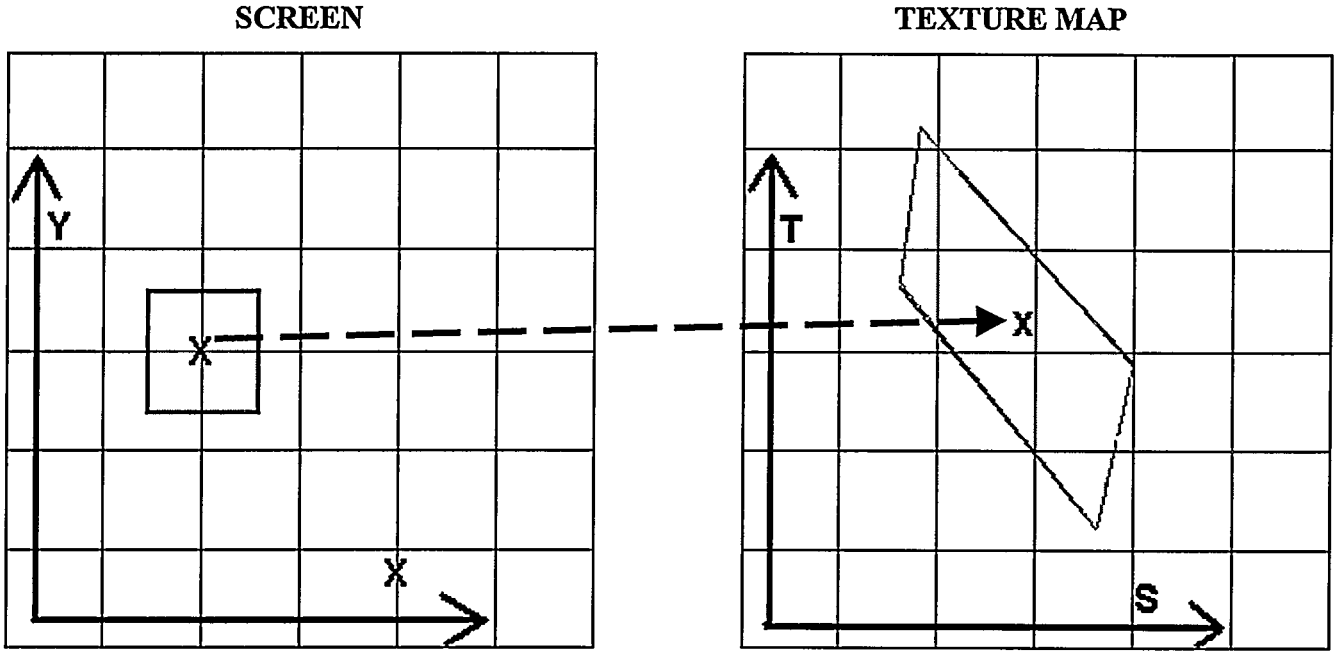


Figure 5: The Projection of a Screen Pixel into the Texture Map

Over the years there has been a great deal of research on how to best map texture data onto a pixel. For the most part analytical solutions have produced very high quality images at a large computational cost. These analytical approaches account for the actual projection of the screen pixel on to the texture map. For example the shaded regions in Figure 5 shows a screen pixel and it's projection back into the texture map.

The pixel projection onto the texture map can give rise to complex geometrical shapes. In hardware these complex shapes are often approximated. One of the most common approximations is Tri-linear MIP mapping. The essence of this approach is to pre-compute several filtered versions of the texture map. A square region that best approximates the projection of the screen pixel is then found. Next a square of four texels from the map that has more detail than the best fit square is blended with the set of four texels from the map level with less detail than the best fit square. This works well in regions where the projection of the screen pixel is nearly isotropic in the texture map. However as the projection changes to a more asymmetrical shape, such as shown in Figure 5, the image quality degrades. An example of the visual results of using the trilinear method was shown in Figure 4.

Note that hardware that uses better approximations for the projection of the screen pixel back into the texture map have been proposed [9, 11]. It is in this sprit that the algorithm used in the reference design was developed.

The hardware required for the anisotropic texturing algorithm used in the Talisman reference design is set up similar to hardware

used for traditional trilinear texture filtering. This includes using the same pre-filtered MIP map structure for storing the texture data. In addition both algorithms use the texture gradient information at each rasterized pixel in order to filter the texel data. These gradients are:

$$\partial s/\partial x \quad \partial s/\partial y$$

$$\partial t/\partial x \quad \partial t/\partial y$$

However, when performing anisotropic filtering the gradient information is used differently than in the traditional trilinear filtering system. These differences are explained using the geometry shown in Figure 6 below.

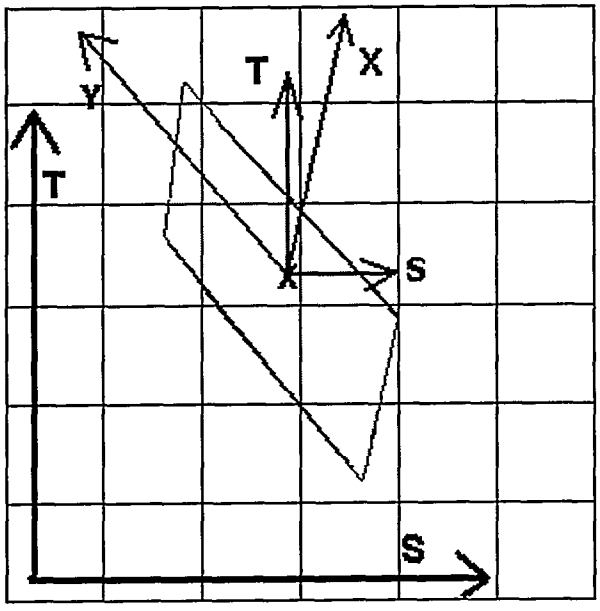


Figure 6: Geometry Used to Find Gradients

The long side of the rectangle is used to determine the orientation and line of anisotropy of the footprint. The shorter side is used to determine the level of detail. In addition the ratio of these two determines the amount of anisotropy.

The actual filtering is now a two step process. These steps are:

- Perform tri-linear interpolation in the MIP map for texture values along the line of anisotropy. Note that when using traditional tri-linear texture mapping hardware, the selection of which MIP levels to access is based on the largest gradient ($\partial s/\partial y$ in Figure 6). In this algorithm the selection is based on the smaller gradient ($\partial t/\partial x$ in Figure 6). This results in using texels from MIP levels with more detail.
- The values that are found by stepping along the line of anisotropy are then filtered. Note if the anisotropic ratio is 1 – to – 1, then the single value that is calculated by performing one tri-linear filtering operation is used as the texel value. If the ratio is between 1 – to – 1, up to 2 – to – 1 then two tri-linear filtering operations are performed and the resulting texel value is found by linear interpolation between the two tri-linear values. If the ratio is greater than 2 – to – 1 then a trapezoidal shaped reconstruction filter is used. In this case the two end points get less weight than the samples in the middle.

To view this process we will examine the texels being accessed in one level of the MIP map. Figure 7 shows the texture map. It is assumed that the texel values read by the hardware are shown at the intersections of the grid lines in the figure. In addition it is assumed that the anisotropy is about 2:1.

In this example the top four sample values read from the texture map are combined in proportion to the geometric location of the top reconstructed value. Note that the color values obtained for this reconstructed value will be the same as if this point were chosen for a single tri-linear filtering operation. However, we will also obtain a set of color data based on the geometric location of the lower reconstruction value. Note that these two reconstruction values lie along the line of anisotropy. The final step of the process is to

linearly interpolate between these two reconstruction values to the projection of the screen space pixel's center on the texture map. Note that at higher ratios of anisotropy additional trilinear filtering steps would be preformed.

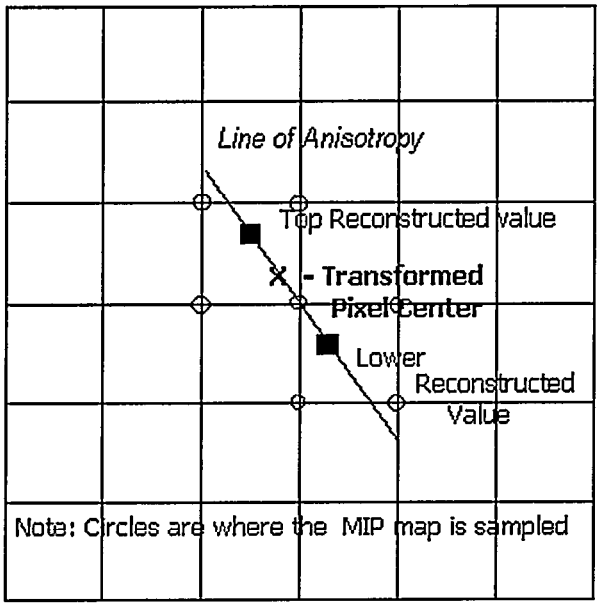


Figure 7: Reconstruction Use the MIP MAP

Fitting Into the Talisman Architecture:

The most striking thing about the implementation of the anisotropic texture filtering algorithm is that it requires more data to be read from the texture map. This apparent bottleneck is overcome in three ways by the Talisman architecture. First the aggressive use of compression. In Talisman the texture data is stored in a compressed format in the "sprite data and texture storage" block shown in Figure 3. This means that the extra bandwidth required for reading this extra data is significantly reduced from the requirements of a non-compressed texture. Second, there are data caches for the texture data. This is similar to most current texture mapping hardware in that recently used texel values are saved on chip. Third, a large amount of the bandwidth is captured on chip. Note that once a block of texels is decompressed, the entire block is stored in a decompressed format on chip. As an example the reference design called for storage for sixteen 8 X 8 blocks of decompressed texture data. In addition to the large cache of data, there is high speed accesses to that data over on chip data buses. In the reference design texture filtering with anisotropic ratios of 2:1 could be performed at the full rasterization rate.

Other Quality Features:

Other quality features such as reflections and shadowing are made possible by the unified sprite and texture storage.

For example: to produce a reflection on a surface the scene is rendered using a view-port that is *looking* at the scene through the object that the reflection is to appear on. Next this rendered image is used as a texture map that is placed on the object as the scene is being rendered from the desired view-port. The effect is that the scene appears to be reflected off of the object. The most difficult part of this process for most hardware is to move the rendered image into the texture storage space. In Talisman the unified memory, shown as the "sprite data and texture storage" block in Figure 3, simplifies this process. In this example the data is rendered into the memory on the first pass. On the second rendering pass it is read as texture data from the memory. Note that in Talisman the texture data and sprite data are stored in the same format.

Just as reflections can be produced, so can other multi-pass rendering effects [10], such as shadows.

Complex Scenes:

One item that is often over looked as a metric of image quality is scene complexity. The highest visual quality often times requires the greatest scene complexity. In Talisman the sprite data that is stored in the memory array can often be used for many frames. In order to make this reuse possible the "sprite image processing block", shown in Figure 3, contains

logic to perform affine transformations on the sprite data.

The idea is to use high quality rendering to produce each sprite. Then as subsequent frames are rendered a test can be performed to determine how much the objects have changed since the sprite was last rendered. If it is determined that the objects in the sprite have not changed significantly, then apply an affine transformation to the sprite and reuse it in the current frame.

Note that there will be an affine library that will accompany **DirectX**. This library will aid applications by finding the best affine transformation to apply to a sprite. In addition it will return error terms that characterize the distortion and color errors that would result from using the affine transformation, instead of re-rendering the object. The application can then use these error terms to balance reuse with re-rendering for the various objects in a scene. The result of using this process is that the rendering engine's effective pixel production rate is multiplied by the amount of reuse that is possible. This allows high quality, complex scenes to be created that can not currently be produced using mass-market graphics devices.

Conclusion

The three primary techniques used in the Talisman architecture to address the memory bandwidth issues can be exploited to improve image quality. Using the Talisman architecture allows unprecedented levels of performance and image quality to be made available at the price point needed for mass-market graphics device. The goal is to bring these unprecedented levels of performance and quality to the broadest number of users.

1997 SIGGRAPH/Eurographics Workshop

References

1. Torborg, J., Kajiya, J., "Talisman: Commodity Realtime 3D Graphics for the PC", *Proceedings of SIGGRAPH '96*
2. Deering, M., et al. "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *Proceedings of SIGGRAPH '88*
3. Fuchs, H. , et al. "Pixel Planes 5: A Heterogenous Graphics System Using Processor-Enhanced Memories", *Proceedings of SIGGRAPH '89*
4. Molnar, S., et al. "PixelFlow: High Speed Rendering Using Image Composition", *Proceedings of SIGGRAPH '90*
5. Barkans, A., "Hardware-Assited Polygon Antialiasing" *IEEE Computer Graphics and Applications* Vol 11 Number 1 Jan. 91
6. Akeley, K., "RealityEngine Graphics", *Proceedings of SIGGRAPH 93*
7. Carpenter, L., "The A-Buffer, an Anti-Aliased Hidden Surface Method", *Proceedings of SIGGRAPH 84*
8. Schilling, A., "A New Simple and Efficient Anti-aliasing with Subpixel Masks", *Proceedings of SIGGRAPH 91*
9. Knittel, G., et al. "Hardware for Superior Texture Performance" Proceedings of *Eurographics Workshop on Graphics Hardware 95* ISSN 1024-0861
10. Segal, M., et al. "Fast Shadows and Lighting Effects using Texture Mapping", *Proceedings of SIGGRAPH 92*
11. Schilling, A., Knittel, G. and Strasser, W. "Texram: Smart Memory for Texturing", *IEEE Computer Graphics and Applications* Vol 16 Number 3 May 96