Evaluating Reconfigurable Texture Units for Programmable Graphics Cards

A Thesis
in STS 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Christopher Palmer

April 23, 2007

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for papers in Science, Technology, and Society courses.

____

Approved ___________________________________ (Technical Advisor)
Prof. Kevin Skadron

Approved ___________________________________ (Science, Technology, and Society Advisor)
Prof. Peter Norton

Table of Contents

Glossary

Anisotropic Filtering	– A texture filtering function that uses two pre-scaled textures in the horizontal and vertical direction to better display textured surfaces at glancing angles with less aliasing
Bilinear Filtering	– A texture filtering function that interpolates between the four nearest texels to give an approximate color for coordinates with non-discrete values
Central Processing Unit (CPU)	– The general-purpose microprocessor in a computer that governs the execution of a computer
Filtering Functions	– Functions that specify how texel data should be combined and interpolated when a requested value lays between discrete texture coordinates
Gather	– An operation that gathers pixel data from surrounding pixels and is combined to produce an output for another pixel
Graphics Processing Unit (GPU)	– A microprocessor optimized for graphics-related operations, specifically parallel execution of floating point arithmetic
Instruction Set Architecture (ISA)	– A specification defining a set of commands and parameters that are valid inputs for a given piece of hardware
Linear Interpolation (LERP)	- A function that combines two values (a and b) using a scaling constant (alpha): c = (1 – alpha)*a + alpha*b
Mip maps	– Pre-scaled array of textures created in texture memory to account for depth and provide more sophisticated filtering
Nearest Filtering	– Given a texture coordinate (u, v), find the texel at the nearest discrete point by rounding down
Pixel	– Colloquially defined as a single square of color that is one component of a group of pixels composing a digital 2-D image

Pre-rendered	– Describes images rendered offline or not in real-time, generally of a much higher quality than images rendered interactively
Rasterization	– A rendering technique that operates on triangles one at a time without regard to reflections or other complex light interactions
Ray Tracing	– A rendering technique using simulated light rays to trace out a camera's viewpoint in a 3-D scene, can capture effects such as caustics, shadows, reflection, and refraction
Render	– A process that creates a 2-D image of a larger, usually 3-D, scene incorporating effects such as lighting and shadows
Texel	– Much like a pixel, but referring to a texture map
Texture Map	– An image, consisting of texels, stored in memory on the graphics card
Trilinear Filtering	– A texture filtering function that takes a bilinear sample from two adjacent mip maps and interpolates for the final color

Abstract

Modern graphics cards have enormous arithmetic capabilities far exceeding those of CPUs. The advent of vertex and pixel shaders allowed programmers to customize the operation of GPUs and use them for applications other than graphics. Recently, however, many scientific applications have been ported to GPUs, but cannot take full advantage of the hardware due to architectural differences in memory accesses. Texture maps, traditionally used for rasterization, are not optimized for the random access patterns used in applications such as ray tracing. This inefficiency introduces delay, increased memory bandwidth, and consequently increased power consumption. This thesis introduces an alternative GPU texture unit design to optimize dependent texture accesses, operations very common in scientific applications. The design attempts to maximize speedup while not affecting the performance of normal texture accesses. The improvements described serve as the foundation of a more formal proposal to graphics manufacturers to be implemented in future hardware.

Chapter 1: Introduction

Computers can never do their work fast enough. The public wants their computers to respond quickly, movies to have better special effects, and video games to be more immersive. Scientists and researchers want simulations that take more variables into account and visualizations that are more life-like, and they want extreme numerical accuracy. One technology can address all these needs: the graphics card.

Also known as a *GPU*, a graphics card is a printed circuit board the *CPU* uses to off-load graphics-related calculations. Instead of performing all the computer's work, the CPU can give graphics operations to the GPU while it executes other tasks. This separation of duties allows computers to perform graphics tasks much faster and more efficiently. Since GPUs perform only these graphics-related operations, the chip inside could be optimized for these very specific commands. With such optimizations, GPUs can achieve enormous computing power, topping 278.6 billion operations per second (Shrout, 2005). That is over 45 times more than an Intel Pentium 4 processor (Hinton, 2001).

Scientists who use math-intensive simulations (for example, to detect white blood cells in live video, to reconstruct CAT scan images, or to generate *3-D* images) need such processing speed. The problem is that graphics cards were designed from the beginning to deal with graphics, not the general computing problems researchers want to use them for. But researchers have found techniques that can use the GPU's processing power for non-graphics applications. Algorithms with operations that are very parallel in nature, like ray tracing, are well suited for the GPU's stream processing model.

1.1 Ray Tracing

Ray tracing is a technique for generating a *2-D* image from a description of a 3-D world viewed from a virtual camera in the scene. The most widely recognized direct application of ray tracing is in computer generated (CG) movies such as Pixar's "Cars" (Figure 1). Every frame in the movie was created by a computer, which *rendered* the scene from a virtual camera placed inside a mathematical description of the world. The results are very realistic. Ray tracing can model many natural phenomena such as reflection, refraction, and depth of field.

Figure 1 – Lightning McQueen from Pixar's "Cars" (Pixar)

Ray tracing uses the physics of light as its foundation. Given a camera in a 3-D scene with objects, a grid of squares which correspond to *pixels* on the screen can be constructed. By sending mathematical rays that go through these grid cells and tracing them into the scene, we can determine which objects in the scene they intersect (Whitted, 1980). When a ray intersects an object, the color of that object is recorded and becomes the color for the corresponding pixel. Repeating this operation for every pixel in the final image produces the completed picture (Figure 2). Ray tracing began with the work of Turner Whitted (1980). The term *Whitted ray tracing* describes his technique of shooting rays in a scene of geometry.

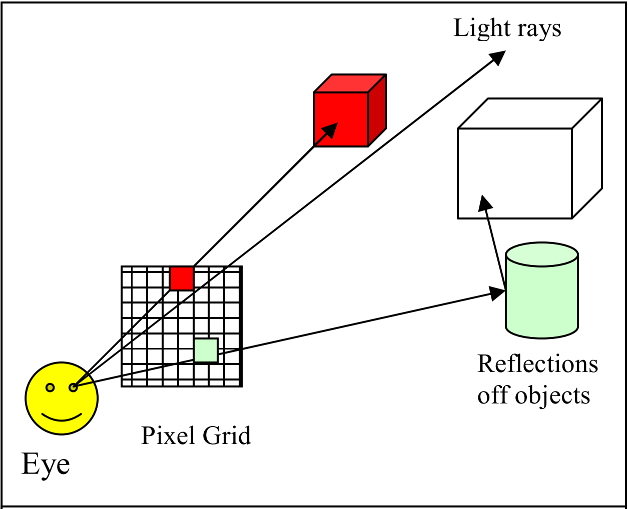


Figure 2 – The ray tracing process (author)

Ray tracing, as a rendering technique, has several key advantages. First, it creates incredibly realistic scenes. Because the technique tries to emulate the physics of light, it can capture effects that traditional rendering techniques, like *rasterization*, cannot. Effects such as water caustics, light magnification, and soft shadows (shadows without hard edges) are a few examples. Yet ray tracing can be orders of magnitude slower than other rendering techniques for similar scene complexity. Even though the resulting image is of much higher quality, the time needed to produce even a single frame can limit ray tracing's interactive applications. A single frame from the movie "Cars" took upwards of 50 hours to render (Lasseter, 2007), even with the help of thousands of computers working simultaneously.

Recently, researchers have begun mapping ray tracing onto GPUs (Purcell, 2004); (Foley, 2005). They are attempting to leverage the sheer computing power of the GPU to solving ray tracing problems on commodity hardware instead of only with expensive computing clusters. Success would make extremely realistic simulations of 3-D scenes possible, including video games and scientific visualizations. Immersive worlds, like those seen in today's CG movies, may soon run in real-time on your personal computer. The effect on scientific visualizations would be equally impressive. Everyone from chemists to mechanical engineers would benefit greatly from increased detail and complexity of their 3-D models.

1.2 Optimizing the Process

This thesis describes a proposed hardware change to current GPUs that would let them perform ray tracing much more efficiently. CPU programs ported to GPUs run into many obstacles. Specifically, memory accesses are very different between the two types of processors. Data structures that are efficient for CPU programs are very inefficient on GPUs because of these differences. CPU memory can be thought of as a large room with a huge row of adjacent cubby-holes (Figure 3). Data can be placed in any cubby-hole you want and you can access any piece of data instantly if you know where is located. GPU memory, conversely, is like walls of PO Boxes spread among many different post offices. You can place data in any PO Box you want, but you first have to travel to the appropriate post office. Once in a post office you can very quickly access all boxes in it,

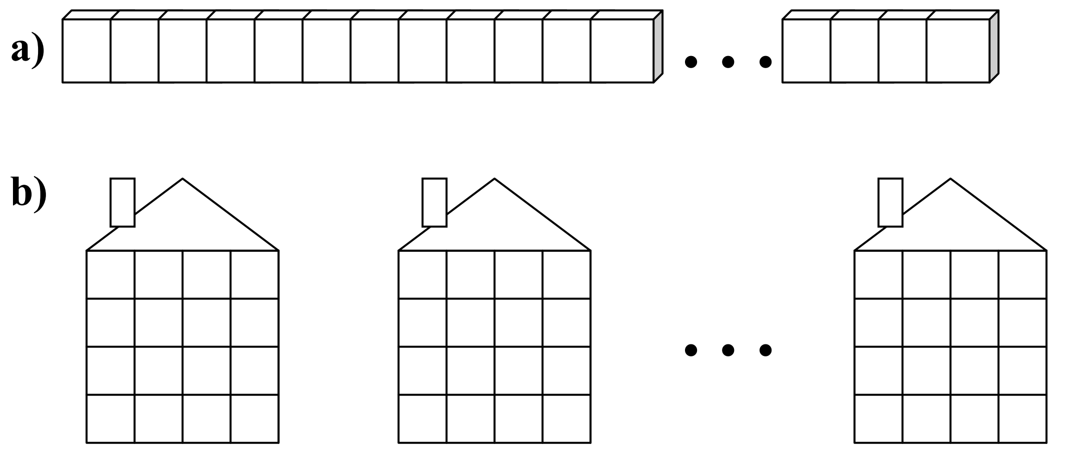


Figure 3 – (a) CPU memory like a row of adjacent bins to place information (author) (b) GPU memory grouped in collections, located in different buildings

but if you need data from a box at another post office, you have to drive home, drop off the data you just got, and drive to the next one.

It is the driving back and forth between home and the post offices that makes this system of storage (GPU memory) slower if you access data from many different locations. With cubby-holes, however, although the row of them is very long, you can stay in the same building to access them all. This thesis was an attempt to speed up accesses to memory (the PO Boxes) by an analogous modification (Figure 4). If you know you want one item from Post Office 1, one from Office 2, and another from Office 3, why go home between trips? Why not consolidate your trips to memory and get all your data before returning home?

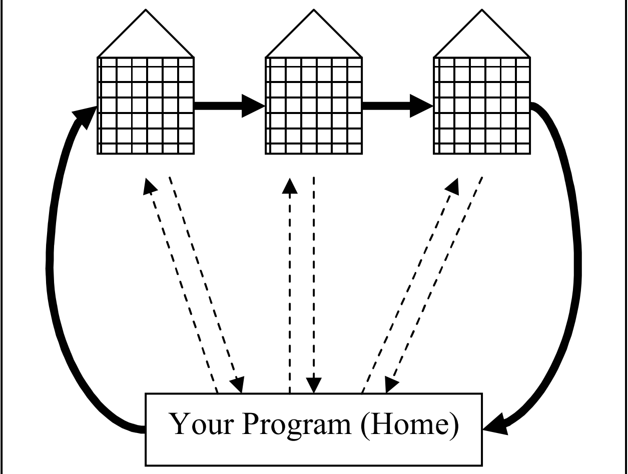


Figure 4 – Dotted lines show an inefficient access pattern, solid lines show the proposed more efficient access (author).

Ray tracing (and other scientific GPU applications) use different sections of GPU memory to store dependent pieces of information. Information retrieval is then much like a scavenger hunt. The data retrieved from Post Office 1 tells you which box to look in at Post Office 2. The data there shows where to look in Post Office 3, and so forth. By going directly between post offices and returning home only with the final result, you save driving time (speed), you save the amount of data you had to carry back and forth (memory bandwidth), and you save gas (power).

1.3 Social and Ethical Implications

Every television advertisement with an animated logo, every special effect in movies, and everything displayed on every computer monitor uses graphics card functionality. Computer graphics imagery has permeated throughout modern life. We can't walk down the street without seeing a flyer with computer drawn artwork, or go to class without a PowerPoint lecture. Computer graphics are most visible to the public in the form of CG animated movies such as: "Toy Story," "Finding Nemo," and "Shrek." Video games (both computer games and console games) use 3-D images to create a story and convey information.

The difference between movies and video games is interactivity. You can't manipulate an animated movie once it is completed. The camera can't be shifted; you can't change the lighting, or affect the finished image in any way. These types of images are said to be pre-rendered. This means that the frames used in the movie were created and put together ahead of time. Video games, on the other hand, are interactive; you can change the image in real-time. Better ray tracing performance would affect both the films and games that use it.

This is not to say that these effects have never been achieved before. Pre-rendered graphics have long been able to create highly realistic effects. Today, however, these effects can be realized in real-time on consumer graphics hardware. A massive array of expensive computers is no longer needed to achieve these effects. Slipped into a computer, a card costing a couple hundred dollars can be slipped into your computer delivers these amazing effects at home. High-quality rendering performance could reach consumers at low cost.

Higher fidelity images might only further some disturbing trends in children's video games and television. Studies have shown that nearly 33 percent of children play video games every day and almost 7 percent play for more than 30 hours per week (BBC, 2000). These figures would frighten some parents and increased graphical realism may only raise them.

Realistic, violent games, such as "Grand Theft Auto," are controversial for their graphic depictions of violence. Some believe they promote criminal activity. Others believe these are simply fantasy worlds and that playing similar games does not influence players' actions. Current research, however, offers some reassurance. Violent crime in the United States has dropped 2.3 percent since 2003 (Game, 2005). Because this decrease happened during the release of the Grand Theft Auto series, these games' effect on crime is open to question.

Chapter 2: Literature Review

2.1 Rasterization

Ferdinand Braun introduced the first practical use for the CRT (cathode ray tube) in 1897. Braun constructed an oscilloscope by attaching a screen covered in a layer of phosphorous to the interior of the tube (Carlson, 2006). Electrons fired out of the cathode and struck the screen, causing the deposits of phosphorous to glow. By controlling the position and intensity of the electron beam, shapes could be “drawn” on the screen (Figure 5). Philo Farnsworth took advantage of this property in 1927 with the introduction of the image dissector (Carlson, 2006). The electron beam swept over the entire screen in a series of horizontal lines, called *raster lines*. Farnsworth’s 60-line raster image began a revolution in display technology leading to uses in early television.

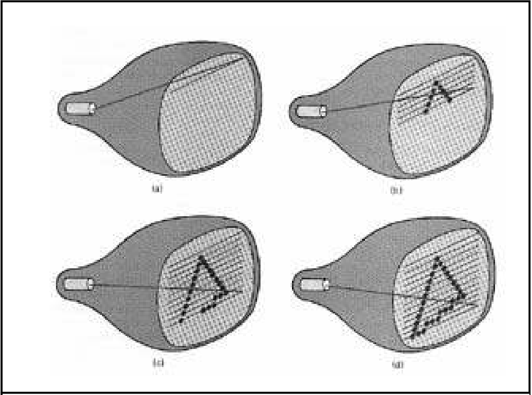


Figure 5 – Raster lines on a CRT (Carlson, 2006)

By 1967 raster lines were used to draw polygons on computer monitors from mathematical representations. This is known as rasterization. Wylie’s algorithm used the three vertices of a triangle to select on-screen pixels to be colored to fill the outline of the triangle (Wylie, 1967). This technique was paired with previous work to draw 3-D worlds on computer monitors very quickly. Gouraud (1971) added to the realism of these scenes with the introduction of a coloring algorithm, creating color gradients across triangles instead of a single color. Phong (1975) built on this scheme by including support for effects such as specular highlighting.

This effect reproduces the sheen seen on some objects when viewed at a glancing angle from a single light source. Yet neither Gouraud nor Phong shading are accurate physical simulations of light. Both are empirical algorithms that approximate the proper effect.

Catmull (1974) advanced the technology of rendering these shaded triangles. He used texture maps in conjunction with rasterization to color rendered triangles with pixel data from textures. Instead of just applying a color to a pixel, users could lookup a color value in a texture and use that as the pixel's color. For example, a texture image of bricks could be applied to a square consisting of two triangles to create a brick wall. Soon after, new uses for texture maps as lookup tables made new special effects possible. Blinn (1976) introduced specular mapping that was variable throughout a surface instead of interpolated between vertices. Combined with bump mapping techniques (Blinn, 1978), this created very complex looking scenes for very little cost in terms of computation time. Both techniques create the illusion of a very high polygon count without increasing the number of actual polygons drawn.

2.2 Ray Tracing

For close to 25 years, rasterization was the main way to create realistic computer-drawn scenes, until Whitted ray tracing was introduced in 1980. In a groundbreaking paper, Turner Whitted introduced a new technique for rendering 3-D scenes (Whitted, 1980). His method of modeling light rays extending from a virtual camera could render scenes unthinkable using traditional rasterization. For example, curved surfaces were supported, which under rasterization required the approximation of the surface using countless triangles. Whitted ray tracing also *implicitly* supported shadows, unlike rasterization which required a complex collection of multiple textures (Cook, 1984) to achieve the same effect.

The increase in scene complexity that ray tracing supported required acceleration structures to speed up the algorithm. It originally worked by testing each ray against every object in the scene for intersection. This functioned for small scenes, but the execution time for larger scenes grows exponentially with the number of objects (Szirmay-Kalos, 2002). One such acceleration structure used a uniform 3-D grid to organize objects so rays only had to be tested against objects located in specific grid cells (Fujimoto, 1986). This technique works well on scenes with uniform distribution of objects, but becomes inefficient for non-uniform, densely populated regions of the grid. Oct-trees generalize the cells of uniform grid by allowing them to subdivide into smaller cells for denser areas of the scene, while staying large in empty areas. Glassner characterizes this approach as a way to reduce rendering times. Less time is then spent testing for ray intersection because grid cells adapt to the structure of each scene (Glassner, 1984).

The success of oct-trees in reducing render time was expanded by Hook (1995). He used a spatial partitioning structure known as a *k*d-tree to trace a ray through a scene. *K*d-trees are a generalization of oct-trees and allow very efficient ray traversals and object intersections. In 2002, a survey of all major ray tracing acceleration structures found that *k*d-trees were the most efficient (Szirmay-Kalos, 2002). Since then, some new optimizations attempt to speed up the operation of *k*d-trees. Most recent attempts aim at speeding up the construction of these trees, which can be time-consuming, by creating trees which can be updated in real-time instead of offline (as was previously done)

(Gunther, 2006). With Gunther's process, a scene's tree can be updated dynamically provided all possible objects that could be in a scene are known ahead of time.

2.3 Commodity Graphics Cards

For many years these rendering methods, especially ray tracing, were available only to scientists with access either to large supercomputers or extremely expensive graphics workstations. The introduction of the commodity consumer graphics card changed the public's access to this technology. One of the first large-selling consumer graphics cards, "Voodoo," was released in 1996 by 3dfx (Hachman, 1996). For the first time, advanced graphics were available on home computers. However, these were rasterized graphics, not ray traced. Ray tracing was still too slow to allow interactive applications.

In 2001, nVidia released its GeForce3 graphics card. It was one of the first cards to support vertex and pixel shaders in hardware (nVidia, 2007). Support for shaders meant that developers could customize how vertices, colors, and textures were combined to create the resulting image. This customizable hardware supported techniques like shadows (McCool, 2001) in real-time which were previously achievable only with ray tracing. Other ray tracing effects were implemented on these programmable GPUs, but Purcell (2004) decided to move the entire process of ray tracing onto GPUs. Purcell's PhD thesis in 2004 described a very novel way to implement a fully functioning ray tracer on standard graphics cards. His technique took advantage of the fact that using pixel shaders, and careful use of texture memory, any operation performed on a CPU could also be mapped onto the GPU (Purcell, 2004). This made the rasterization-based

GPU effectively equivalent in functionality to a general-purpose CPU, with performance and speed as the trade-off.

The GPU, a commodity piece of already in most computers, could now be used to provide professional-quality ray traced images. The problem was that ray tracing on the GPU was much slower than rasterization, which the GPU was designed for. Many attempts were made by Christen (Christen, 2005) and Buck (Buck, 2004) to speed up GPU ray tracing, but speeding up hardware designed for one purpose to perform another proved very difficult. Purcell's own implementation made heavy use of dependent texture fetches (Purcell, 2004), as do many other GPU acceleration structures. These types of texture lookups use the result of one lookup to perform another in series, and are critical to GPGPU applications like ray tracing. This access pattern is poorly optimized on current GPUs because this is not a feature traditionally used in raster graphics.

A current solution proposed by Woop (2005) does not try to improve these computations on the GPU, but rather offload them from the GPU all together. Creating a custom piece of hardware, Woop described a specialized circuit designed to help the GPU perform tasks like *k*d-tree traversals. While this solution does work, integrating such a circuit into current graphics card designs will not be easy. The cards already have very little room to place large new components, and the circuit is not very general. It solves only the *k*d-tree traversal problem.

McCool (1999) proposed an alternative solution to this problem; adding another fully programmable shader to the texture unit, much like existing vertex and fragment shaders (McCool, 1999). This would allow for generality in operation, but would cost a lot in terms of chip area and operating speed.

Here we propose an alternative to McCool's design by adding texture unit programmability only for indirect texture lookups. Because it is not a general solution it takes less space and optimizes one of the main bottlenecks. Indirect lookups are used in almost all GPGPU applications (Buck, 2004) and the cost in terms of chip area would not be large because existing texture unit components can be leveraged to support these optimized accesses.

Chapter 3: Methods

To design the customized texture unit, I needed a schematic of the current texture unit. However, because card architecture details are proprietary, designing hardware components to integrate with current graphics hardware is difficult. For competitive reasons card manufacturers like ATI and nVidia do not release details of their cards' implementations. I therefore created a model based on empirical observations of the card's functionality. This model would then be used as the base to build a unit capable of performing custom filtering functions in the texture unit.

Graphics card texture units have several different modes that affect how texture memory is accessed. With these access modes, called filtering functions, the card can change how texture memory is processed before it returns to the fragment shader. The four filtering functions are nearest, bilinear, trilinear, and anisotropic filtering. Since the texture unit can perform all four of these functions, there must already be hardware in place to switch between these operations. To do this the card must already have a certain degree of flexibility and reconfigurability. My approach was to leverage this existing mechanism for customization and add support for indirect texture filtering.

To design the initial model I started with the nearest filter. This is the simplest filtering option and consists of a multiplication and an addition step followed by a memory lookup. Bilinear filtering takes four samples from the texture and bilinearly interpolates between the values. In determining a probable design for this filter I discovered that the circuitry for the nearest filter is merely a subset of the bilinear filter (Figure 6). The bilinear filter performs four nearest filters followed by the interpolation step. If correct, this design would explain why bilinear filtering is proportionally slower

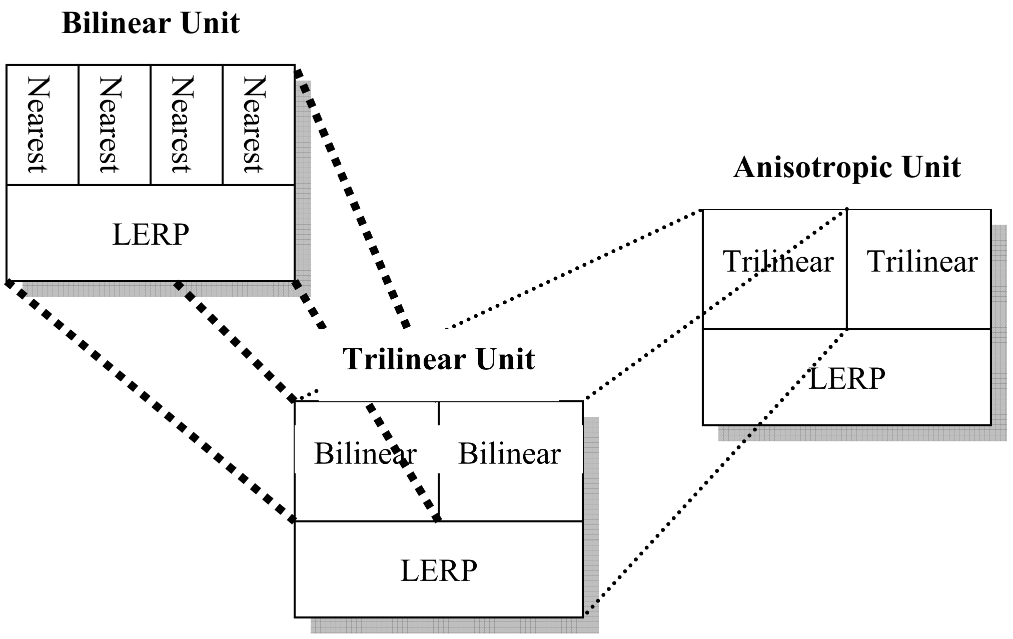


Figure 6 – Hierarchy of texture filtering operations (author)

than nearest filtering on a 3-D scene. Four nearest texture accesses could be performed in the same time as one bilinear access. Because the two filtering schematics overlap, they would scale easily to larger filtering units.

Trilinear filtering also fits easily into the model because it is two bilinear samples followed by interpolation. The hardware could use two of the bilinear filtering blocks to perform a single trilinear filter. This design would also be very easy to scale and make very efficient use of the existing resources. *Anisotropic filtering* fits into the schematic in much the same way. The final design breaks the texture unit into anisotropic filtering blocks which can be used differently depending on the situation. The same block could be used to perform one anisotropic sample, two trilinear samples, four bilinear samples, or 16 nearest samples. This design accounts for the proportional performance delay between different filtering methods. The block also shows the high degree of circuit reuse production card manufacturers want, and is highly scalable.

Once the base hardware model was in place, I analyzed the design, modifying it to and add the custom functionality. Since indirect texture lookups very often use nearest filtering to retrieve discrete texture values, I used that as the starting point to optimize the architecture. An indirect texture lookup uses the value stored in one texture as the coordinates to lookup another value in another texture (Figure 7). In this example the red and green components in the primary texture specify the row and column to lookup in the secondary texture. In a ray tracing application, the three values in the primary texture might represent the three vertices of a triangle and the secondary texture contains the actual (x, y, z, w) coordinates of those vertices.

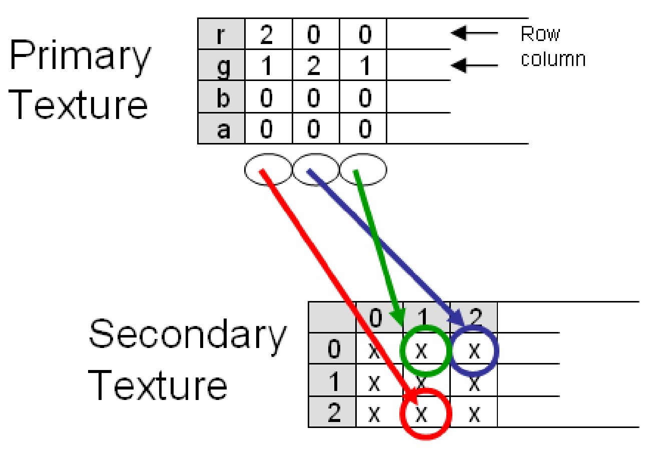


Figure 7 – Indirect Texture Lookup (author)

The model showed that a large amount of multipliers and adders, which would be used for the LERP operations, are free when doing nearest filtering (see Appendix C). The result of texture lookups could also be fed back into the filtering unit allowing indirect accesses without the roundtrip to the shader program.

Several factors guided the additions to the base texture unit model. The most important was speed. Texture accesses take a large amount of time during normal shader execution time so any changes should increase this delay as little as possible. I examined the various paths through the circuit, selecting those which were unused. Those components were reused in different ways instead of adding many more arithmetic units into the design. Reutilizing existing parts of the circuit would decrease design time, keep area to a minimum, and save power. Second most important was area on the chip. I needed to be sure that any extra components did not add excessively to the circuit's real estate requirements on-chip since space is already very limited. Multipliers and adders take the most space of any logic element, so I opted to reuse as many existing components as possible before adding new ones.

The third design constraint was power consumption. The improved texture unit uses less power because of fewer roundtrips between the texture unit and the shader program are required. This power savings should not be lost by adding additional paths into the texture unit. I therefore reused circuitry that was already there, but idle during nearest filtering. That hardware can be put to other uses during the time it would take to do one nearest filtering operation. Using these criteria an indirect texture lookup component was implemented on top of the existing texture unit model.

Chapter 4: Results

4.1 Summary of Results

The final texture unit design successfully supports indirect texture lookups as well as all original texture filters. These additions, however, have a minimal effect on the original functions' operating speed. By changing the control unit to handle texture indirection separately, control paths for normal nearest and bilinear filtering operations are not affected. With the control unit, the texture unit can indirectly lookup a maximum of four indirect values in one pass. These values can then be returned individually to the shader, or can be combined to form their sum, product, or average. These additional operations are controlled by the control signals listed in Table 1. Since the texture unit can tell in advance whether it will have to perform an indirect lookup, these control signals need only be sent when one is performing.

Name	**Bits**	**Description**	
Operation	2	Aggregate operation to perform on final values:	
		00:	Nothing, return individual values
		01:	Sum
		10:	Product
		11:	Average
Iterations	2	Number of additional values to lookup besides initial coordinate	
Offset - u	16	Offset added to initial u coordinate to compute multiple u values	
Offset - v	16	Same as above but for v	
u field	2	Specifies which component of direct lookup is used for the u coordinate in the indirect lookup:	
		00:	Use the "r" component
		01:	Use the "g" component
		10:	Use the "b" component
		11:	Use the "a" component
v field	2	Same as above but for v	
Total:	**40 bits = 5 bytes**		

Table 1 – Indirect Lookup Control Signals

These input control signals come from the shader program. Three additional instructions added to the *ISA* allow these parameters to be sent (see Appendix E). The control unit uses these signals to correctly animate the datapath to perform the correct operation. Outputs from the datapath are then fed back into the control unit for processing. Those outputs would normally flow back to the shaders. In the case of an indirect lookup, however, those values are given back to the datapath to perform the second dependent texture lookup. The high-level view of this process shows how the control unit and datapath are linked (Figure 8).

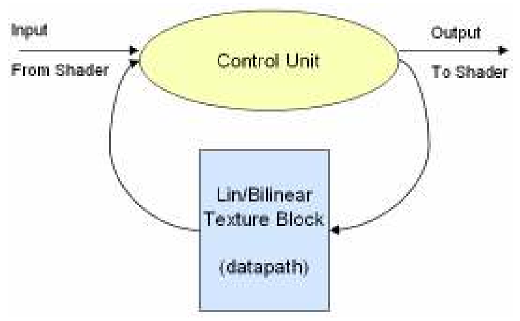


Figure 8: Filtering unit data flow (author)

The control unit has to be changed to allow for the new signals outlined in Table 1 and to perform new logic to handle indirect lookups (Figure 9). After receiving an indirect lookup request, the state machine transitions from its initial "Wait" state to "Indirect Calc." This state computes all the texture values for the primary texture in parallel. These include the original texture coordinates (u, v) as well as all additional values computed through multiples of the offset control signal: $(u + o_u, v + o_v)$, $(u + 2o_u, v + 2o_v)$, and $(u + 3o_u, v + 3o_v)$. The iterations control signal sets the exact number of coordinates that will be used. Figure 7 in the previous chapter was an example where iterations = 2, and the coordinate offsets are $o_u = 1$ and $o_v = 0$.

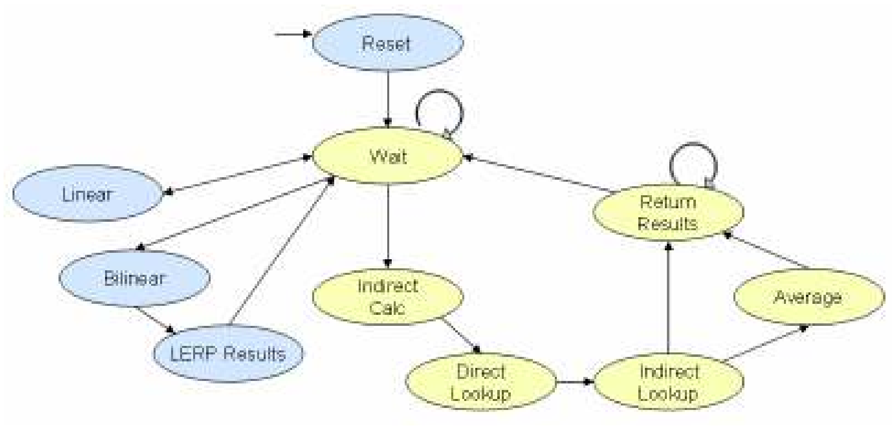


Figure 9: Bilinear unit control unit state machine, changes in yellow (author)

The control unit then moves to the “Direct Lookup” state where the coordinates are passed to the datapath. Output pixel values from the primary texture are sent back to the control unit. Control then moves to the “Indirect Lookup” state. The values just calculated are treated as the texture coordinates that are sent back into the datapath. The next output values from the datapath are the indirectly accessed values we wanted. In the case of an aggregate operation (sum, product, or average), the values are combined before being sent to the control unit. Then, if the operation control signal is an average, control passes to the “Average” state, otherwise control moves to “Return Results.” The average state immediately transfers to the results state after dividing the intermediate sum by the number of elements. The results state sends back the final results to the shader that originally requested the texture lookup. This could either be a single value in the case of a sum, product or average operation, or each individual value. Finally, control moves back to “Wait” to await the next filtering operation.

Changes to the datapath's main components are minimal. The goal was to reuse as many components as possible so as not to increase the area requirements of the texture unit. The three LERP blocks at the bottom of the datapath (see Appendix A) were reused because they contain two adders and two multipliers apiece. These LERP blocks (Figure 10) are not used during nearest filtering operations, which means all six adders and six multipliers can be reused for other purposes. The figure below shows the addition of several multiplexers to allow the LERP blocks to perform the calculations in the "Indirect Calc" stage discussed above. By altering these LERP blocks, the only additional datapath components added were three adders used for the texture coordinate calculation step, and one multiplier used for computing the aggregate average (see Appendix D). These four new components take up less space than a single LERP block, which is proportionally very small compared to the size of the bilinear block. The appendix contains full diagrams of all components and altered LERP blocks.

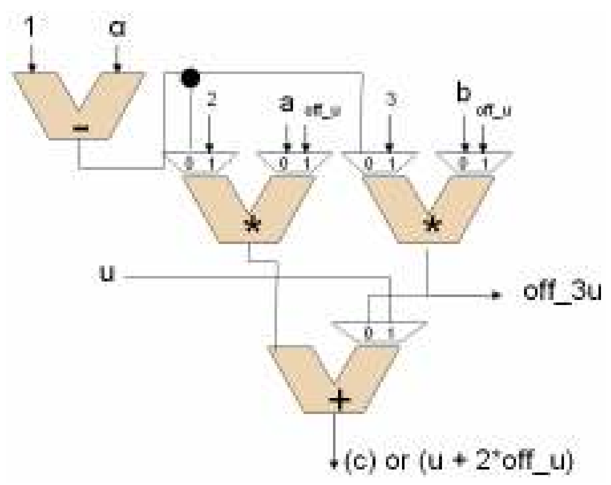
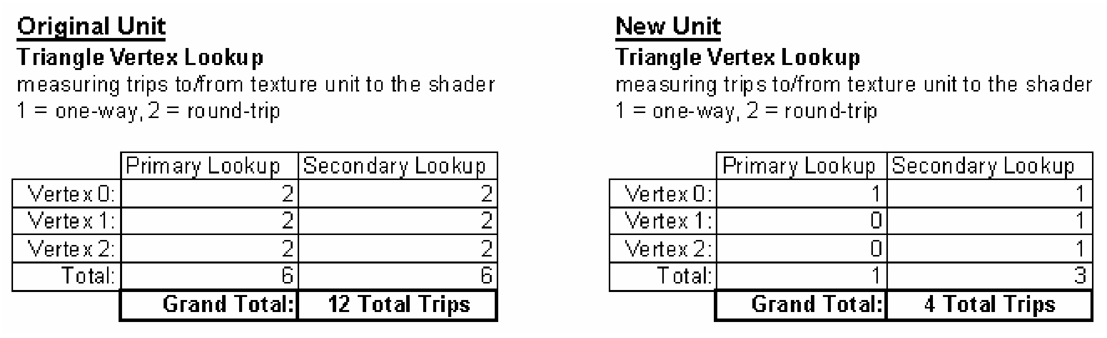


Figure 10: LERP with changes to also allow computation of coordinate offsets (author)

4.2 Discussion

These results show that texture units used for multiple indirect accesses can be optimized with few modifications. Execution time for various applications of indirect lookups show a large performance increase (Figure 11). Using indirect texture accesses would operate with three times fewer trips as compared to the unmodified layout. Each saved trip results in less memory bandwidth used and a speed increase since the texture



Original Unit
Triangle Vertex Lookup
measuring trips to/from texture unit to the shader
1 = one-way, 2 = round-trip

	Primary Lookup	Secondary Lookup
Vertex 0:	2	2
Vertex 1:	2	2
Vertex 2:	2	2
Total:	6	6
	Grand Total:	**12 Total Trips**

New Unit
Triangle Vertex Lookup
measuring trips to/from texture unit to the shader
1 = one-way, 2 = round-trip

	Primary Lookup	Secondary Lookup
Vertex 0:	1	1
Vertex 1:	0	1
Vertex 2:	0	1
Total:	1	3
	Grand Total:	**4 Total Trips**

Shows 33% Reduction

Figure 11 - Indirect Triangle Vertex Lookup Comparison (author)

unit does not have to wait for the texture unit to perform the next access. Normal texture accesses will not be slowed because the indirection circuitry is simply bypassed when not in use.

Extra control data (Table 1) must be sent with each texture access, however, specifying whether an optimized indirection should happen and what its parameters are. Yet these additional pieces of data are worth the savings in memory bandwidth and power consumption. Just one saved roundtrip to the texture unit saves at least 28 bytes of bandwidth. The original unit, in this example, would consume around 168 bytes of memory bandwidth. The new unit use only 52 bytes. That is a bandwidth savings of 31 percent, which also translates into a proportional power savings.

Ray tracing performance on such a design benefits greatly. Appendix F shows a triangle intersection fragment shader from an open source ray tracer (Christen, 2005). When run, the shader will execute approximately 49 total instructions. Nine of those instructions, 18%, are texture fetches. The function *intersect_check* tests the possible intersection of a ray and a triangle by first retrieving the triangle's three vertices through a series of dependent, indirect texture lookups. The result of the first lookup is treated as a coordinate with an offset and used in three secondary texture fetches. These four texture operations can be reduced to just one using the techniques introduced previously. In this particular code sample, changing those lines to use the functions described in Appendix E, would allow for many of the arithmetic calculations done in that section to be performed beforehand and simply baked into the lookup texture. These optimizations approximately decrease the instruction count by ten, a 20% decrease. The total number of texture operations would also drop by almost 33%, from nine to six.

Other applications besides ray tracing will benefit greatly from the ability to sum, multiply, or average across accessed pixels. This is a very common operation known as *gather* and is used by many image-processing algorithms and scientific applications. Performing gather in hardware instead of the shader will save on bandwidth, because the pixels are combined inside the texture unit, and the results sent to the shader only when completed. With this parallelism, the shader is free to possibly execute other instructions while waiting for the results, thus increasing efficiency. While not directly helpful in ray tracing, this aggregating ability would be useful in many scientific applications, in computer vision and graphics in general.

4.3 The Next Step

To implement the design improvements discussed previously, researchers should refine these results and eventually presenting them to graphics card manufacturers such as nVidia and ATI. Card makers are always looking for ways to speed execution and enhance the features their cards can offer. These companies also have the infrastructure in place to manufacture this design.

My design tried to reuse components that were already on-chip, but a much more extensive redesign may give better performance metrics. Integrating components such as morphable-multipliers into the LERP design of bilinear blocks would be an interesting extension. These morphable blocks are capable of performing either addition or multiplication and are 17 percent smaller than implementing each unit separately (Dale, et al, 2006). Even more advanced filtering functions would be possible with these more customizable building blocks. With the correct components, aggregate functions such as min and max would be possible.

The results of this thesis should also be integrated with the recent introduction of CUDA, a new programming interface developed by nVidia. With CUDA, graphics programmers can treat the GPU as a CPU by writing code in C instead of shader languages. They can thereby make nVidia cards perform scientific applications. CUDA allows programmers to treat memory linearly or as texture maps. Data for GPGPU applications would no longer need to be stored and processed into a texture, since CUDA would abstract that away. Obviously the data must still be stored in textures because normal texture accesses are still allowed; the difference is that this is now abstracted from the programmer. If this technology becomes the future of graphics cards, their modified texture unit would be architecturally different than the one developed here. Researchers would therefore have to look further into how CUDA enabled texture units work to adapt the techniques discussed here to this new architecture.

References

BBC News. (2000). “Video Games: Cause for Concern?”. *BBC News, World Edition*. Nov. 26, 2000. <http://news.bbc.co.uk/2/hi/uk_news/1036088.stm>

Blinn, J., Newell, M., (1976). ‘‘Texture and Reflection in Computer Generated Images’’, *CACM*, (19):10, Oct. 1976, pp. 542-547.

Blinn, J., (1978). ‘‘Computer Display of Curved Surfaces’’, *PhD thesis*, CS Dept., University of Utah.

Buck, I., Purcell, T., (2004). “A Toolkit for Computation on GPUs.” *GPU Gems*. Ed. Randima Fernando. Boston, MA: Addison-Wesley, 2004, pp. 621-636.

Carlson, W. (2006). “History of Computer Graphics and Animation”, Winter, 2006. <http://accad.osu.edu/~waynec/history/>

Catmull, E., (1974). “A Subdivision Algorithm for Computer Display of Curved Surfaces”, *PhD thesis*, Dept. of CS, U. of Utah, Dec. 1974.

Christen, Martin. (2005). Ray Tracing on GPU. University of Applied Sciences Basel (FHBB), Diploma Thesis. January 19, 2005. <http://olivier.nocent.free.fr/papers/christen05.pdf>

Cook, R., (1984). ‘‘Shade Trees’’, *Computer Graphics, (SIGGRAPH ’84 Proceedings)*, 18(3): July 1984, pp. 223-231.

Dale, K., Sheaffer, J., Vijay Kumar, V., Luebke, D., Humphreys, G., Skadron, K.. (2006) "Applications of Small-Scale Reconfigurability to Graphics Processors." In *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC2006)*, Springer-Verlag LNCS, Mar. 2006.

Foley, T., Sugerman, J. (2005). KD-tree acceleration structures for a GPU ray tracer. SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware. <http://portal.acm.org/ft_gateway.cfm?id=1071869&type=pdf&coll=GUIDE&dl=GUIDE&CFID=3598999&CFTOKEN=78745218>

Fujimoto, A., Takayuki, T., Kansei, I.. (1986). “Arts: Accelerated ray-tracing system”, *IEEE Computer Graphics and Applications*, 6(4): pp. 16–26.

Game Revolution. (2005). “The Truth About Violence Youth and Video Games.” October 15, 2005. <http://www.gamerevolution.com/oldsite/articles/violence/violence.htm>

Glassner, A., (1984). “Space subdivision for fast ray tracing”, *IEEE Computer Graphics and Applications*, 4(10): pp. 15–22.

Gouraud, H. (1971). “Continuous shading of curved surfaces”, *IEEE Transactions on Computation*. C-206, June 1971, pp. 623–629.

Gunther, J., Friedrich, H., Wald, I., Seidel, H., Slusallek, P. (2006). “Ray tracing animated scenes using motion decomposition”, *Proceedings of Eurographics, Computer Graphics Forum*, 25(3): September 2006.

Hachman, M., (1996). “Powerful Voodoo from 3dfx”, *Electronic Buyer's News*, No. 1034, Nov. 25, 1996.

Hinton, G., Upton, M., et al. (2001). “A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit.” IEEE Journal of Solid-State Circuits, 36(11):1617-1627.

Hook, D., (1995). “Using kd-trees to guide bounding volume hierarchies for ray tracing”, *Australian Computer Journal*, 27(3): Aug. 1995, pp. 103-108.

Lasseter, J. (2007). “The Cars Interview.” FutureMovies Interview. <http://www.futuremovies.co.uk/filmmaking.asp?ID=182>

McCool, M., (1999). “Texture Shaders”, *Proceedings of EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*, 1999, pp.117-126.

McCool, M. (2001). “Shadow Volume Reconstruction from Depth Maps”, *ACM Transactions on Graphics*, January 2001, pp. 1-25.

Montrym, J., Moreton, H. (2005). “The GeForce 6800”, *Micro, IEEE*. (25): pp. 41-51.

nVidia. (2007). “GeForce3”. *nVidia Graphics Products*, <http://www.nvidia.com/page/geforce3.html>

Phong, B., (1975). “Illumination for computer generated pictures”, *Communications of the ACM*, 18(6): June 1975, pp. 311–317.

Pixar. (2006). Lightning McQueen. <http:/adisney.go.com/disneyvideos/animatedfilms/cars/main.html?sec=1&car=>

Purcell, T., (2004). “Ray Tracing on a Stream Processor”, *Ph.D. dissertation*, Stanford University. March 2004.

Shrout, Ryan. (2005). “nVidia GeForce 7800 GTX GPU Review.” PC Perspective. June 22, 2005. <http://www.pcper.com/article.php?aid=150&type=expert>

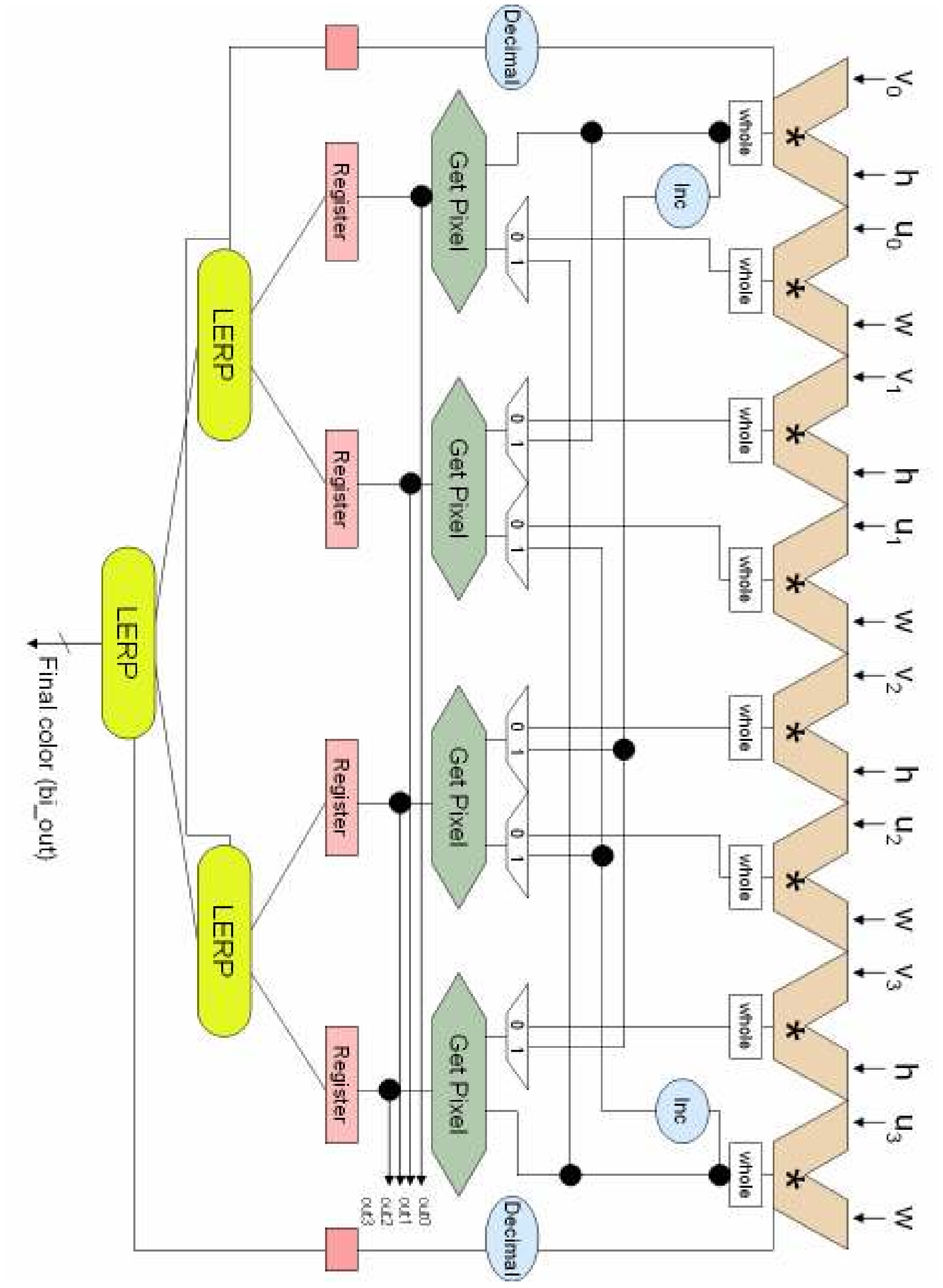
Szirmay-Kalos, L., Havran, V., Balazs, B., Szecsi, L., (2002). “On the Efficiency of Ray-Shooting Acceleration Structures”, *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, Spring 2002, pp. 97-106.

Whitted, T. (1980). “An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6): pp. 343-349.

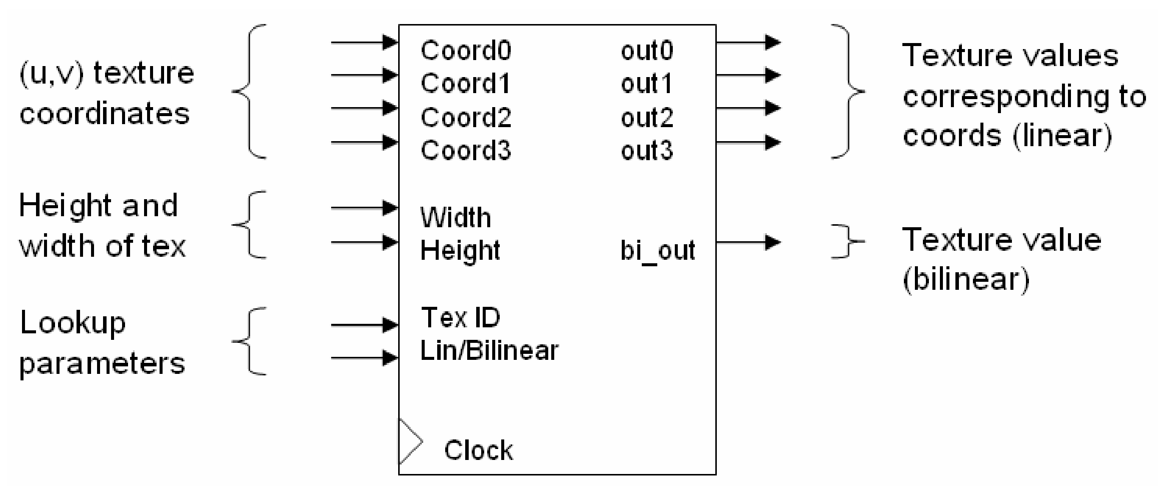
Woop, S., Schmittler, J., Slusallek, P. (2005). “RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing”, *SIGGRAPH 2005*.

Wylie, C., Romney, G., Evans, D., and Erdahl, A., (1967) "Halftone Perspective Drawings by Computer," Proc. AFIPS FJCC, 1967, 31(49).

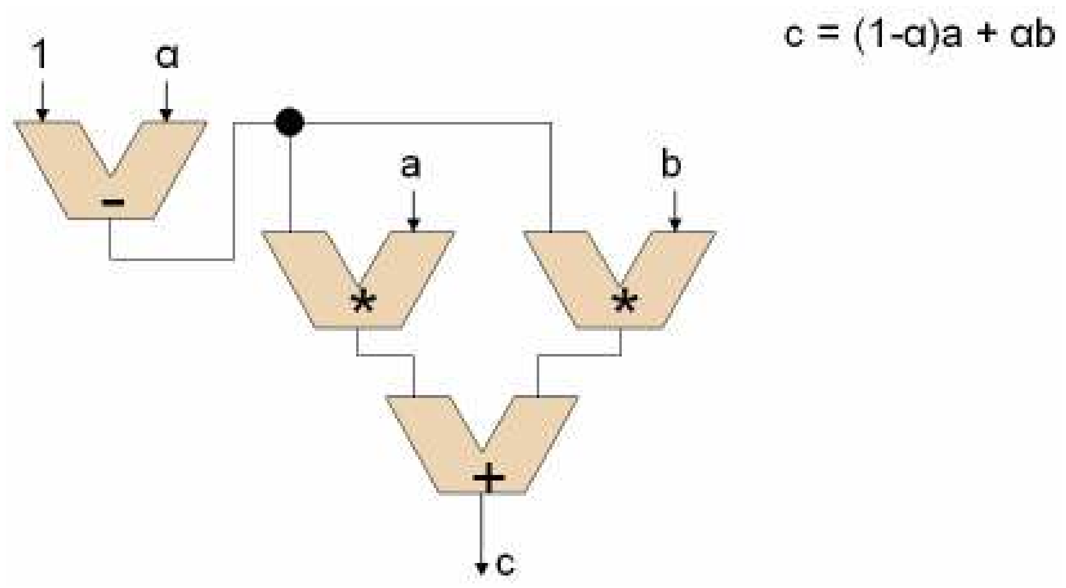
Appendix A: Datapath Schematic



Appendix B: Bilinear Filtering Unit Schematic Symbol:



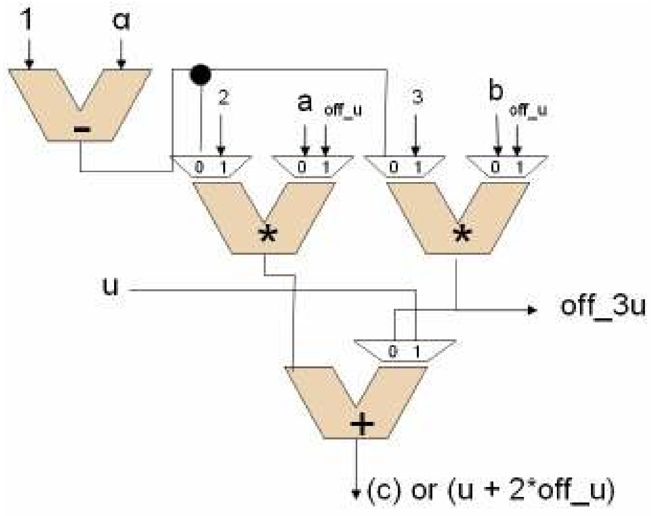
Appendix C: Linear Interpolation (LERP) Unit



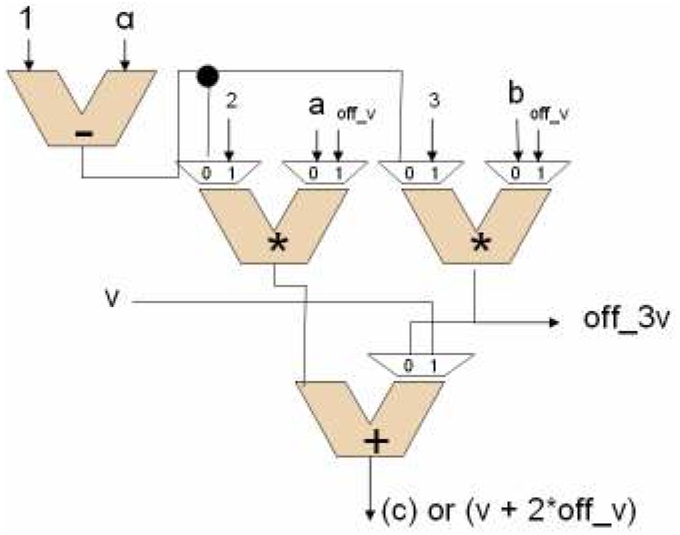
Appendix D: Altered LERP Units

The following three schematics are the altered LERP units that would be inserted into the datapath shown in Appendix A, and also the three additional adders required.

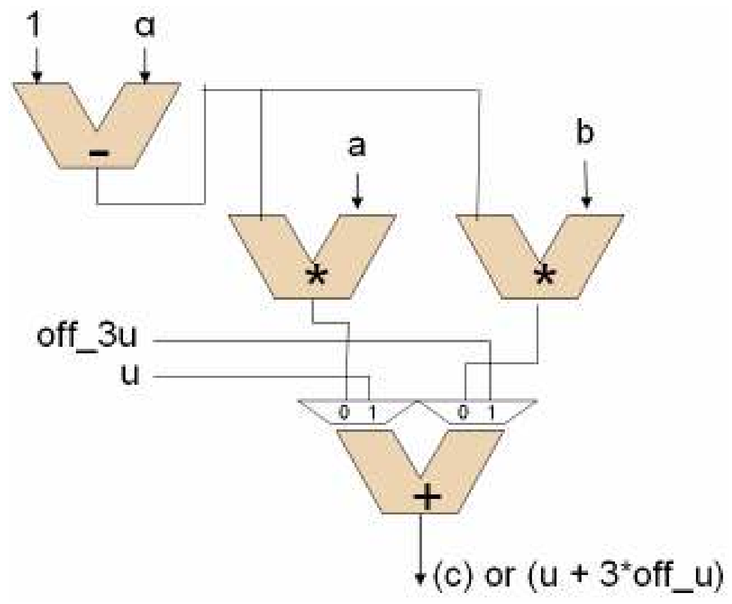
Calculates (u + 2*offset_u):



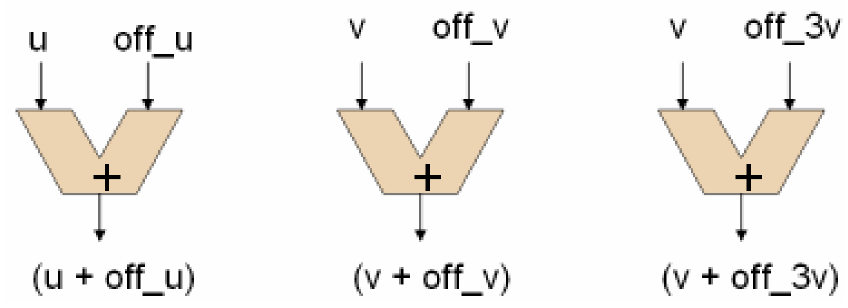
Calculates (v + 2*offset_v):



Calculates (u + 3*offset_u):



The three additional adders needed to calculate all the primary texture coordinates:



Appendix E: ISA Modifications

Three instructions, similar to the following, would need to be added to the shader ISA to pass all required parameters for indirect lookups. These functions are not written for a specific shading language, but should be adaptable to Cg, HLSL, or any other. One possible optimization is to make *texSetupIndirect* a function called only once by the graphics API prior to loading the fragment shader. This way, the setup information will only be transferred once and can be used by all subsequent executing shader units.

Function Name:	texSetupIndirect	
Return Type:	void	
Prototype:	void texSetupIndirect(int operator, int iterations, int2 offset, int swizzle)	
Arguments:	int operator	Enumeration determining which aggregate operation to perform: 0 – None (all secondary values will be returned) 1 – Sum (only the sum returned) 2 – Product (only the product returned) 3 – Average (only the average returned)
	int iterations	Number of primary texture values to read by applying the offset to the base primary texture coordinates, range: 1-4
	int2 offset	Vector of two numbers holding the offset of u and v, respectively, which will be added to the base texture coordinates *iterations* times
	int swizzle	Enumeration specifying how retrieved primary texture values should be interpreted as secondary coordinates: RR, RG, RB, RA GR, GG, GB, GA BR, BG, BB, BA AR, AG, AB, AA Ex: If the primary texture value retrieved is (3, 7, 1, 2) and *swizzle = AG*, then the coordinate

		looked up in the secondary texture will be *(u,v) = (2, 7)*
Description:	This function should be called first before any indirect texture lookups are performed. This sets up the require state for subsequent indirect lookups.	

Function Name:	texIndirect	
Return Type:	float4	Returns the result of the indirect lookup, if multiple values are expected this is the first of those values
Prototype:	float4 texIndirect(texID primaryTexID, int2 coord, texID secondaryTexID)	
Arguments:	texID primaryTexID	Identifier specifying the primary texture used in the lookup
	int2 coord	A vector of two numbers storing the (u, v) coordinates corresponding the primary texture used, this is the base point from which all offset calculations will be started
	texID secondaryTexID	Identifier specifying the secondary texture used in the lookup
Description:	This function actually performs the indirect lookup. If an aggregate operation was specified in *texSetupIndirect* then this will return the first value. Use *texIndirectValue* the retrieve subsequent data values	

Function Name:	texIndirectValue	
Return Type:	float4	Returns the next indirect value calculated by a previous call to *texIndirect*
Prototype:	float4 texIndirectValue( )	
Arguments:	none	Identifier specifying the primary texture used in the lookup
Description:	Returns the next value in the list of values calculated by *texIndirect.* This should be called *(iterations – 1)* times after the initial call to *texIndirect* to retrieve all calculated values.	

Example: Retrieve all three vertices of a triangle where vertex indices are in adjacent columns starting at (10, 0) in the R and G positions of the color in texture1. Vertices are located in texture2.

```
// Setup called from the graphics API (OpenGL, etc.)
int2 vertexOffset = (1, 0)
texSetupIndirect( NONE, 3, vertexOffset, RG )



// Inside the shader program
int2 primaryCoord = (10, 0)

float4 vertex1 = texIndirect( tex1, primaryCoord, tex2 )
float4 vertex2 = texIndirectValue( )
float4 vertex3 = texIndirectValue( )
```

Appendix F: GPU Ray Tracer Example

Below is an excerpt from the ray intersection fragment shader from an open source ray tracer (Christen, 2005). This shader is run when calculating ray intersections between triangles in a uniform grid accelerated scene. The indirect lookups occur in the *intersect_check* function in the first seven lines.

```
//-----------------------------------------------------------------------------
// Kernel 03: Ray Intersect
// Version 1.0 - GLSL
// Author: Martin Christen, christen@clockworkcoders.com
//-----------------------------------------------------------------------------
// Input Arrays:      Texture0: Rays
//                    Texture1: Position
//                    Texture2: Voxel + Voxel-Number
// Input Data:        Voxel Index
//                    Grid Element
//                    Element Data
// Output Arrays:     Texture1: Raypos
//                    Texture2: Voxel
//                    Texture4: uvt
// Entry Condition:   emtpyflag >= 0 [=triangle number]
// Discard Condition: emptyflag < -0.5
//-----------------------------------------------------------------------------
```

```
#define TRIANGLE_ELEMENTS  2     // Wert muss vom C++ Programm übernommen
werden.
#define inbox(v, b1, b2) ((v.x>=b1.x) && (v.y>=b1.y) && (v.z>=b1.z) &&
(v.x<=b2.x) && (v.y<=b2.y) && (v.z<=b2.z))
#define INTEGER(x) (x-fract(x))
#define _EPSILON_   1e-15
#define _INF_       1e20

#define TextureDef      samplerRECT
#define TexLookup       texRECT

uniform vec3 g1;          // Grid Grösse (min max Vektoren der Bounding Box)
uniform vec3 _len;        // Länge der Zelle (x-,y- und z-Richtung)
uniform float GEsize;     // Grösse der Grid-Element-Textur
uniform float EDsize;     // Grösse der Element-Daten-Textur

uniform TextureDef tex2;
uniform TextureDef tex3;
uniform TextureDef tex4;
uniform TextureDef tex1;
uniform TextureDef GE;
uniform TextureDef ED;
uniform TextureDef tex0;

struct Ray
{
   vec3 startpoint;
   vec3 direction;
};

struct Triangle
{
   vec3 A;
   vec3 B;
   vec3 C;
};


//--------------------------------------------------------------------------------
//Triangle Intersection
//
// return value
//   ret.x --> barycentric u
//   ret.y --> barycentric v
//   ret.z --> Hitpoint t (ray)
//   ret.w --> reserved for triangle num
float intersect(inout Ray r, inout Triangle tri, inout vec4 ret)
{
   vec3 edge1, edge2, tvec, pvec, qvec;
   float det;
   float res = -1.0;
   ret = vec4(0.0,0.0,0.0,0.0);
   qvec = vec3(0.0,0.0,0.0);

   edge1 = tri.B-tri.A;
   edge2 = tri.C-tri.A;

   pvec = cross(r.direction, edge2);
   det = dot(edge1, pvec);

   if (det > _EPSILON_)
   {
      res = 1.0;
      tvec = r.startpoint - tri.A;
      ret.x = dot(tvec, pvec);
      if (ret.x < 0.0 || ret.x > det) res = -1.0;
      if (res > 0.0)
      {
         qvec = cross(tvec, edge1);
         ret.y = dot(r.direction, qvec);
         if ((ret.y < 0.0) || ((ret.x + ret.y) > det)) res = -1.0;
      }
   }

   if (det < -_EPSILON_)
   {
      res = 1.0;
      tvec = r.startpoint - tri.A;
      ret.x = dot(tvec, pvec);
      if (ret.x > 0.0 || ret.x < det) res = -1.0;
      if (res > 0.0)
      {
         qvec = cross(tvec, edge1);
         ret.y = dot(r.direction, qvec);
         if ((ret.y > 0.0) || ((ret.x + ret.y) < det)) res = -1.0;
      }
   }

   if (res > 0.0)
   {
      ret.z = dot(edge2, qvec);
      ret /= det;
   }

   if (ret.z<_EPSILON_) res = -1.0;

   return res;
}

vec3 voxel2world(vec3 xyz)  { return xyz * _len + g1;}

void voxelbounds(vec3 ijk, out vec3 v1, out vec3 v2)
{
    v1 = voxel2world(ijk);
    v2 = v1 + _len;
}
vec2 Coord2D(float listnumber, float texturesize)
{
vec2 res;
   res.y = listnumber/texturesize;
   res.y = INTEGER(res.y);
   res.x = listnumber-texturesize*res.y;
   return res;   //texturesize;
}

vec2 ElementCoord2D(float listnumber, float texturesize, float align)
{
vec2 res;
   //for non-normalized texture coordinates:
   //--------------------------------------
   float a = texturesize / align;
   res.y = listnumber / a;
   res.y = INTEGER(res.y);
   res.x = listnumber - a*res.y;
   res.x *= align;
   return res;
   //
   //for normalized texture coordinates:
   //-----------------------------------
   /* float align_size = texturesize / align;
   res.y = listnumber/align_size;
   res.y = INTEGER(res.y);
   res.x = listnumber-align_size*res.y;
   res.x /= align_size;    // auf [0,1] skalieren
   res.y /= texturesize;   // auf [0,1] skalieren
   return res;
   */
}

void intersect_check(inout Ray r, inout Triangle tri, inout vec4 raydir, inout
vec4 raypos, inout vec4 voxel, inout vec4 uvt)
{
   vec2 GEcoord = Coord2D(voxel.w, GEsize);  // Koordinaten für Grid Element
List
   float elementnum = TexLookup(GE, GEcoord).x;   // Element-Nummer für EDlist.

   bool c0=false, c1=false, c2=false, c3=false, c4=false;
   vec4 intersect_result = vec4(0,0,0,0);

   vec2 EDcoord = ElementCoord2D(elementnum, EDsize, (TRIANGLE_ELEMENTS*4.0));

   tri.A = vec3(TexLookup(ED, EDcoord));
   tri.B = vec3(TexLookup(ED, vec2(EDcoord.x+2.0,EDcoord.y)));
   tri.C = vec3(TexLookup(ED, vec2(EDcoord.x+4.0,EDcoord.y)));

   bool p0 = bool(elementnum == -1.0);
   bool p1 = bool(uvt.w == -1.0);
   bool p2 = bool(uvt.w >= 0.0);

   if (p0 && p1) // FAIL -> Traversierung fortsetzen
   {
         voxel.w = -10.0;
         uvt.xyz = vec3(0.0,0.0,_INF_);
   }
   else if (p0 && p2) // DONE -> ein Schnitt wurde gefunden
   {
         voxel.w = -20.0;
   }

   if (elementnum >=0.0)
   {
      voxel.w = voxel.w + 1.0; // nächstes Element (für nächsten Durchgang)

      float intersecttest = intersect(r,tri, intersect_result);

      // calculate voxel bounds
      vec3 vb1 = voxel.xyz * _len + g1;    //voxelbounds(voxel.xyz, v1, v2);
      vec3 vb2 = vb1 + _len;

      vec3 hitpoint = raypos.xyz + raydir.xyz*intersect_result.z;

      c0 = inbox(hitpoint, vb1,vb2);
      c1 = bool(uvt.z >= intersect_result.z);
      c2 = bool(intersecttest == 1.0);
      c3 = bool(elementnum != raypos.w);   // gleiches Element darf nicht 2x
getestet werden.
      c4 = bool(c0 && c1 && c2 && c3);
   }
   if (c4)
   {
     uvt.xyz = intersect_result.xyz;
     uvt.w = elementnum;
     raypos.w = elementnum;
   }


}

void main(void)
{
   vec4 voxel =  TexLookup(tex2,gl_TexCoord[0].xy);
   vec4 tMax  =  TexLookup(tex3,gl_TexCoord[0].xy);
   vec4 uvt   =  TexLookup(tex4,gl_TexCoord[0].xy);
   vec4 raypos = TexLookup(tex1,gl_TexCoord[0].xy);
   vec4 raydir = TexLookup(tex0,gl_TexCoord[0].xy);


   Triangle    tri;
   Ray         r;
   r.startpoint = raypos.xyz;
   r.direction = raydir.xyz;

   tri.A = vec3(0,0,0);
   tri.B = vec3(0,0,0);
   tri.C = vec3(0,0,0);

   if (voxel.w < 0)  // Voxel nicht in {wait} Zustand
   {
      discard;
   }

   intersect_check(r, tri, raydir, raypos, voxel, uvt);
   //if (voxel.w >=0) intersect_check(r, tri, raydir, raypos, voxel, uvt);
   //if (voxel.w >=0) intersect_check(r, tri, raydir, raypos, voxel, uvt);
   //if (voxel.w >=0) intersect_check(r, tri, raydir, raypos, voxel, uvt);


   gl_FragData[0] = voxel;
   gl_FragData[1] = tMax;
   gl_FragData[2] = uvt;
   gl_FragData[3] = raypos;

}
```