# RADEON WHITEPAPER

## Table of Contents

## Disclaimer and Copyright

## Executive Summary

With its new RADEON graphics processor, ATI is introducing a host of new technologies that make possible the goal of photorealistic 3D graphics generated in real time on a consumer PC. The main applications that are pushing technology forward in the area of 3D graphics are games. Each of ATI's new technologies contains several unique features that allow application developers to overcome the existing barriers to truly immersive 3D graphics.

The Charisma Engine provides features that make 3D characters and environments look and behave believably. It includes high performance, versatile support in hardware for transformation, clipping, and lighting (TCL) calculations. This reduces CPU workload and allows 3D scenes to contain more polygons and more complex lighting than any competing technology. The Charisma Engine is also the only available engine that accelerates advanced features like 4-matrix vertex skinning and keyframe interpolation.

Pixel Tapestry architecture includes features that add detail to an image, such as reflections and shadows, without compromising performance. This unique architecture supports a wide array of versatile features like single pass multitexturing with three textures, 3D textures, bump mapping (emboss, dot product 3 and environment mapped), texture transformations, priority buffer support, shadow mapping, and range-based fog. Since these features can be enabled without impacting frame rates, they can be used more extensively to increase the realism of 3D scenes.

HyperZ Technology incorporates innovations that make more efficient use of memory bandwidth. As graphics chips take over more of the processing duties formerly handled by the CPU, use more advanced features, and work with larger textures and complex scenes, memory bandwidth has emerged as the key performance-limiting bottleneck. HyperZ focuses on optimizing Z-buffer reads & writes, which typically consume more than half of all the memory bandwidth required by a 3D application. The result is a major boost in 3D performance, which translates into a smoother, more enjoyable experience for the user.

## State of the Industry

The insatiable demands of PC users for applications that do more, look better, and run faster ensure that there continues to be a large market for powerful 3D graphics processors with rapidly increasing performance. 3D graphics capability has become so important lately that it has caused the speed and complexity of consumer 3D graphics processors to exceed that of the CPU, which has traditionally been considered the most complicated and advanced component of a PC. Graphics accelerators now have more transistors and higher floating point operation performance than the fastest available CPU, despite selling for a fraction of the cost. The end result is graphics that approach, and in some ways exceed, the quality found on workstations costing tens of thousands of dollars… available at consumer prices of a few hundred dollars.

Considering that 3D computer games have become the leading drivers of graphics technology, what can consumers expect? Well, most game developers would agree that their goal is to create a product that looks as good as a big-budget, special effects-laden Hollywood movie, but is interactive. That is, users control the action and have as much freedom as the designer wishes them to have. Ideally the designer should be able to "suspend the disbelief" of the user, so they can temporarily believe they are in another place. This has not quite been possible to date, but with the introduction of ATI's new Charisma Engine, Pixel Tapestry, and HyperZ technologies, that will soon change.

## Barriers to Immersive, Real-time 3D Graphics

3D computer graphics have certainly come a long way since their first appearance on the desktop PC. There is no question that the graphics in recent 3D games like Quake 3 and Unreal come much closer to approximating the real world than older games like Doom. However, even when combining today's most advanced 3D technologies with top quality artwork and animation, it is quite clear when looking at a moving image that one is not looking through a window into a living, breathing world. With currently available hardware and software technology, it has become possible to create moving images that appear nearly photorealistic. However, this is only practical for pre-rendered 3D sequences. An interactive application requires a 3D scene to be rendered in real-time, which requires a tremendous amount of computational power.

There are three main barriers that prevent the current generation of PC graphics hardware from rendering a photorealistic 3D environment in real-time:

### Lifelike Characters

The 3D characters in today's games suffer from the following problems:

- Blocky, unnatural appearance (especially when viewed up close)
- Stiff, robotic movements
- Short, repetitive animations without variety
- Lack of visible emotions or facial expressions
- Too few character models (all characters have the same shape)
- Limited number of characters on screen at any given time

### Detailed Objects

The various objects and items that populate a 3D scene have the following limitations:

- Low complexity (only simple shapes and forms can be represented)
- Lack of texture detail when viewed up close (appear flat and smooth)
- Often fixed and immovable (to avoid having to represent all sides and parts of the object)
- Reflections are absent, poorly detailed, or do not reflect changes in the environment
- Do not cast dynamic shadows (must be pre-rendered or not rendered at all)

### Lush Environments

The 3D environments that characters move around in and interact with have the following issues:

- Simple architecture with few curves and fine details
- Static, cannot be changed or deformed in real time
- Do not respond to user's actions
- Use 2D representations of 3D effects like rippling waves or billowing clouds

While today's graphics chips can do a lot of things to make characters, objects, and environments more realistic and detailed, they suffer unacceptable losses in performance that offset these benefits. The result is that the creativity of game developers is being constrained by graphics technology. Developers are forced to spend a great deal of time and effort looking for creative ways to work around hardware limitations, which could be better used creating compelling content. ATI's RADEON graphics processor is set to finally overcome these barriers without compromising performance, using a host of new technologies.

## Realistic Characters and Environments

The heart and soul of any interactive 3D game is the characters you encounter. Characters are dynamic, expressive, and unique entities that bring a game to life, whether your goal is to talk to them, admire them, or blow them to smithereens. It's no coincidence that most game developers put such a great deal of time and effort into designing, modeling, and animating 3D characters.

Ideally, we want the characters in games to look, move, and behave as we would expect them to in real life, whether they are human beings or fantastic creatures. Take one look at a typical 3D game today, however, and it will be immediately obvious that this goal has yet to be achieved. Most characters have a blocky or chunky appearance that only gets worse as you get closer to them. Their motions are mechanical and repetitive, more like robots or cartoon characters than like living things. The expressions on their faces flicker suddenly from emotion to emotion, if they change at all.

Characters aren't the only things lacking realism in today's 3D games. Consider the environments they are moving in. In most cases, the environments appear even more blocky and angular than the characters. While the first games are starting to appear that use curved surfaces in the environment, designers are still restricted as to how many curves they can use, and how smooth they can be. Moreover, the environments are mostly static. Many games use animated textures and other tricks to give the illusion of moving liquids and fabrics, but in reality these objects are not actually moving or changing shape.

These problems all arise from the fact that current graphics technology can only approximate reality, not emulate it. Game developers are forced to spend time coming up with clever tricks to bypass these limitations. The Charisma Engine makes it easier than ever to produce believable 3D characters and immersive environments.

## The CHARISMA ENGINE™

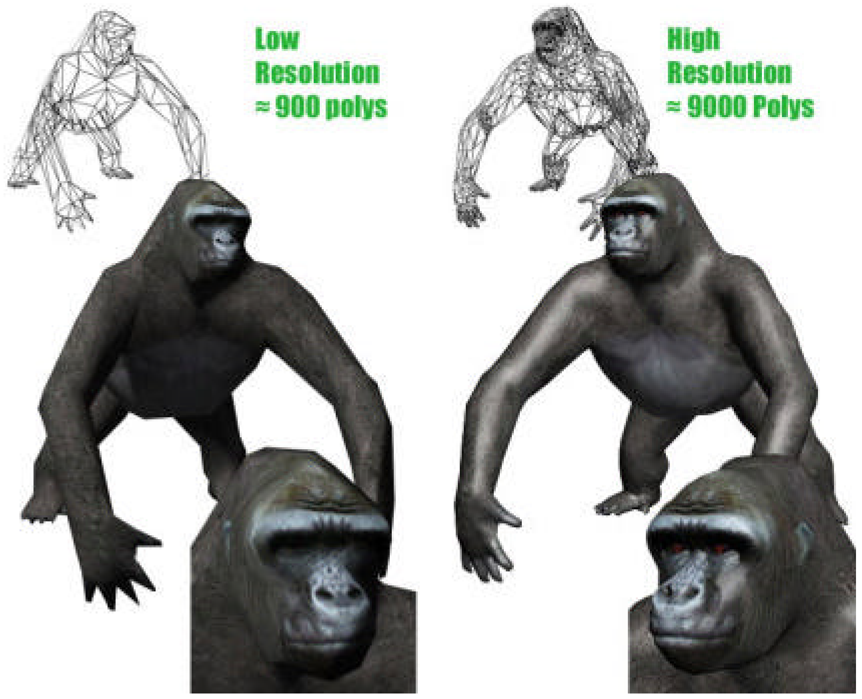
The Charisma Engine takes 3D characters and environments to the next level of realism. It can perform complex transformation, clipping, and lighting calculations faster than any CPU or 3D graphics chip on the market. It also adds innovative and useful new features such as 4-matrix vertex skinning and keyframe interpolation that give game developers more freedom and flexibility than ever before.

### Fast, Flexible Hardware Transformation, Clipping & Lighting (TCL)

A graphics processor is a highly specialized device that is optimized for the purpose of creating graphics and outputting them to a display device. A CPU, by contrast, must be capable of handling a much wider range of generalized applications and calculations. It can be said that the CPU is a jack-of-all-trades, but master of none. Even as they get faster and more efficient, and add special "multimedia instructions" that are designed with graphical applications in mind, standard CPUs still cannot match the speed and efficiency of dedicated, application-specific processors. Thus, in the quest for higher graphics performance, it is inevitable that specialized graphics processors take over more and more of the graphics-related workload from CPUs.

There are other benefits that accrue from transferring work from the CPU to the graphics processor. In most graphical applications, graphics-related calculations can account for a large percentage of the CPU's workload. With these calculations offloaded, the CPU can use the available time to perform more calculations for other uses. These can include physics, artificial intelligence, collision detection, etc. Also, because the graphical calculations are done in hardware, they need only be dealt with at a high level through an application interface like Direct3D® or OpenGL®. This saves the developer the time and effort required to write and optimize code to process the calculations, meaning they can devote more of their attention to other aspects of the game.

The last major revolution in the computer graphics industry occurred when 3D graphics functions started getting accelerated by the graphics processor. This enabled the first true 3D games to start appearing, which provided a quantum leap in visual quality compared to earlier 2D and "pseudo-3D" games. The revolution going on today involves the transformation, clipping, and lighting calculations required to render any 3D scene. The number of calculations that need to be done in these three steps is directly related to the number of polygons present in the scene. The Charisma Engine not only accelerates all of these functions at speeds much faster than any existing CPU, but it accelerates more features than any existing hardware T&L engine. This means 3D scenes and characters can contain many more polygons, which will result in the next big leap in visual quality. Look at the characters in the illustration below:



(Low vs. High Polygon Characters)

The character on the left is composed of about 900 polygons, which puts it on the high end of polygon count for today's latest games. The character on the right is composed of approximately 10 times as many polygons as the one on the left. Notice how much smoother it appears. The silhouette of the character on the right is rounded naturally, while the outline of the character on the left is composed of straight lines and angles. The fingers, toes, and facial features of the character on the right look much more detailed and realistic. Even though both characters use the same vertex lighting technique, the lighting looks much better on the character on the right. This illustrates the advantages of using vertex lighting (which is hardware accelerated) on character models with more polygons.

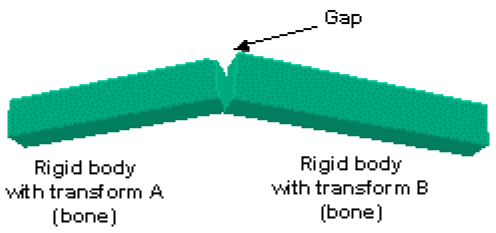
Fast polygon processing and lighting calculations don't just improve character quality. Today's games tend to use a limited number of polygons to represent environmental features (structures, items, landforms, etc.) since they are usually in the background relative to the 3D characters. The result is objects that are angular and simplistic. With high-speed transform and lighting, objects can have more complex shapes and curves. Small objects like crates, stones or shrubs can be modeled in 3D rather than drawn into the background, allowing them to be moved and manipulated. In short, environments can become much more realistic, dynamic, and interactive.

While increasing the number of polygons and lights that can be included in a scene has a major impact on visual quality, it is not quite enough to truly bring characters to life. Character animation is equally important in this regard, since it defines how a realistic a character's movements and behaviors appear. The Charisma Engine includes accelerated support for unique features that take character animation to the next level. These features and their benefits are described below.

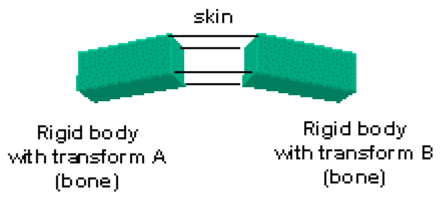
**Skeletal Animation and Skinning**

Most 3D characters today are modeled using a group of attached polygons, sometimes referred to as a mesh of vertices. To change the "pose" of a character, the positions of the vertices in the model must be changed and stored, creating a frame of animation. A character can then be made to move by quickly cycling through several frames of animation. This works fairly well, but many 3D game developers are starting to use an exciting new technique known as skeletal animation. This involves defining character models using a series of "bones", to which is attached a "skin" consisting of a mesh of vertices or polygons. The position of each vertex can be influenced by the movement of one or more bones, according to a given weighting value. Thus, a character model can be animated by simply defining the movement of its skeleton, and the movements of the vertices that define the skin can be generated mathematically. Skeletal animation not only simplifies the animation process, but also requires far less memory than traditional animation methods. This is because the position of every vertex in the model does not need to be stored for every frame of animation. Only the initial vertex positions and weights need to be stored, as well as the movement of the bones for each frame. The extra memory freed up using this technique can be used to make animations longer and more detailed, or to include a wider variety of character models and animations.

One weakness of skeletal animation is the way it handles joints between bones. Each bone is rigid, and its movement is defined by a transform. If the transforms cause the joint to bend, an unsightly gap can be created.


Gap
Rigid body with transform A (bone)
Rigid body with transform B (bone)

This weakness is overcome using a technique called skinning, which adjusts and blends the positions of the vertices around the joint to create a continuous, flexible skin.

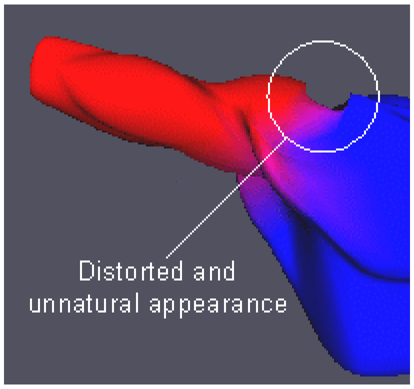

skin
Rigid body with transform A (bone)
Rigid body with transform B (bone)

*Vertex Skinning With Up To 4 Matrices*

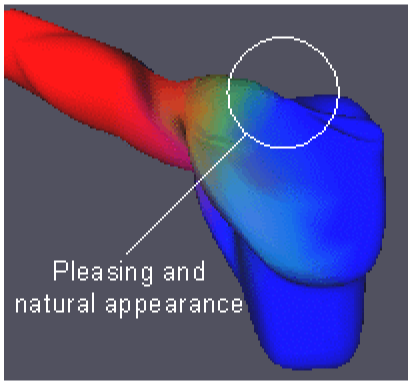
In order to compute the location of each vertex in each frame of a skeletal animation, matrix transformations are required. Matrix transformations are mathematical functions that are used to calculate the position of an object in 3D space. A separate matrix transformation is required for each bone that influences a given vertex. The weightings of each matrix can vary for each vertex, which is important for vertices located near the joints between bones. In order to make joints that flex naturally, the matrix weightings for each vertex around a joint must blend gradually from one bone to another. This technique is called vertex skinning.

The matrix transformations required for vertex skinning are very computationally intensive, and the complexity increases with each additional matrix used. The powerful Charisma Engine can accelerate vertex skinning with up to 4 matrices in hardware, allowing 3D animations that are more complex and believable than has ever been seen before - without requiring any extra CPU work.

The illustrations below demonstrate the quality improvements that can be realized using additional transformation matrices for vertex blending. Each color represents the weightings of different matrices (red/blue for the 2-matrix case, red/blue/green for the 3-matrix case). Notice how the weighting of each matrix blends across the shoulder joint.



2-Matrix Blending



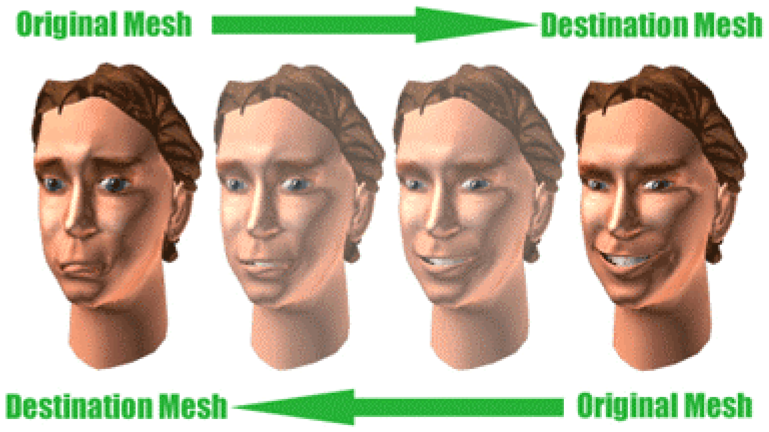
3-Matrix Blending

The 3-matrix example above uses the extra (green) matrix to control the "bulge" of the joint, making it look more realistic. Extra matrices can also allow accurate modeling of more complex joints involving more than two bones. Existing transform & lighting engines that support only 2-matrix blending cannot accelerate these types of animations. The only way they can achieve a comparable effect is to use an entirely software-based solution, which kills performance.

## *Keyframe Interpolation (Vertex Morphing)*

Even with skeletal animation techniques, animating a 3D model can be a difficult and time-consuming process. The smoother the animation is to be, the more frames must be created, and animations with high frame counts can quickly gobble up large amounts of memory. This is what prevents the character animations in today's games from being as smooth and lifelike as they could be. In addition, certain complex animations, such as realistic facial expressions or rippling muscles, can be difficult to model accurately using bones, even with 4-matrix transformations. Yet these kinds of subtle animations are what truly make a 3D model come alive.

This is where the Charisma Engine's keyframe interpolation feature comes into play, providing a simple yet powerful method of overcoming these issues. It works by interpolating between successive frames of animation (called keyframes), automatically generating new frames to fill in the gaps. The positions of the model vertices in the first keyframe are seamlessly morphed into their positions in the second keyframe. The number of interpolated frames can be scaled as necessary, with more frames providing smoother animation and less frames providing better performance. Using the example of a face, say you want to take a frowning face and make it smile. With keyframe interpolation, all you would need is two keyframes of animation: one with a vertex mesh defining a frowning face, and one defining a smiling face. The Charisma Engine could then smoothly morph one into the other, creating as many additional frames as desired, without any additional CPU overhead (see illustration below). The incredible effect of this feature is impossible to capture in still screen shots… it must be seen in motion to be fully appreciated!



*(Keyframe Interpolation)*

Keyframe interpolation makes character animations scalable, meaning that their level of detail can be adjusted to trade off speed vs. quality. For example, if a game using this technique began to run too slowly and fell below a certain frame rate threshold, the application could detect this and compensate by reducing the number of frames of animation used. In addition to adding scalability to character animations, keyframe interpolation makes them less time-consuming to generate, and reduces their memory storage requirements. The Charisma Engine is the first technology to provide hardware acceleration for keyframe interpolation.

## Visual Details - Getting the Most Out of Every Pixel

They say the devil is in the details. This is definitely the case when creating realistic 3D objects and environments. There are so many small, subtle things that can betray an object's artificial nature. The way bright light glances off a shiny surface. The way your face is reflected in a brass doorknob. The subtle shadows cast by every object from every light source. The fine, detailed textures of wood, cloth and stone. These details might seem insignificant, but they are the kinds of things that you only notice when they aren't there.

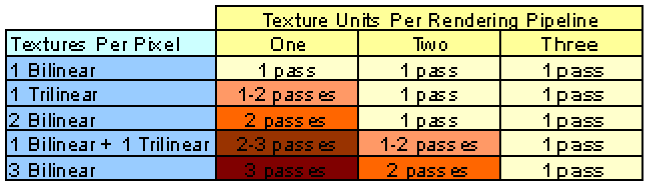
3D games are gradually increasing the level of graphical detail they support. The amount of processing done on each pixel in a scene has been rising significantly. Unfortunately, the performance sacrifices that need to be made to achieve these high detail levels have limited their practicality. ATI's unique Pixel Tapestry architecture makes it easy to include unprecedented levels of detail on 3D objects and environments, without trading away performance.

### PIXEL TAPESTRY™ architecture

ATI's next-generation graphics products will utilize the most advanced and flexible texturing system available. Pixel Tapestry is the first and only graphics architecture to include three independent texture units in each rendering pipeline. This unique architecture opens up a whole range of new possibilities for making 3D surfaces look more realistic and detailed than ever before.

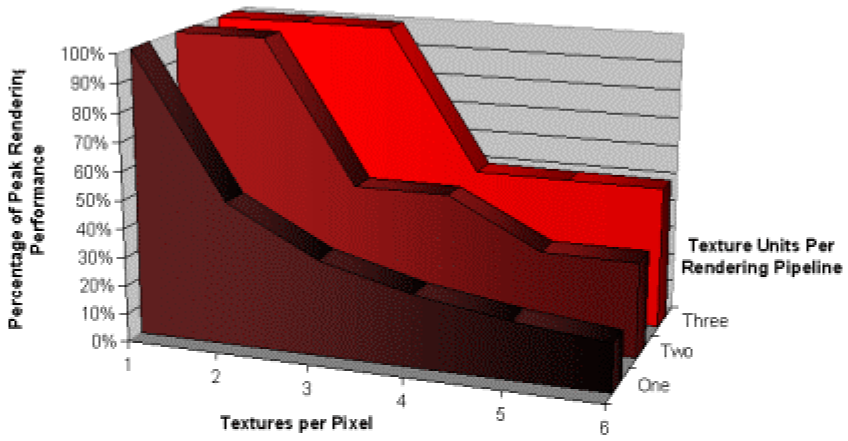
### Three Filtered Textures Per Pixel - At Full Speed

Current generation graphics processors have just one or two texture units per rendering pipeline. They are capable of applying up to two textures to a pixel in a single clock cycle, but with one texture unit per pipeline this requires using two rendering pipelines to generate a single pixel. Thus, the fill rate of these products is effectively cut in half when multitexturing with two textures per pixel. In order to add a third texture to a pixel, these chips must use a second clock cycle, which cuts the fill rate in half again. Thus, it takes most current generation processors four times as long to apply three textures to a pixel than it does to apply a single texture. Even those that have two texture units per pipeline take twice as long to apply three textures per pixel. Thus, the number of rendering passes required to output a single pixel is becoming increasingly critical to providing acceptable graphics performance.



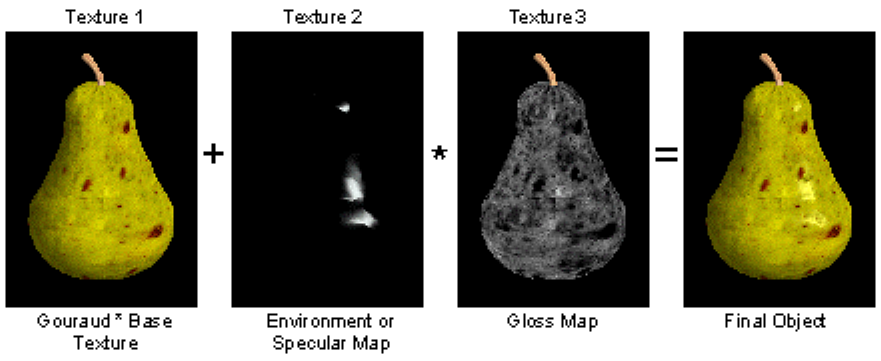
| | Texture Units Per Rendering Pipeline | | |
|---|---|---|---|
| Textures Per Pixel | One | Two | Three |
| 1 Bilinear | 1 pass | 1 pass | 1 pass |
| 1 Trilinear | 1-2 passes | 1 pass | 1 pass |
| 2 Bilinear | 2 passes | 1 pass | 1 pass |
| 1 Bilinear + 1 Trilinear | 2-3 passes | 1-2 passes | 1 pass |
| 3 Bilinear | 3 passes | 2 passes | 1 pass |

One of the most talked about factors for determining graphics performance today is "peak fill rate". This refers to the maximum number of pixels a graphics product can render to the screen per second. However, peak fill rate is calculated with the assumption that each pixel has only a single, unfiltered texture applied. In reality, 3D games today make heavy use of filtering and multitexturing, and future games will use these techniques even more extensively. The result is that graphics architectures that do not fully take multitexturing requirements into account will suffer rapidly decreasing performance as more textures are applied to each pixel. As the following chart illustrates, including more texture units per rendering pipeline allows a graphics chip to maintain high performance even with large numbers of textures per pixel.



This already sounds bad, but memory bandwidth issues compound the problem even further. Every time an extra texture is applied to a pixel, the pixel must first be written to the frame buffer, and then read back to be blended with the new texture. This consumes large amounts of precious memory bandwidth, which could better be used in preparation to draw the next pixel in the pipeline.

The primary benefit of Pixel Tapestry is that it allows up to three filtered textures to be applied to every pixel in a scene with virtually no performance drop. With three texture units per rendering pipeline, additional textures can be applied to a pixel without requiring additional clock cycles, tying up a second rendering pipeline, or wasting memory bandwidth on unnecessary frame buffer reads & writes.

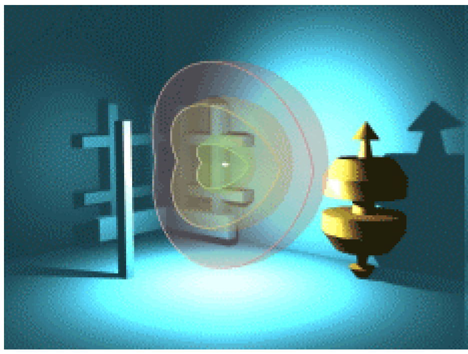


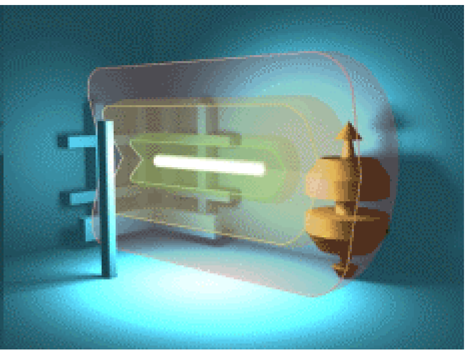
Most recent 3D games already make extensive use of multitexturing. In addition to the base texture on an object, applications can add light maps, shadow maps, reflections, bump maps, specular maps, detail textures, and more. On current high-performance graphics processors, every additional texture added to a surface beyond the base texture doubles the rendering time. Each additional texture adds another level of realism and detail to the surface, but adding more than one or two will bring the performance of even the fastest chip to a crawl. With Pixel Tapestry, however, you get two additional textures without a significant impact on performance.

**3D Textures**

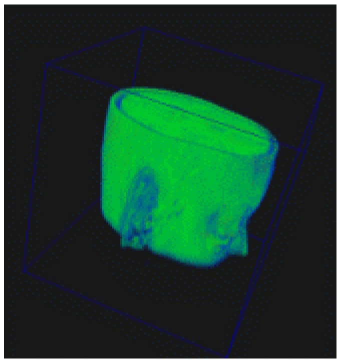
3D texture support is an exciting new feature of ATI's Pixel Tapestry architecture. Although polygons are traditionally used to represent 3D objects, each polygon is actually a 2D surface. Thus, all existing mainstream graphics hardware to date has used 2D textures to apply to these 2D surfaces. These 2D textures are like stickers applied to a polygon's surface. A 3D texture is a volume of texels just like a 2D texture is a plane or "sheet" of texels. Even though it "occupies" volume in 3D space, it is still a texture and therefore is only visible where it intersects polygon surfaces.

3D textures have a wide variety of applications that add to the realism of a 3D scene. One example is to use a transparent or translucent sphere-shaped texture to define a "globe" of light around a light source. The texture would be brightest in the center of the sphere and dimmer toward the edges. Any polygon surfaces that intersect the sphere, including those making up walls, objects or characters, would be appropriately lit according to how far away from the light source they are. This is a far more simple, elegant, and flexible method of modeling dynamic light sources than using light maps as many present-day applications. It makes it easy to vary the texture in real time, to make flickering or pulsing light sources. It also allows accurate modeling of light sources with extent (i.e. non-point light sources). The illustrations below demonstrate how a cylindrical light source modeled with 3D textures lights a scene differently from a point light source (note the softer shadows cast by objects in the image on the right):


**Point Light Source with 3D Textures**


**Cylindrical Light Source with 3D Textures**

3D textures can also be used to great effect with dynamic or procedurally generated geometry. Imagine, for example, a cube of marble. The cube could be defined with a simple geometric model, and have a cube shaped 3D texture applied. If an application then altered the cube model to take chunk out of one side, the 3D texture would allow the marble veins running through the cube to be clearly visible. With traditional 2D textures, new textures would have to be generated or pre-defined to achieve a similar effect. The illustration below shows how this technique can be used for medical imaging:



**3D Texture Used For Volume Visualization of MRI Data of a Human Head**

The examples above just scratch the surface of the kinds of benefits that can be realized with 3D textures. Game developers will be certain to come up many more inventive uses for this capability. 3D textures are already supported in OpenGL® 1.2, and will likely be supported in future versions of DirectX®.

## Bump Mapping

Bump mapping refers to a range of techniques that use multitexturing to create the illusion of three-dimensional detail on a surface, without changing the geometry of the object or requiring additional polygons. There are three methods commonly used to accomplish this effect, each of which is more useful in some applications and less useful in others. While existing graphics architectures support only one or two of these methods, ATI's Pixel Tapestry architecture is the first to support all three. This gives developers the ability to use whichever method is most appropriate for their particular application.

### *Emboss*

The simplest and easiest, but also least realistic method is called emboss bump mapping. This effect is accomplished using a height map texture, in which the "color" of each texel is used to represent bump height. This texture is shifted one or more pixels away from the light source, then subtracted from the original un-shifted texture to produce a bump map. The bump map is then combined with the base texture to produce the final effect. Emboss bump mapping is the easiest to implement because it does not require that special texture formats be used like other bump mapping methods.

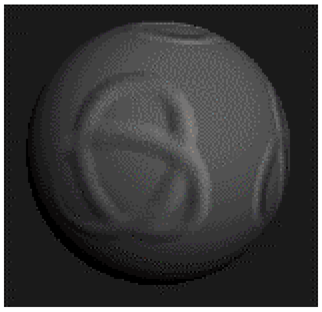
*Dot Product 3*

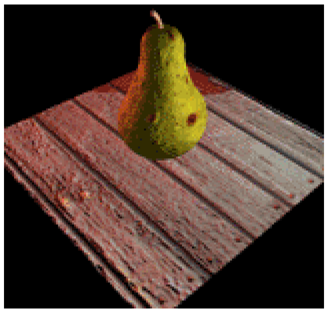
A more accurate and flexible version of bump mapping is Dot Product 3. It works by using a specially formatted bump map texture. Each pixel in the texture includes a 3D vector that represents the slope of the surface. A vector is then generated pointing from each pixel in the bump map to the light source, and a dot product operation is performed between this vector and the vector contained in the bump map to determine which direction light will be reflected off of the surface. This method allows control of the bump map down to the individual pixel level, which can give the appearance of a highly detailed surface when light is applied.

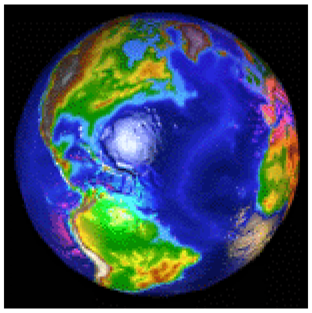
*Environment Mapped Bump Mapping (EMBM)*

The most detailed and flexible variation of bump mapping is Environment Mapped Bump Mapping (also referred to as EMBM or Perturbation Bump Mapping). Emboss and Dot Product 3 bump mapping are suitable for most matte objects, but they do not work well on glossy, shiny, or reflective surfaces like metals, plastics, or liquids. They also lack the flexibility to change the height map in real-time, which can be used to create effects like rippling waves. EMBM adds these capabilities at the expense of additional computational and texture requirements. Like Dot Product 3 bump mapping, EMBM requires the use of specially formatted textures as bump maps, and also requires a separate environment map to provide the reflections. The bump map is used to "perturb" the environment map before blending it with the base texture, and make it appear both shiny and bumpy.

A comparison of the three types of bump mapping supported by Pixel Tapestry is shown below:


Emboss Bump Mapping


Dot Product 3 Bump Mapping


Environment Bump Mapping

## Texture Transformations

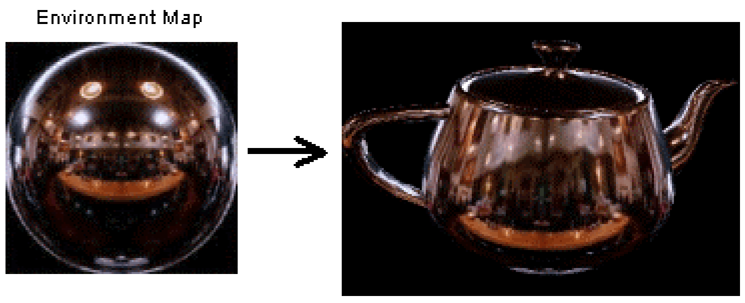
Creating detailed textures is only part of the process required to bring objects to life. There are also many different ways to apply a texture to a surface, each of which can be used to achieve a specific effect. This group of effects is accomplished using texture transformations. These transformations involve applying various mathematical formulae to generate texture coordinates. All of the techniques listed below can be accomplished using software, but they involve complex calculations that devour CPU time and hurt performance. As a result, most current games use them sparingly, if at all. ATI's Pixel Tapestry architecture changes all that by implementing hardware acceleration for texture transformations. With Pixel Tapestry, widespread use of these features becomes practical - and it looks great!

**Environment Mapping**

Environment mapping involves the use of textures that mirror the environment. These textures can then be transformed and applied to any surface to give accurate reflections. There are several variations of this technique, the most common of which are detailed below. OpenGL® supports spherical, dual-paraboloid, and cubic environment mapping, while DirectX® 7 only directly supports the cubic method. ATI's Pixel Tapestry accelerates all forms of environment mapping in hardware.

*Spherical Environment Mapping*

Spherical environment mapping uses a single texture as a reflection map. To understand how this texture is created, imagine you are sitting still in one position, but moving your head around. Depending on which direction you look, you will see something different. Now imagine everything you see is not real, but is actually projected on to a sphere, with you at the center. If the image on the sphere was captured and unwrapped so that it became a flat, two-dimensional sheet, this would be the spherical reflection texture or environment map. This environment map can then be wrapped around any three-dimensional object to create accurate reflections, as shown in the illustration below:

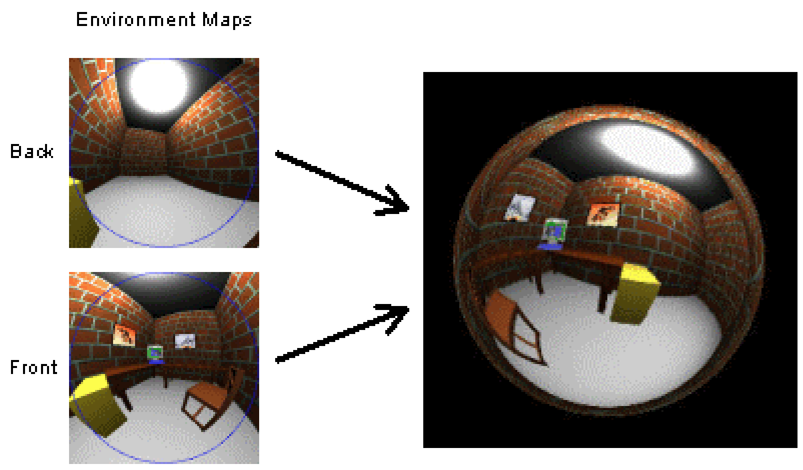


**Spherical Environment Mapping**

This environment mapping method is simple, but has a few drawbacks. Most importantly, it assumes the viewpoint is fixed. If the viewer (at the center of the sphere) moves, the whole texture must be re-created. Also, like any spherical projection, it can introduce significant distortion to the reflection, since it is difficult to accurately portray the surface of a sphere on a two-dimensional plane. Spherical environment mapping is useful for creating specular highlights on fixed, shiny objects, and it works well in conjunction with environment mapped bump mapping. It also consumes the least memory of all the environment mapping techniques.

## *Dual-Paraboloid Environment Mapping*

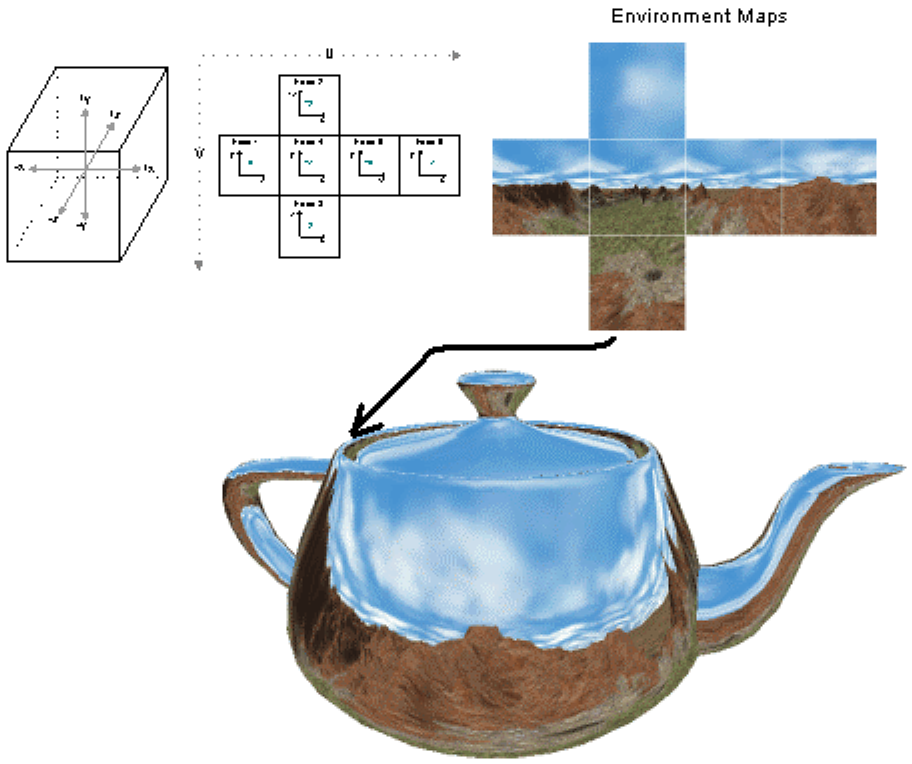
Dual-paraboloid environment mapping is more complex than the spherical method and uses two textures as environment maps (one representing the environment in front of the object, the other representing the environment behind). These textures are square, but mathematically distorted into a paraboloid shape. Dual-paraboloid environment mapping has the advantage of being viewpoint independent. This means that, unlike with spherical environment mapping, the reflection map textures do not have to be updated if the viewpoint moves. Instead, the texture co-ordinates are updated and the texture is re-applied. The downside of this method is that because the textures have to be distorted, they are harder to generate and harder to update on the fly than the textures used in cubic environment mapping. The dual-paraboloid technique is useful for complex reflections that do not need to be dynamically updated, since it requires less memory than the cubic technique.



**Dual-Paraboloid Environment Mapping**

## Cubic Environment Mapping

Cubic environment mapping is the most flexible and the most complex method of environment mapping. It uses six environment map textures, each representing one face of a cube with the viewer at the center. This is illustrated below:

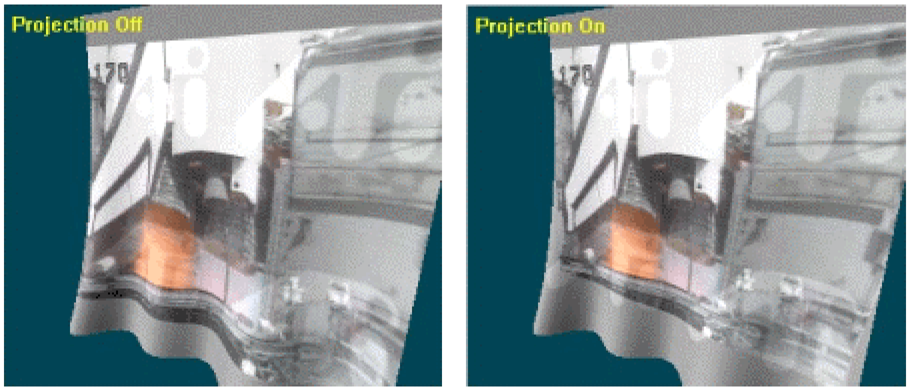


Cubic Environment Mapping

This environment mapping method has several advantages. Because the reflection textures used are simple 2D squares, they are easy to generate and update in real time. The method is also viewpoint independent, so the viewer can move around but the reflection textures only need to be updated i the reflected scene itself changes. The disadvantage of this technique is that it requires a substantial amount of memory to store the six required textures. However, it enables slick effects that were never before possible, like being able to see yourself moving in the reflection on an object.

## Projective Textures

As mentioned before, standard 2D textures work like stickers attached to the surface of a polygon. Texture transformations, however, allow textures to be projected onto a surface, much like the way a slide projector projects an image on a screen. A texture can be "beamed" from any location, and all or part of it will appear on any polygon surfaces it contacts. The following screen shots illustrate the difference this makes. Note how the standard texture on the left is warped on to the curtain surface, while the projected texture on the right stays square:



Projective Textures

Projective textures work much like spotlights, but they have all the capabilities of textures, including filtering and animation capabilities. For example, a movie projector could be set up that projected a moving image (stored as an animated texture or video texture) onto a wall. If a character then passed between the projector and the wall, part of the movie would appear on the character, and an appropriate shadow would be cast on the wall. One of the most exciting uses for projective textures is shadow mapping, which is described later in this document.

### Priority Buffers

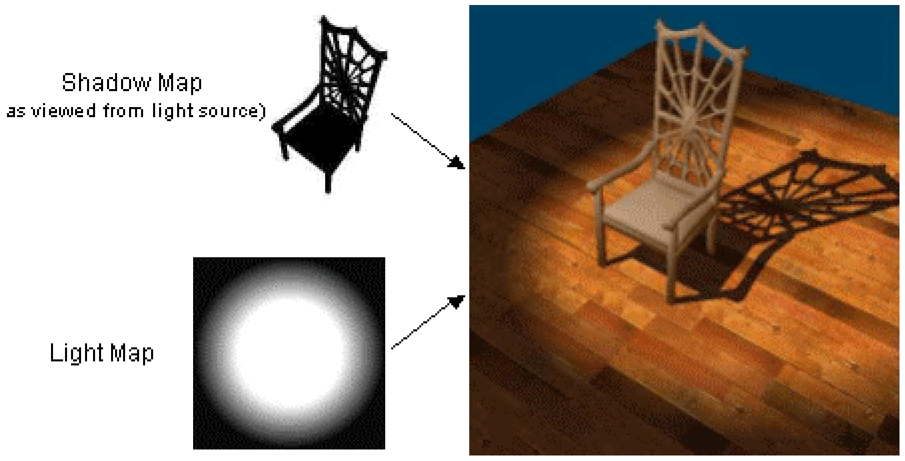
Yet another innovation included in the Pixel Tapestry architecture is support for priority buffers. A priority buffer stores polygons and/or objects according to priority, which is determined by how close the object is to the viewpoint. The closest object or polygon would be assigned a value of 1 in the priority buffer, the next closest a value of 2, and so on. This is similar to a depth buffer or z-buffer, with the difference that object order is more important than distance from the viewpoint. Priority buffering can be performed in software, but for 3D scenes with more than a few objects or polygons it is too slow to be useful. Pixel Tapestry is the first architecture to provide priority buffer support in hardware. Although priority buffers can be used for a variety of purposes, they are best used in combination with projective textures to perform shadow mapping. This next-generation capability is described below.

## Shadow Mapping

Shadows are a very important aspect of a 3D image. They make lighting appear more realistic and help convey a sense of depth. Generating accurate 3D shadows can be very difficult, especially in a dynamic environment with many moving objects and light sources.

Most of today's games, if they implement shadows at all, use pre-rendered shadows. These can look highly realistic and natural, but have the major disadvantage that they can't be used with moving light sources or character models. A few games employ a more advanced method to create volumetric shadows. This involves creating additional polygons that extrude behind a character or object to create a shadow volume. This shadow volume is then used with a stencil buffer to create dynamic shadows. While much more flexible than pre-rendered shadows, volumetric shadows still have serious drawbacks. Every shadow in a scene creates additional geometry to be transformed and processed, which severely impacts geometry. Also, complex cases like objects casting shadows on to themselves (e.g. a character's arm casting a shadow on its torso) or on to complex curved surfaces are prohibitively difficult to implement using this method.

Shadow mapping is a much easier and more elegant method that can be implemented using priority buffers. Using a light source as a viewpoint, a scene is rendered to a priority buffer so that the closest shadow casting object has the highest priority, the next closest has the second-highest priority, and so on. Background objects that do not cast shadows have the lowest priority. A shadow texture is then generated by rendering silhouettes of each shadow casting object, in the order defined by the priority buffer. Finally, the shadow texture is projected on to the scene from the light source, and blended with light maps and existing base textures to produce the final image. This process is illustrated below.

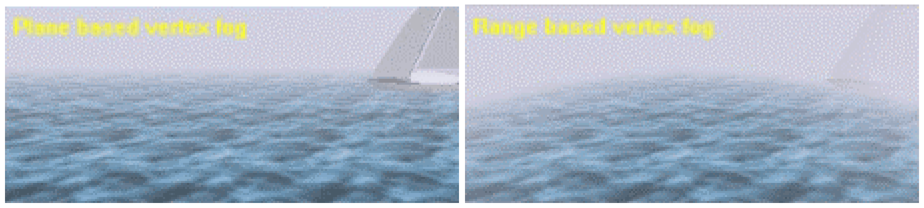


Dynamic Shadow Mapping

Shadow mapping has many advantages over existing solutions that make it much more practical. Because it is accomplished using a single texture per light source, rather than a large number of polygons, it requires much less computation than volumetric shadows. By using hardware-accelerated texture transformations, it can easily handle multiple moving light sources, and accurately warps shadows over other objects. The priority buffer also makes it much easier to create models that cast shadows on to themselves, just like real life.

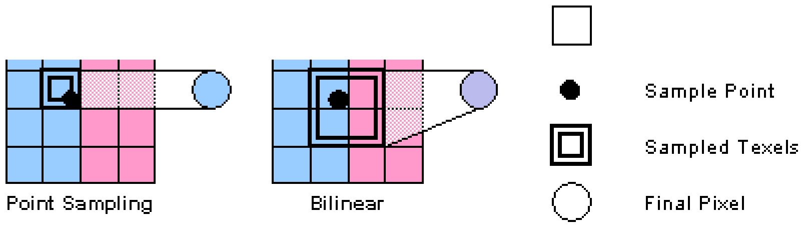
## Range-Based Fog

Fog is a great environmental effect that can be used for a variety of purposes. Most graphics chips use a simplistic implementation of fog that increases its "thickness" according to an object's depth (i.e. the distance from the viewer's "plane" to the object). The Pixel Tapestry architecture depicts fog more realistically by varying thickness according to an object's range (i.e. the direct distance between the viewer and the object). The following diagram demonstrates the advantage of range-based fog:

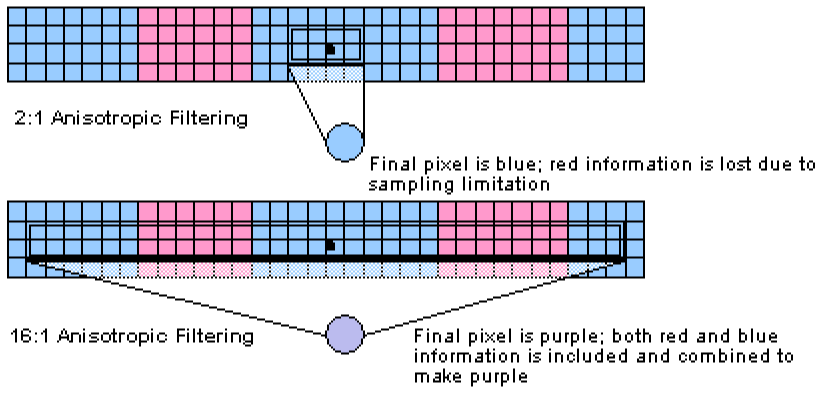


## Improved Anisotropic Filtering

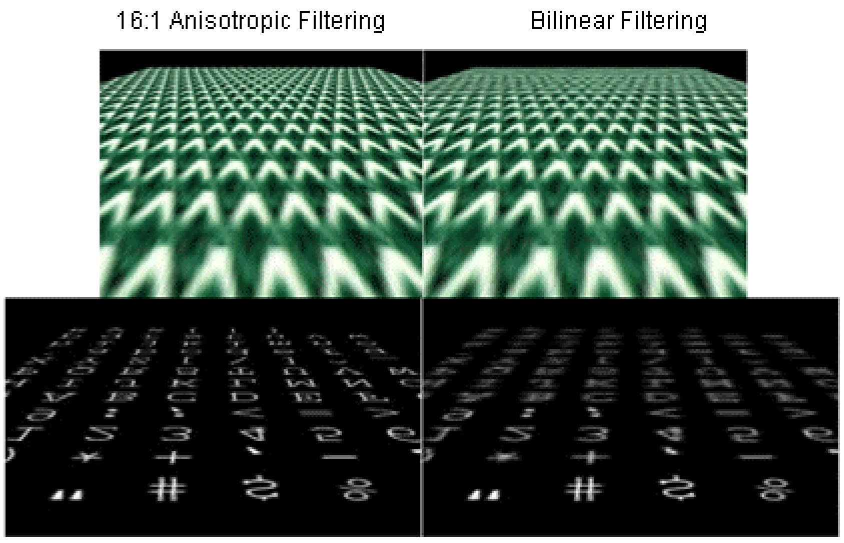
Texture filtering is an important technique that has been included in the past few generations of 3D graphics accelerators. While 3D objects are composed of polygons that can easily be scaled up in size as you get closer to them, the textures applied to the surfaces of these polygons are bitmaps of fixed resolution. Without filtering, the color of each pixel on the surface is determined by sampling a single texel, a technique known as point sampling. As you approach a textured surface in a 3D environment with point sampling, it begins to suffer a severe loss of detail and appears blocky and unattractive. The first filtering method used to overcome this problem is known as bilinear filtering, which smooths out the blockiness by sampling the two nearest texels in the vertical and horizontal dimensions (4 texel samples altogether) and blending them together using a weighted average to determine the color of each pixel.



Bilinear filtering generally does a decent job of improving image quality when used on textures that are square or almost square. However, it doesn't work so well on surfaces that are sloped at a steep angle to the user. In this case, the texture must be "stretched" horizontally or vertically, meaning its resolution must be greatly reduced in one dimension relative to the other. The result is that textures appear sharp and detailed on the part of the surface that is close to the viewer, but blurry and indistinct on the part that is farther away. To deal with this situation, a more advanced form of filtering called anisotropic filtering must be used. It works just like bilinear filtering, except that it uses more texel samples in the direction of the stretch to retain detail on all parts of the surface.


2:1 Anisotropic Filtering
Final pixel is blue; red information is lost due to sampling limitation
16:1 Anisotropic Filtering
Final pixel is purple; both red and blue information is included and combined to make purple

The effectiveness of a particular anisotropic filtering algorithm is determined by the maximum "degree of anisotropy" it allows. This is a ratio which determines how far a texture can be stretched before its image quality starts to degrade. The Pixel Tapestry architecture is the first to allow a maximum degree of anisotropy of 16:1, much higher than any existing graphics product. This ensures that textures stay sharp and clear no matter what angle they are viewed at. This is particularly important for text, as demonstrated in the following examples:



## Anti-Aliasing Effects

Anti-aliasing effects such as full-scene anti-aliasing, motion blur, depth of field, and soft reflections have received a lot of attention recently as means for improving the realism of 3D scenes. They are all related by the fact that they are implemented using some form of an accumulation buffer to blur or smooth parts of the scene. While enabling these effects tends to be detrimental to graphics performance, they are certainly of interest to game developers for certain applications, including working with low resolution displays, 3D cut-scenes, and limited special effects. For this reason, ATI's Pixel Tapestry architecture includes support for all of these effects.

**Full-Scene Anti-Aliasing**

Aliasing specifically refers to a set of visual artifacts that result from limited display resolution. These include jagged "staircase" effects on the edges of objects, as well as shimmering or "popping" effects on narrow objects viewed at a distance. The lower the display resolution, the more pronounced these artifacts become. Spatial anti-aliasing refers to a set of techniques used to counteract these effects. Most existing graphics chips implement a limited form of spatial anti-aliasing called edge anti-aliasing, which uses an algorithm that attempts to identify strong edges in a scene and smooth them out. Unfortunately, this technique requires significant pre-processing and sorting of polygons to work properly, making it impractical for most applications.

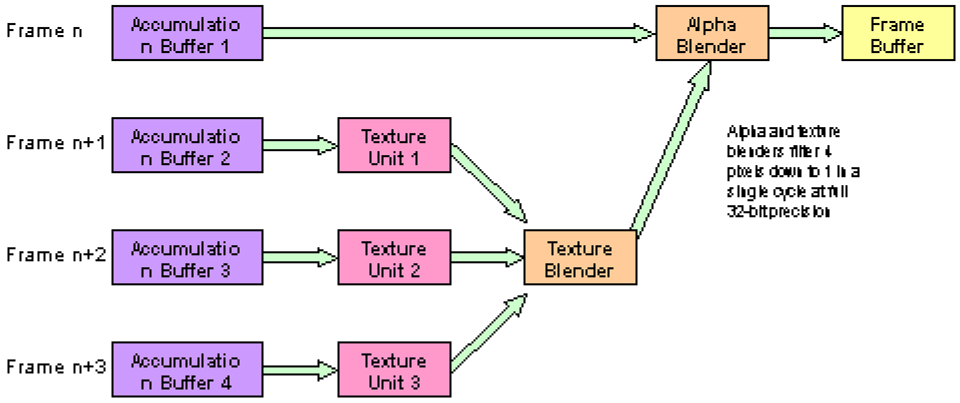
A more useful, effective, and flexible technique is full-scene anti-aliasing. Rather than identifying the edges in a scene, full-scene anti-aliasing uses a method known as supersampling to reduce or eliminate aliasing artifacts. This involves rendering a scene at a higher resolution than the final output, and then filtering and scaling the image down to the desired resolution. The result is that more detail is captured in the final image, which lessens or removes aliasing artifacts.

Although this method can greatly improve 3D image quality at low resolutions, the fact that multiple copies of an image must be rendered (multiplying the fill rate requirement by a factor of 4 or more) makes it impractical at higher resolutions. Graphics chips that are being released this year will enable smooth, high quality 3D gaming at resolutions of 1280x1024 and higher. At these resolutions, aliasing artifacts are difficult to detect, and most users will prefer the improved sharpness and quality to low-resolution anti-aliased images.

**Motion Blur**

Moving 3D images rendered in real time differ considerably from the moving images captured with a video camera that records to film. In the case of the video camera, an image is created by opening a shutter on a lens and exposing film to a scene for a fixed period of time. If an object in the scene is moving while the shutter is open, this motion is captured on film as a blurring effect. If you look at the individual frames on a roll of film, this blurring is clear, even though it is not apparent when the film is played back on a movie screen or a television set. In the case of 3D rendered images, however, moving objects are not blurred. They remain sharp and clear regardless of how fast they are moving. Blurring is what helps make motion appear more smooth and fluid of TVs and movie screens than it does on monitors. Monitors must compensate by displaying images at much higher frame rates (at least 60 frames per second to achieve comparable quality to a 24 frame per second movie projector).

In order to simulate the smooth transition between frames on a monitor without increasing frame rates, a technique called motion blur (also known as temporal anti-aliasing) can be used. It works by storing copies of successive frames in an accumulation buffer, and blending them together to produce a final image in which moving objects are blurred in the direction of motion. The more copies are stored, the better the effect looks. The diagram below illustrates how ATI's Pixel Tapestry architecture uses its three texture units to create a 4-frame motion blur effect:



*(Motion Blur)*

Much like full-scene anti-aliasing can be used to improve the quality of low resolution images, motion blur can be used to compensate for low frame rates. At higher frame rates, the blurring effect becomes less noticeable. In fact, at frame rates of 60 frames per second or higher, the human eye will automatically begin blurring successive frames together. In general, most users will prefer the more precise control and quicker response that is afforded by high frame rates to lower frame rates combined with motion blur.

## Soft Reflections and Soft Shadows

Materials can have widely varying degrees of reflectivity, ranging from matte (no reflections) to mirror-like (near perfect reflections). Many materials, like plastics and lacquered surfaces for example, are reflective but produce only blurred, partial reflections of the environment. To realistically reproduce this effect in a 3D scene, reflections can be anti-aliased or blurred. The result is a "soft" reflection. Similarly, most light sources in real life do not produce sharp-edged shadows. More realistic shadow effects can be produced using anti-aliasing.

**Soft Reflections**

**Soft Shadow**

These effects use accumulation buffers to store multiple copies of an object's reflection or shadow, and then blend them together for the final image. As with other anti-aliasing effects, the more copies are used the better the final image looks, but the tradeoff is reduced performance and frame rate. If used judiciously, these subtle effects can greatly enhance the quality of a 3D scene while still maintaining reasonable frame rates.

### Depth of Field and Fresnel Effects

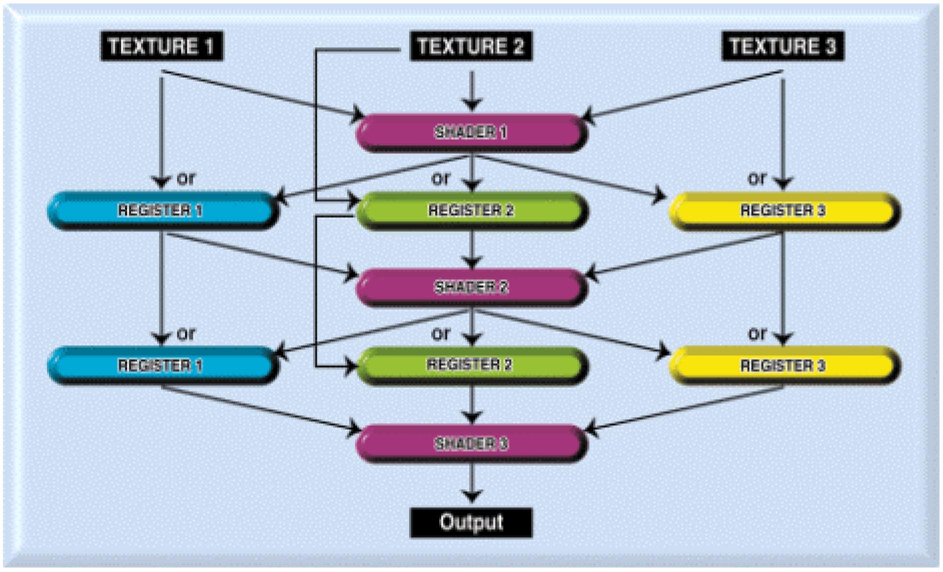
Another application of anti-aliasing is known as focal anti-aliasing, which causes parts of an image to blur according to their distance from the viewer. This can be used to simulate the effects of focus, known as a depth of field effect. It can also be used to recreate the effects of looking at an image through frosted glass, known as a Fresnel effect.

## Programmable Pixel Shaders

As you can see from the list of topics described thus far, ATI's Pixel Tapestry architecture is capable of applying a wide range of effects to 3D surfaces in order to enhance their level of realism and detail. The RADEON graphics processor can accelerate all of these features in real time for maximum performance. But game developers are a creative bunch, and the best ones frequently come up with ideas for new graphical techniques to achieve a desired visual effect. In the past, these new techniques could not be hardware accelerated without creating a new graphics chip. Game developers therefore had to write special non-accelerated software routines if they wanted to use these new techniques, and the performance cost associated with this type of solution could often end up being prohibitive. The developer might be forced to leave the feature out of the game altogether, in the hopes that it could be used in their next game when faster computer hardware is available to consumers.

Programmable pixel shaders are an exciting innovation that gives developers a new degree of freedom in creating advanced visual effects. A pixel shader is a simple routine that determines the color of a pixel based on variety of inputs (base material color, light color, surface reflectivity, bumpiness, transparency, etc.). Pixel Tapestry architecture allows a developer to actually program custom shader routines into the graphics processor itself, allowing them to be accelerated in hardware.

Pixel Tapestry architecture's advanced pixel shader support allows up to three stages of mathematical operations with up to three inputs each to be performed on every pixel in a scene. Each stage of the shader routine can use the results of a previous stage as one of its inputs, and can perform a number of operations including addition, subtraction, multiplication, alpha blending and dot product. The following diagram illustrates the flexibility of the pixel shader hardware found in the RADEON graphics processor:



**Programmable Pixel Shader Data Paths in the RADEON Graphics Processor**

## Performance

A pretty picture is only part of the story when talking about realistic, immersive 3D worlds. 3D games are interactive applications. No matter how detailed a 3D environment is, the illusion of reality falls apart if you don't get smooth, immediate feedback from your actions. When you turn your head in real life, the view from your eyes changes immediately. There is no delay, flickering, or sudden jumping from one view to another. This need for smoothness and instant feedback is so important that game developers will usually limit the level of detail and image quality they display in order to maintain smooth gameplay on a standard system.

To achieve instant feedback and fluid movement in a 3D application, the software and hardware must be capable of updating the displayed image with new information fast enough that the human eye cannot detect the change. On low-resolution televisions and movie screens, this can require as little as 30 or even 24 frames per second. On high resolution monitors and flat panels, however, a frame rate of about 60 frames per second is required. This is important for fast-paced action games where sudden movements, split-second reactions and fine aim are vital.

### Measuring Graphics Performance

When looking for one number that gives the most accurate indication of the relative performance of a graphics processor, the most commonly quoted statistic is fill rate. Unfortunately, fill rate is an ambiguous term that can refer to several different statistics. These include:

- Pixel Fill Rate: The number of pixels the graphics processor can render per second. Calculated by multiplying the graphics engine clock speed by the number of rendering pipelines, and measured in megapixels per second (Mpix/sec).

- Texel Fill Rate: The number of texels (where a texel refers to a single pixel from a texture bitmap) the graphics processor can access per second. Calculated by multiplying the graphics engine clock speed by the total number of texture units, then multiplying again by the number of filtering samples that can be accessed for each pixel in a single clock cycle (1 for point sampled or unfiltered, 4 for bilinear filtered, 8 for trilinear filtered, etc.). Typically measured in gigatexels per second (Gtex/sec).

- Effective Fill Rate: The statistic most commonly used in graphics marketing when referring to fill rate. Equal to the number of pixels a graphics chip can process each second multiplied by the maximum number of textures that can be applied to a pixel in a single clock cycle (determined by the number of texture units). Usually measured in megatexels or gigatexels per second (where 1 gigatexel = 1000 megatexels).

To complicate things further, fill rate numbers only represent the maximum "theoretical" performance of a graphics processor. They are based on the assumptions that the processor will always run at 100% efficiency, and that performance will never be constrained by other factors such as memory bandwidth, AGP bus bandwidth, or CPU speed. When running real applications, however, these assumptions are almost never valid.

There are many factors that affect the frame rates a 3D application can achieve. In every situation, however, there is usually one bottleneck that has the largest effect on performance. The key to improving speed is finding and eliminating these bottlenecks. For example, when the PCI bus became a bottleneck for graphics, AGP was developed to overcome it. This exposed the CPU as a new bottleneck. Technology has since compensated for this by offloading most of the graphics work formerly done by the CPU (such as triangle setup and transform & lighting) on to the graphics processor. Now the CPU is no longer a major limiting factor in graphics performance. For the current generation of graphics hardware, the primary speed differentiator has become memory bandwidth.

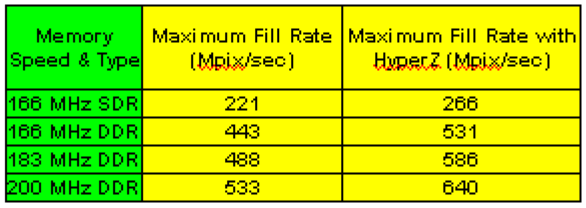
As a scene becomes more complex, the amount of data required to accurately set up and render it grows exponentially. The speed at which graphics chips can process data has increased so rapidly that it has outstripped the capability of today's most advanced memory technologies to deliver it. Graphics chips try to cache frequently used data to avoid having to fetch it from external memory, but these caches must be kept fairly small to keep costs reasonable. Without a solution to this problem, things like increasing the clock speed of the graphics processor or adding many parallel rendering pipelines are just like spinning the tires of a car stuck in the mud. Until memory designers come up with completely new ways to improve performance, a different kind of solution must be found.

## **<u>HyperZ™ Technology</u>**

ATI's RADEON chip takes a new approach to the memory bandwidth limitation issue. The innovative HyperZ technology improves memory bandwidth efficiency without the added cost of exotic embedded or proprietary memory technologies. By eliminating this key bottleneck, the final barrier to realistic, real-time 3D is overcome.

The speed and type of memory accompanying a graphics chip places upper bounds on the fill rate it can achieve. The table below shows how HyperZ boosts the maximum number of 32-bit color pixels a graphics chip can render each second when equipped with various types of frame buffer memory:

Memory Speed & Type Maximum Fill Rate(Mpix/sec) Maximum Fill Rate with HyperZ (Mpix/sec)



| Memory Speed & Type | Maximum Fill Rate (Mpix/sec) | Maximum Fill Rate with HyperZ (Mpix/sec) |
|---|---|---|
| 166 MHz SDR | 221 | 266 |
| 166 MHz DDR | 443 | 531 |
| 183 MHz DDR | 488 | 586 |
| 200 MHz DDR | 533 | 640 |

Figures assume typical 3D game application with average depth complexity of 3.

All 3D applications today use some form of depth buffer to keep track of which objects are visible to the viewer. The most commonly used type of depth buffer is the Z-buffer, although other variations such as W-buffers can also be used. With the current generation of graphics hardware, depth buffer reads and writes typically account for more than half of all memory bandwidth usage. HyperZ slashes the amount of bandwidth required for the depth buffer without sacrificing any quality in the final image. HyperZ employs a set of three techniques that combine to take 3D performance to new heights. These are Hierarchical Z, Z Compression, and Fast Z Clear.

## Hierarchical Z

A major problem that all game developers have to face when designing 3D worlds is known as overdraw. To understand what overdraw is, consider a 3D scene where you are looking through a small window into a room beyond. Some of the walls and objects in the room will be visible through the window, and some will not. Most graphics processors have no way of knowing what parts of the scene will be visible and what parts will be covered until they begin the rendering process. They must then check the depth buffer for each pixel and determine whether to draw it or not. In this process, many pixels will be written to the frame buffer, then overwritten by new pixels from objects that are closer to the viewer. Overdraw is the term for this overwriting of pixels in the frame buffer. A measure of the amount of overdraw in a scene is called depth complexity, which represents the ratio of total pixels rendered to visible pixels. For example, if a scene has a depth complexity of 3, this means 3 times as many pixels were rendered as were actually visible on the screen. This also means that 3 times the fill rate would be needed to display the scene at a given frame rate as would be needed if the was no overdraw.

Overdraw is a major source of inefficiency in 3D games. Depending on the content of a scene, depth complexity can vary from 1 to as high as 10 or more, although values around 2 or 3 are most common. Hierarchical Z represents a new, more efficient way of dealing with overdraw on the graphics chip. It works by examining scene data before it is rendered, to determine which pixels will be visible and which will not. Any pixels that will not be visible to the viewer are discarded and not rendered at all. This dramatically reduces overdraw and significantly boosts effective fill rate, with a corresponding improvement in performance.

## Z Compression

Data compression is a commonly used technique in any situation where a large amount of information must be transferred over a bandwidth-limited medium. The Z-buffer is a good example of just such a situation. Since transfers to and from the Z-buffer account for such a large percentage of memory bandwidth usage, it is an obvious target for data compression. HyperZ uses an advanced, lossless compression algorithm to reduce the amount of data transferred during Z-buffer reads and writes. Compression and decompression is done in real time, freeing up additional memory bandwidth without sacrificing image quality.

## Fast Z Clear

After each frame of a 3D scene is drawn, the depth buffer allocated to that scene must be cleared before it can start accepting data for the next frame. Conventional graphics processors clear the buffer by writing a value of zero to each location. This process still counts as a Z-buffer write, and therefore requires memory bandwidth over and above that which is required to render the frame. HyperZ allows the Z-buffer to be cleared very quickly (approximately 64x faster than conventional architectures) without having to write anything to the Z-buffer, further reducing memory bandwidth consumption and enhancing performance.

## Summary

3D graphics technology is currently being driven by 3D games. Game developers are seeking the ultimate goal of photorealistic, real-time 3D rendering to take their games to the next level. This requires the ability to render lifelike characters and environments, capture visual details like reflections and shadows, and do it all without compromising frame rate. ATI's trio of new technologies (Charisma Engine, Pixel Tapestry and HyperZ) overcome existing technical barriers to make these things possible.

This document has described just a selection of the things that can be done with ATI's new technologies. With advanced programming interfaces like DirectX® 7 and OpenGL® 1.2, most of the techniques described are simple to implement in real applications. The result is an environment where the only limit to what can be experienced on one's PC is a developer's creativity. The Charisma Engine, Pixel Tapestry architecture, and HyperZ technologies will appear in ATI's RADEON family of graphics products.