Smart Texture Magnification Filtering

Robert Jr Ohannessian

Abstract - **The project consists of a simple scanline rasterizer coupled with a texture unit implementing the SmartFlt magnification algorithm. In typical Graphics Processing Units, the best magnification texture filter available is a simple bilinear filter. Unfortunately, bilinear filtering leaves the texture blurry and diminishes the perceived detail of the image. SmartFlt, on the other hand, recognizes patterns in the texture and will adapt the filtering algorithm to better highlight contrast and detail in the original texture.**

I. INTRODUCTION

Texture mapping is a technique used to add detail to 3D models. It can be loosely described as "pasting an image onto a triangle". Essentially, each vertex[1] of a 3D mesh[2] is assigned texture coordinates. Those coordinates are interpolated for each fragment[3] of the triangle. Finally, the interpolated coordinates are used to index an image (or texture), and the color value that was looked-up is shaded and drawn in a frame buffer[4].

Magnification occurs when several pixels get mapped to the same texel[5]. That is, the texture does not have enough precision to accurately describe the detail it was meant to, for some set of pixels.

In this paper we make the simplifying assumption that all textures are two dimensional arrays of four-component color values. We will also assume that we are always in magnification, since this is the area of interest.

The interpolated texture coordinates are usually fractional; when indexing an image, we need to round the coordinate to some integer for computing the texel address in the texture. If we just round to nearest, each fragment will have the nearest texel corresponding to its texture coordinate assigned to it. This leads to images that are blocky: due to what is known as the "mach band effect", the boundary between texel colors is exaggerated.

To reduce this effect, textures are usually filtered. One such filtering scheme is bilinear filtering: For each texture coordinate, we look-up the four nearest texels. We then use the fractional parts of the texture coordinate to interpolate the texel values.

Bilinear filtering gives significantly better results than nearest filtering. However, when magnifying too much (more than two or three pixels per texel), the resulting image appears blurry, soft and lacking in detail, even though no detail is actually lost.

The Smart Texture Filtering (SmartFlt) addresses those concerns. SmartFlt manages to retain detail without much aliasing or blurring. As we will see, SmartFlt is almost as inexpensive as bilinear hardware wise, and is just as fast. There is one caveat however: the texture needs to be preprocessed first.

To demonstrate the feasibility of such a texture filter, we have built a simple scanline rasterizer implementing this algorithm. Real Graphics Processing Units (GPUs) are highly parallel, and need to operate at high clock rates. There is thus a need to have circuits be both small and fast, so that they may be replicated any number of times.

II. BACKGROUND

As we have mentioned above, magnification occurs when several pixels on screen get mapped to the same texel. To be precise, magnification occurs when $\lambda(x,y) < \frac{1}{2}$, where

$$\lambda(x,y) = \log_2(\rho(x,y))$$
$$\rho(x,y) = max(\rho_x(x,y), \rho_y(x,y))$$
$$\rho_x(x,y) = \sqrt{(\frac{\partial u}{\partial x})^2 + (\frac{\partial v}{\partial x})^2}$$
$$\rho_y(x,y) = \sqrt{(\frac{\partial u}{\partial y})^2 + (\frac{\partial v}{\partial y})^2}$$

where $\frac{\partial u}{\partial x}$ is the derivative of the u component of the texture coordinate with respect for the screen's x coordinate, and similarly for the other partial derivatives.

Essentially, when moving from one pixel to the next in any one dimension, we end up referring to the same texel. As mentioned earlier, without filtering, the resulting image looks blocky and aliased.

1 Point in n-dimensional space with corresponding attributes, such as color and texture coordinates
2 Set of triangles forming some 3D shape
3 Ccolor value, position, texture coordiante and other attributes placed together
4 Memory that contains the final rendered image. It is usually displayed on a monitor via a DAC
5 Texture Element

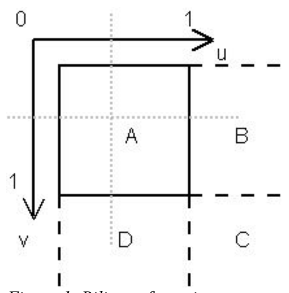


Figure 1. Bilinear footprint

Bilinear filtering is used to smooth out the color discontinuities, preventing the mach band effect. For each texture coordinate, we look-up the four nearest texels and interpolate their colors in two dimensions using the fractional parts of the texture coordinate as filter weights.

The texel color c_t is computed by the following equations:

$$c_t = lerp(lerp(A, B, u), lerp(D, C, u), v)$$

$$lerp(A, B, u) = (B - A) \times u + A$$

A, B, C, D are the four texels we interpolate on, and u and v are the fractional components of the 2D texture coordinate. This is performed for each color component of the four texels.

Typical GPUs perform these computations using floating-point numbers. In the interest of rapid development, we have opted for a fixed-point scheme instead. The various quantities use different arrangements of integer and fractional parts, as needed. Throughout this paper, we will refer to fixed-point formats in the following format: [u | s]i.f, where u means unsigned, s means signed, i is the number of bits in the integer part and f is the number of bits in the fractional part. For example u0.32 is an unsigned number with 0 bits of integer and 32 bits of fraction, yielding numbers in the range [0..1[.

IV. DESIGN

SmartFlt Algorithm

SmartFlt is an algorithm developed by Maxim Stepin to retain hard edges in textures when filtering. As previously mentioned, the texture needs to be preprocessed before having SmartFlt applied to it. The preprocessing step consists in determining, for each texel, which of the 14 patterns it corresponds to based on the neighboring texels. That is, we look at each 2x2 block of texel and try to match it some of the patterns.

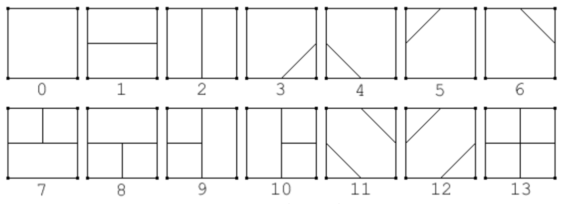


Figure 2. SmartFlt Texel Patterns

For example, if three of the four texels in a 2x2 block are yellow, and the fourth one is blue, then we select pattern 3. Note that the yellow shades in our example need not be identical. Indeed, the idea is that we still want to apply bilinear filtering on those texels that are of similar color so that we get rid of the banding. The patterns, on the other hand, are used to determine where the hard edges are.

Taking the case of pattern 3 again, the texture filter algorithm will determine which region the texture coordinate falls in based off its fractional parts. It will then apply one of the 3 filters, depending on the region, as shown in Figure 3.

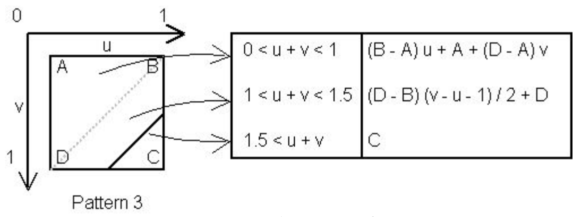


Figure 3. Regions and Equations for Pattern 3

The other patterns are handled similarly. Appendix A contains the complete list of equations we used.

The full SmartFlt algorithm actually uses over 200 patterns for a much better final image. In the interest of time, we have opted to implement just the basic SmartFlt algorithm.

Texture Storage Format

Since we need to store the pattern number for each texel, we could use a separate image, much like for mipmaps. However, for simplicity, we have opted to store the pattern number in the alpha channel of the texture. Thus, the alpha channel now contains just the SmartFlt pattern number instead of transparency information.

To preserve compatibility with the full SmartFlt algorithm, and in case we decide to implement the full algorithm at some later time, the pattern number only occupies the top four bits of the 8-bit alpha channel. The bottom four bits are reserved for future use, and are ignored in the current implementation.

Design Constraints

We have opted for a design clock rate of 100 MHz on a Xilinx Virtex II Pro, and have pipelined the architecture accordingly. Re-pipelining can always be done if a different target clock rate or implementation on an alternative FPGA or technology is desired.

The hardware was also designed to process a single pixel every clock cycle. Higher throughput can be achieved by placing several of the pipelines in parallel.

System Design

The system is conceptually comprised of three parts: A user application, a driver and the graphics hardware. In our case, we merged the driver and the user application into a single program. This test program will basically generate some geometry (several cubes randomly rotated), then transform every vertex in the scene. Perspective projection is then performed, followed by a computation of the screen-space edge equations of the triangles (also known as Setup). Finally, the driver generates a list of scanlines with their attributes via scan conversion and interpolation. The driver code is based off Allegro's software rasterizer.

The scanlist list is then read by the hardware's front-end, which will control the fragment generation. The hardware renders the scene described by the scanline list and finally outputs an image.

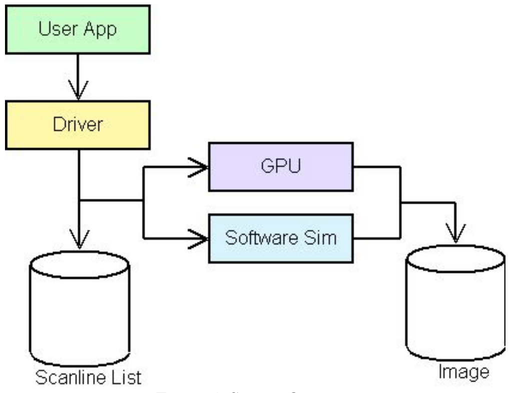


Figure 4. System Overview

The hardware was actually implemented twice. First, a C version was built to test the algorithms involved. The C version isn't bit accurate or even cycle accurate. It is only an equivalent software implementation of the hardware. It will read the same data as the driver, and output the an image similar to that of the hardware. The second implementation is in VHDL.

This dual system provides testing redundancy: We have a second implementation to validate against, increasing the robustness of the final hardware implementation.

Command Stream

The command set supported by the rasterizer is as follows:

Cmd	*Opcode*	*Description*	*Format*	*Type*
stxy	`FE_SET_XY`	x,y coords of the start of span	`xxyy`	u16
stxe	`FE_SET_X_END`	span x end coordinate (excluded)	`xx..`	u16
stiz	`ATTR_SET_Z`	1/z of the start of the span	`zzzz`	u0.32
stdz	`ATTR_SET_DZ`	d 1/z for the span	`dddd`	s0.31
stuz	`ATTR_SET_U`	u/z at the start of the span	`uuuu`	u0.32
stvz	`ATTR_SET_V`	v/z at the start of the span	`vvvv`	u0.32
stdu	`ATTR_SET_DU`	d u/z for the span	`dddd`	s0.31
stdv	`ATTR_SET_DV`	d v/z for the span	`dddd`	s0.31
stc0	`ATTR_SET_C0`	Red/Green color	`rrgg`	u0.16
stc1	`ATTR_SET_C1`	Blue color	`bb..`	u0.16
std0	`ATTR_SET_DC0`	Red/Green deltas per fragment	`rrgg`	s0.15
std1	`ATTR_SET_DC1`	Blue delta per fragment	`bb..`	s0.15
draw	`DRAW`	Draw span with current values	`....`	
txen	`TEX_ENABLE`	Enables texturing	`...e`	bool
txsz	`TEX_SET_SIZE`	Set texture size, as 2^w, 2^h	`wwhh`	u16
txad	`TEX_SET_ADDR`	Set texture FB address (ignored)	`....`	
txfl	`TEX_SET_FILTER`	Sets the texture filtering mode	`...m`	enum
fbsz	`FB_SET_SIZE`	Set frame buffer size	`wwhh`	u16
fbdp	`FB_DUMP`	Dumps content of frame buffer	`....`	

Table 1. Rasterizer Command Set

The Cmd column describes the command name in the scanline file, whereas the opcode field is the actual name used in the rasterizer.

The scanline attributes should be correctly set up prior to issuing a DRAW command.

Rasterizer Design

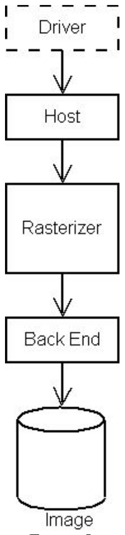


Figure 5. Rasterizer overview

The Host module we implemented is a stub for an actual host. Instead of managing he AGP or PCI port, it reads a command stream from a file and feeds it to the rasterizer. The rasterizer core then generates shaded pixels, which are passed to the Back-End. The Back-End would normally perform a depth test and optional blending before passing the result to a memory controller. Instead, our Back-End just writes the incoming fragments into a pixel buffer, which will later be written to an image file, in Targa (TGA) format.

The rasterizer is essentially a top-down pipeline. The Host connects to the Front-End of the pipeline to issue commands. The Front-End will then pass commands and attributes to the Attribute Interpolation block, which feeds into Perspective Correction. Perspective-corrected texture coordinates are then converted into a texture address by the Texture Addressing Unit, and passed to the memory controller. The other attributes are queued in a latency FIFO. Once the memory request is fulfilled, the Texture Filtering unit reads the attributes from the latency FIFO and the texels from memory and filter them to generate a fragment. This fragment then gets lit in the Shader unit, before being passed down to the Back-End.

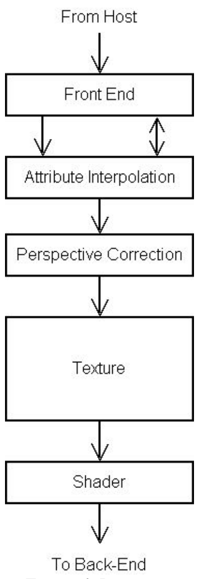


Figure 6. Rasterizer Block Diagram

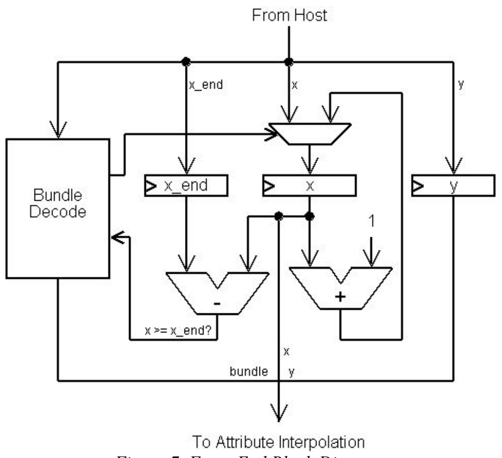
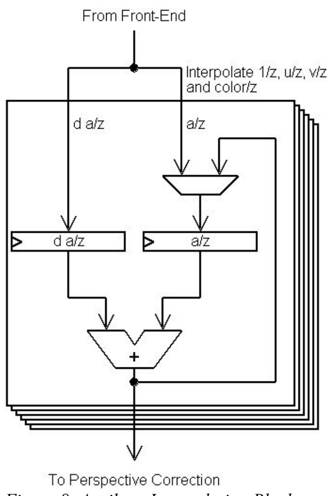


Figure 7. Front-End Block Diagram

The Front-End of the rasterizer decodes the command from the Host. The Front-End implements a simple state machine. It starts initially in the `IDLE` mode. In `IDLE` mode, it reads commands from the Host. If the command is a state change, then the command is passed down the pipeline unchanged. The relevant unit will pick it up and perform the state change. On the other hand, if the command is a draw command, then the Front-End switches to the `DRAWING` state. In that state, the rasterizer will appear busy to the Host and commands will no longer be read. The Front-End will then generate draw commands for the rest of the pipeline, one for each pixel in the scanline that needs to be rendered, looping from the scanline's start position to the end position. Once the end of the scanline has been reached, the Front-End reverts to its `IDLE` state and resumes command decode.

As draw commands are passed down from the Front-End, the Attribute Interpolation block will interpolate the scanline attributes for the next fragment. This is done via simple addition of a delta value, which was precomputed by the driver. We interpolate a specular color (all four channels), inverse depth, and perspective divided texture coordinates. That is, we interpolate *1/z*, *u/z* and *v/z* instead of *z*, *u* and *v*, simply because *1/z*, *u/z* and *v/z* are linear functions in screen-space, whereas *z*, *u* and *v* are hyperbolic, which makes interpolation significantly more complicated. This is the first step of perspective correct texture mapping.


From Front-End
Interpolate 1/z, u/z, v/z and color/z
d a/z
a/z
d a/z
a/z
+
To Perspective Correction
Figure 8. Attribute Interpolation Block Diagram

Because we use accumulation to interpolate attributes, the original attributes are overwritten by the interpolated values. This means that if a second scanline has the same attributes as the first one, the driver will still need to resend those attributes.

The interpolated scanline attributes are then passed down to the Perspective Correction block. This module will perform the second step of perspective correction - multiplication by *z* of *u/z* and *v/z* to obtain *u* and *v* for each fragment. However, since we only know the fragment's *1/z* value, we first need to reciprocate that to obtain *z*, then multiply the result by *u/z* and *v/z*. Normally, we would also do the same for color. However, this would require additional hardware and extra complexity for very little quality gain, since Gouraud shading does not usually appear distorted.

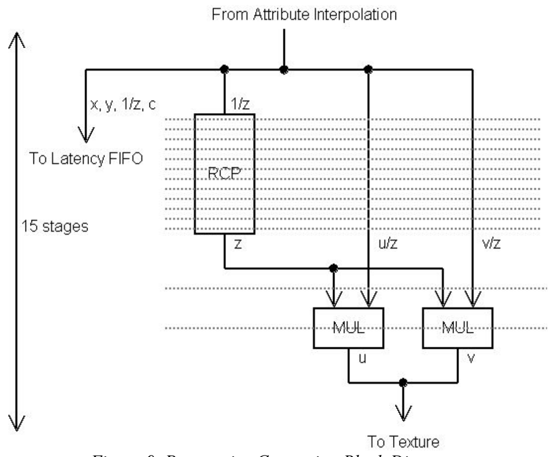


Figure 9. Perspective Correction Block Diagram

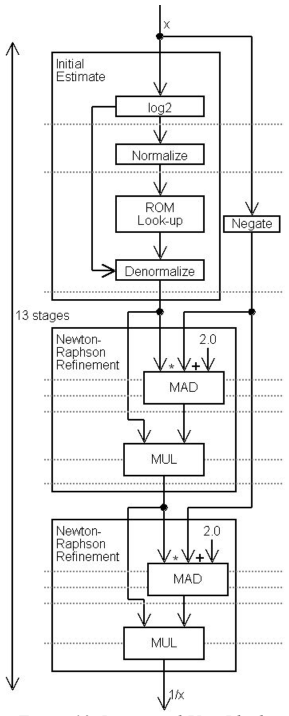


Figure 10. Reciprocal Unit Block Diagram

The reciprocal unit is a hybrid design, using a look-up table for the initial approximation, followed by two Newton-Raphson iterations to produce an approximation of the inverse of the input. The maximum error is 1.2 ulps when the input (a u0.32 number) is in the range `[2^-24..1[`. The output is a u24.8 number, clamped to the largest value on overflow.

$$y_{n+1} = y_n(2.0 - x\, y_n)$$

Figure 12. Newton Raphson Iteration

Beyond this, no further optimizations were made to the reciprocal unit, as it is beyond the scope of this project.

Texture coordinates are converted to u8.24 format and clamped on overflow. The results are sent down to the Texture unit.

The Texture unit is comprised of three sub-modules: Texture Address, Latency FIFO and Texture Filter. Note that we only support a single texture throughout the Texture unit.

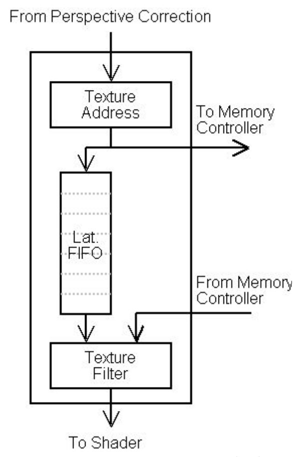


Figure 11. Texture Unit Block Diagram

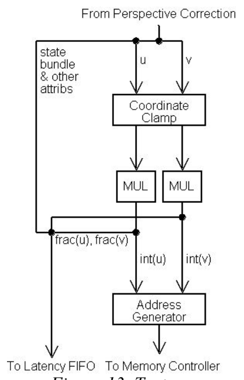


Figure 13. Texture Addressing Block Diagram

The Texture Address unit first clamps the texture coordinate components to the range [0..1]. It then multiplies the result by the texture size to obtain physical coordinates. Finally, the texture coordinate components are split into 16-bit integer and 8-bit fractional parts. The integer parts are used to generate a texture address whereas the fractional parts are queued in the Latency FIFO, to be used in filtering. The state bundle and other state are also queued in the FIFO, pending completion of the memory request.

Because texture requests need to go through memory, where latency is both very high and unpredictable, we use a latency absorbing FIFO. The FIFO should be implemented as an embedded dual-ported block RAM, to hold the intermediary values until the texture request is complete.

The Texture Filter unit waits for a memory request to be fulfilled, then picks up the corresponding attributes from the FIFO. Filtering is then performed on the received texels. The MUX select lines are determined from the pattern encoded in the alpha channel of texel A, combined with the fractional texture coordinates. The proper filtering equation is thus selected. Refer to Appendix A for the complete list. The filtering hardware is very similar to that of plain bilinear filtering. We have simply added some logic and MUXes for the filter weights. We also added swizzling MUXes between the two interpolation steps to allow the second step to source any of the results of the first step. These simple additions allow us to implement all of the filtering equations needed for the basic SmartFlt algorithm.

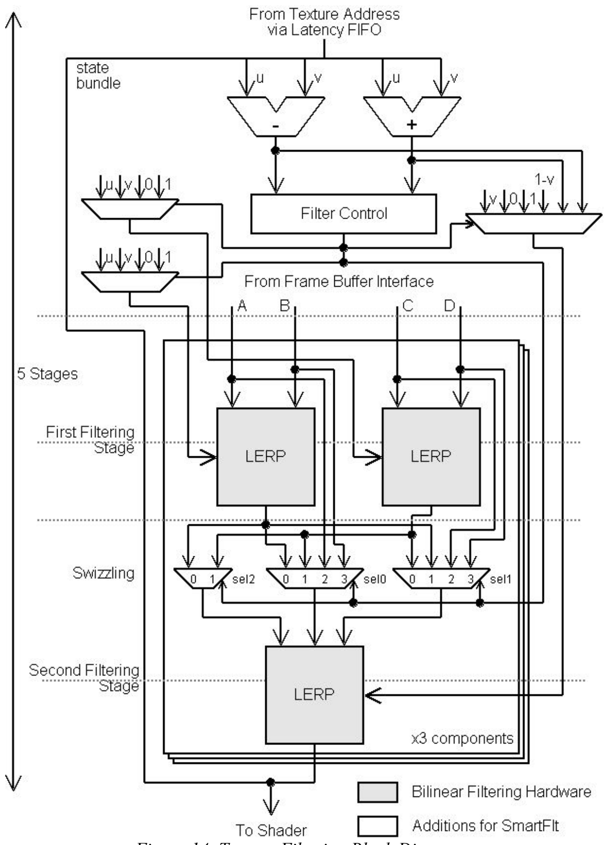


Figure 14. Texture Filtering Block Diagram

The only texture format we support is packed RGBA with 8-bit components. As we can see from Figure 14, the additional hardware needed for SmartFlt is quite small. Moreover, pipelining allows us to make it just as efficient.

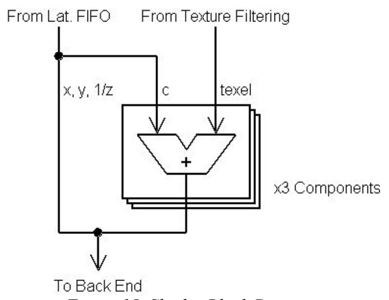


Figure 15. Shader Block Diagram

The filtered texel, along with the remaining fragment attributes, is then passed over to the Shader unit. The Shader unit is very simple. It was designed to do one thing: specular lighting. As such, it will simply add the fragment color to the filtered texel, and clamp on overflow. The final result is then output to the Back-End for final rasterization.

V. CHALLENGES

The first main challenge we faced was converting the SmartFlt software algorithm into hardware. SmartFlt has 14 different code paths for the 14 different patterns, all written in x86 assembly without much documentation. We essentially had to rewrite the filtering equations for each region of each pattern, using barycentric coordinates or point-to-line projection for the filter weights. Then we had to map all the equations into hardware. We noticed that they all resembled each other to a certain extent: they were all of the bilinear filtering form, with different weights and swizzled inputs for the final interpolation. From there, it was a simple matter to build the corresponding hardware.

The second main challenge was the fact that we needed a fixed-point reciprocal unit. We first looked up various algorithms to achieve this. A selected algorithm (Newton-Raphson with LUT hybrid) was then implemented in C, with checks for the maximum error, and tweaked to minimize that error. Then, we converted the code to VHDL and pipelined it as needed to achieve the 100 MHz target speed.

Finally, debugging proved to be more difficult than originally anticipated. There are thousands of signals to look at, and simulations must run for hundreds of thousands of cycles to produce meaningful results. This means that looking at signal graphs is a mostly futile exercise unless the bug was already located in both time and space. Locating bugs was done using a set of regression tests, which are incrementally more complex. The generated images are then compared to those created by the C version of the code. The tests stress particular parts of the pipeline, so it is easier to determine which unit is at fault. Moreover, the tests are quite short, needing only several hundreds of cycles to run. With that in mind, we can then look at the signals generated by said unit during the test to determine where the bug is.

VI. RESULTS

Synthesis

We synthesized the design onto the Xilinx Virtex II Pro FPGA, model 2VPX70ff1704. This particular FPGA was selected for its built-in multiplier blocks. We use many large multipliers in our design, so we would like to use built-in modules to both save on area and increase the speed of the design.

The blocks that were synthesized were:

– Front-End
– Attribute Interpolation
– Perspective Correction (with Reciprocal)
– Texture Addressing
– Texture Filtering
– Shader

Host and Back-End are stubs for actual units that would perform their tasks. They process files for I/O and are only really useful for simulation purposes. The Texture Latency FIFO should be implemented as embedded dual-ported block RAM instead, so it has also not been synthesized.

We obtained the following synthesis results:

Area	2880 CLB slices + 33 Block Multipliers
Speed	107.7 MHz
Throughput	Up to 1 pixel/clock

We have met our speed target, and the area usage is not very high.

SmartFlt vs Bilinear

We have also synthesized the Texture Filtering block by itself in two different configurations: SmartFlt with Bilinear and Bilinear alone. We obtained the following results:

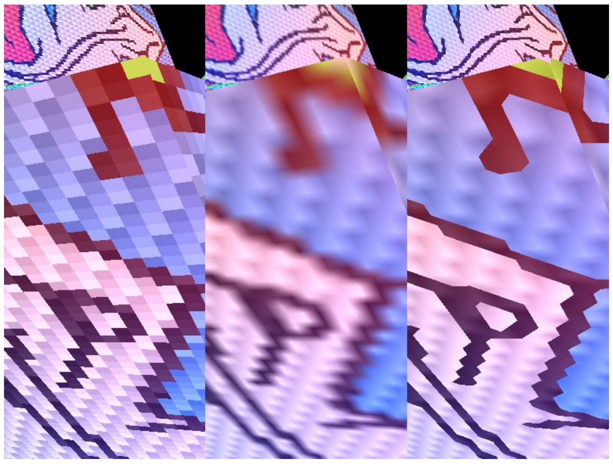
	SmartFlt	Bilinear	SmartFlt Increase
CLB Slices	554	443	+25%
DFFs	1107	886	+25%
Multipliers	12	12	
Latency	5 clocks	4 clocks	+25%
Speed	122.3 MHz	129.1 MHz	-5%
Bandwidth	4 samples/clock	4 samples/clock	
Throughput	1 pixel/clock	1 pixel/clock	

As we can see, the additional area for SmartFlt is quite small: only 111 CLB slices and 221 DFFs, which includes the two 8-bit adders. Although this looks like a 25% increase in area, it is actually much less, since multipliers consume a lot of die space and their number remains constant between the two circuits.

Latency has increased by one cycle, but this is less important than throughput for graphics applications. In case latency did matter, however, the first stage of the pipeline could be mostly moved in parallel to Texture Addressing, with the remains staying at the beginning of Texture Filtering, where there is some slack space left.

In terms of memory bandwidth, SmartFlt has the same usage pattern and footprint as Bilinear. This means that the image quality enhancements are essentially "free". Bandwidth is still a tight commodity in GPUs, even in the realm of many tens of GB/sec.

Generated Images



The above image was extracted from three 800x600 frame generated by the hardware, each with a different texture filter. On the left, Nearest Filtering was used. As we can see, the image is blocky. The individual texels are perfectly visible. In the center is Bilinear Filtering. The edges are blurred out, making it seem as if detail was missing. On the right is SmartFlt. The edges are kept sharp and distinct from the background, which is just bilinear filtered.

Generating these images took 2.5 ms in simulation, each (around 5 hours each using ModelSim XE Student Edition). This means that our scene with 96 large triangles renders at 400 frames per second.

VII. ISSUES REMAINING

SmartFlt seems to be both cheap and fast, so why not use it? There are still several issues that need to be resolved before an actual commercial application can be made.

Firstly, the texture preprocessing algorithm is not fully automated yet. It requires that a segregation image be provided along with the original texture.

Interaction with minimification has not been properly defined. We should make sure there are no image discontinuities when moving between the magnification and minimification cases.

Interaction with anisotropic sample selection has not been defined either. We should make sure there are no discontinuities in that regard.

VIII. CONCLUSION

SmartFlt is both cheap and fast. The image quality enhancement is quite noticeable. It has a good potential of being a worthy additional to bilinear filtering.

Moreover, the full SmartFlt algorithm has even more patterns, for an even higher filtering quality, making it potentially more interesting.

However, before SmartFlt can be implemented in commercial GPUs, some issues need to be resolved, such as interaction with minimifaction and anisotropic sample selection.

IX. REFERENCES

1. Maxim Stepin, "HiEnd3D, Smart Texture Filtering" http://www.hiend3d.com/smartflt.html
2. Shawn Hargreaves and others, "Allegro Gaming Programming Library" http://alleg.sf.net/
3. Mark Segal, Kurt Akeley, "The OpenGL Graphics System: A Specification (Version 1.5)" http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf

X. Appendix A - Patterns

List of filtering equations for each pattern:

P	Region	Filtering Equation
0		$lrp(lrp(A,B,u), lrp(D,C,u), v)$
1	$v<0.5$	$lrp(A,B,u)$
	$v\geq 0.5$	$lrp(D,C,u)$
2	$u<0.5$	$lrp(A,D,v)$
	$v\geq 0.5$	$lrp(B,C,v)$
3	$u+v\geq 1.5$	C
	$1\leq u+v<1.5$	$lrp(D,B,\frac{u-v+1}{2})$
	$0\leq u+v<1$	$lrp(A,B,u)+(D-A)\times v$
4	$v-u>0.5$	D
	$v>u$	$lrp(A,C,\frac{u+v}{2})$
	$v\leq u$	$lrp(A,B,u)+(C-B)\times v$
5	$0\leq u+v<0.5$	A
	$0.5\leq u+v<1$	$lrp(B,D,\frac{v-u-1}{2})$
	$1\leq u+v$	$lrp(D,C,u)+(B-C)\times(1-v)$
6	$v-u<0.5$	B
	$u>v$	$lrp(A,C,\frac{u+v}{2})$
	$u\leq v$	$lrp(D,C,u)+(A-D)\times(1-v)$
7	$v\geq 0.5$	$lrp(D,C,u)$
	$u<0.5, v<0.5$	A
	$u\geq 0.5, v<0.5$	B
8	$v<0.5$	$lrp(A,B,u)$
	$u<0.5, v\geq 0.5$	D
	$u\geq 0.5, v\geq 0.5$	C
9	$u\geq 0.5$	$lrp(B,C,v)$
	$v<0.5, u<0.5$	A
	$v\geq 0.5, u<0.5$	D
10	$u<0.5$	$lrp(A,D,v)$
	$v<0.5, u\geq 0.5$	B
	$v\geq 0.5, u\geq 0.5$	C
11	$v-u<-0.5$	B
	$v-u\geq 0.5$	D
	$\frac{-1}{2}\leq v-u<\frac{1}{2}$	$lrp(A,C,\frac{u+v}{2})$
12	$1.5\leq u+v$	C
	$0\leq u+v<0.5$	A
	$0.5\leq u+v<1.5$	$lrp(B,D,\frac{v-u+1}{2})$
13	$u<0.5, v<0.5$	A
	$u<0.5, v\geq 0.5$	D
	$u\geq 0.5, v<0.5$	B
	$u\geq 0.5, v\geq 0.5$	C

Where $lrp(A,B,u)=(B-A)\times u+A$, *A*, *B*, *C* and *D* are the four texels to filter, and *u* and *v* are the fractional parts of the texture coordinate.

XI. APPENDIX B – SOURCE CODE

XII. APPENDIX C – SYNTHESIS SCRIPTS