Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

The Visual Vulnerability Spectrum: Characterizing Architectural Vulnerability for Graphics Hardware

Jeremy W. Sheaffer[1†] David P. Luebke[2‡] Kevin Skadron[1§]

[1]Dept. of Computer Science, University of Virginia, Charlottesville, Virginia, USA
[2]NVIDIA Research, Santa Clara, California, USA

Abstract

With shrinking process technology, the primary cause of transient faults in semiconductors shifts away from high-energy cosmic particle strikes and toward more mundane and pervasive causes—power fluctuations, crosstalk, and other random noise. Smaller transistor features require a lower critical charge to hold and change bits, which leads to faster microprocessors, but which also leads to higher transient fault rates. Current trends, expected to continue, show soft error rates increasing exponentially at a rate of 8% per technology generation. Existing transient fault research in general-purpose architecture, like the well-established architectural vulnerability factor *(AVF), assume that all computations are equally important and all errors equally intolerable. However, we observe that the effect of transient faults in graphics processing can range from imperceptible, to bothersome visual artifacts, to critical loss of function. We therefore extend and generalize the AVF by introducing the* Visual Vulnerability Spectrum *(VVS). We apply the VVS to analyze the effect of increased transient error rate on graphics processors. With this analysis in hand, we suggest several targeted, inexpensive solutions that can mitigate the most egregious of soft error consequences.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics Processors

1. Introduction

Exponential device scaling has produced incredible advances in the capability of today's computing infrastructure. Graphics processors have taken advantage of these scaling trends to achieve dramatic increases in throughput. Semiconductor devices, however, have now become so small that they are vulnerable to transient faults caused by cosmic and terrestrial radiation; and to noise due to crosstalk, di/dt induced voltage droop, and parameter variations. As the importance of these phenomena all grow exponentially with decreased feature size or supply voltage [SABR04], the 'free lunch' of Moore's Law for graphics architects approaches its end. Future designs must be more aware of such low-level physical challenges.

A transient, single bit corruption in a microelectronic circuit is termed a *soft error*. Soft errors have long been an important design constraint in general purpose processor design, especially in engineering reliable memory systems for enterprise servers. They have yet to become a major consideration in the design of graphics processing systems, probably because the primary market is the consumer desktop, where reliability requirements are lower. Yet as soft error rates are projected to continue their current trend of increasing at a rate of about 8% per technology generation [HKM*03]—making soft error rates at 16-nm nearly 100 times that of the 180-nm generation [Bor05]—they will soon become a driving concern for graphics architects. Advanced 3D graphics capabilities and expanding requirements for 3D rendering in next generation operating systems, such as *Microsoft Windows Vista*, will only exacerbate a problem that otherwise would only have been important to competitive game players, bringing soft error tolerance quickly to the forefront of GPU reliability.

A soft error is distinguished from a *hard error* by its transient nature—a soft error is random, temporary, and unpredictable. Soft errors are referred to by several names, including *transient fault*, *transient error*, and *single event upset* (SEU). While these are often used interchangeably, 'soft error' and 'SEU' have classically referred only to radiation-induced transient faults. This subtlety seems to be largely forgotten, and we choose to ignore it in this paper.

Not all errors are cause for concern. If errors do not matter for *architecturally correct execution* (ACE)—in other words, if they do not affect the final outcome of the computation—they are harmless. An error might be harmless, for example, if it strikes a storage location that is not currently in use (i.e., not ACE). Figure 1 illustrates a taxonomy for the classification of soft errors.

The dominant metric for quantifying the chance of an er-

† jws9c@cs.virginia.edu
‡ david@luebke.us
§ skadron@cs.virginia.edu

ror as a result of a transient fault is the *Architectural Vulnerability Factor* or AVF [MER05]. The AVF of a structure is a fraction from zero to one which represents the likelihood that a transient fault in that structure will lead to a computational error. AVF takes into account the total amount of time that each bit can contribute to a computation, the total number of bits in the structure, and the size of the structure. More formally, Architectural Vulnerability Factor is:

$$\mathrm{AVF} = \frac{\sum_{b \in B} t_b}{|B| \times \Delta t} \tag{1}$$

where B is the set of all bits in the structure, t_b is the total time that bit b is ACE, and Δt is the total time necessary to complete the computation.

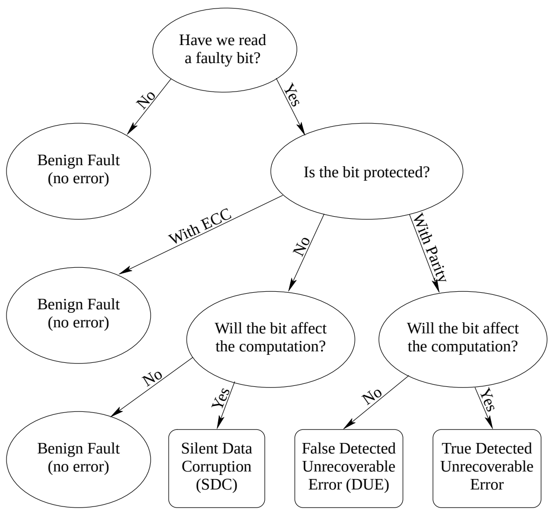

Have we read a faulty bit?
No
Yes
Benign Fault (no error)
Is the bit protected?
With ECC
No
With Parity
Benign Fault (no error)
Will the bit affect the computation?
Will the bit affect the computation?
No
Yes
No
Yes
Benign Fault (no error)
Silent Data Corruption (SDC)
False Detected Unrecoverable Error (DUE)
True Detected Unrecoverable Error

Figure 1: *A transient fault can lead to no error due to correction or the effected bit being un-ACE, silent data corruption—an error which is never discovered—or a true or false detected unrecoverable error. This figure is based on Figure 1 in Mukherjee et. al. [MER05].*

The history of soft errors is long and interesting, dating back to the mid-1950s and in some cases involving some amazing detective work to track down causes. The many hard-learned lessons have had a huge influence on modern fab technology. Unfortunately, discussion of this topic is outside of the scope of this paper. Interested readers should start with the papers by Ziegler [ZCM*96, Zie96] and Normand [Nor96].

In general-purpose computer systems, any ACE bit must be assumed important. Even a single error in a low-order bit in a commercial or scientific computation can invalidate a computation. What makes graphics hardware unusual is that most state on the graphics card—despite being technically ACE—can tolerate some degree of error. *Errors only matter if they affect the user's perceived experience.* An error in a single pixel, for example, may not be noticeable even if it changes the color from white to black. Errors in other state may create more visible errors, but if those errors only last a single frame, the harm is minor.

This observation obviously does not apply to graphics hardware used in non-visual applications (e.g. GPGPU) where CPU error metrics are more directly applicable. If these applications are of sufficient commercial value, they may require cards with full error protection, such as ECC-guarded video memory, and an efficient implementation of that for graphics cards is left for future work.

This paper explores the implications of transient faults in graphics hardware used for interactive consumer applications. It observes that common CPU metrics for determining transient fault vulnerability, such as AVF, do not fit well with the workloads and expectations of graphics systems and presents the Visual Vulnerability Spectrum as a more suitable taxonomy for classifying vulnerability on GPUs. Finally, this paper presents some initial suggestions for fault protection and recovery mechanisms specifically tailored to GPUs.

2. The Visual Vulnerability Spectrum

For CPU architectures, it must generally be assumed that any transient fault resulting in a change in the final computation is unacceptable. The workload of a graphics processor tends to be more forgiving of most soft error effects. For this reason, we argue that attempting to apply AVFs to GPUs can be misleading.

Most soft errors in computation and memory on graphics cards are acceptable or even unnoticeable. Consider, for example, the color framebuffer. We ran a sequence of 589 frames from id Software's *Doom 3* with all features enabled through an instrumented version of Mesa at 1600×1200 resolution with 32 bits of color. The color buffer for this application is about 7.32MB. The mean depth complexity during the sequence was 4.09, implying that many errors in the color buffer are likely to be overwritten. However, the AVF depends also on how long non-overwritten values are resident in the framebuffer, and a detailed calculation using Equation 1 (see Section 3.1) gives a framebuffer AVF of 0.48 (some representative images from this study appear in Figure 2). In other words, any single bit error in the framebuffer has a 48% chance of affecting the final image. By traditional AVF analysis, this is very high, arguing that we should consider the color framebuffer a critical structure and heavily protect it. Of course, in practice the opposite is true: a user is quite unlikely to care about or even perceive a single-bit error in a single pixel for a single frame!

This example underscores a key point of this paper: because all errors are not equal in the graphics workload, it is more useful to think of GPU architectural vulnerability as a multi-dimensional continuum rather than as a single scalar chance for an error. We call this continuum the *Visual Vulnerability Spectrum* to emphasize its continuous nature, and identify three primary axes to quantify important and orthogonal qualities of graphics computation vulnerability: extent, magnitude, and persistence.

1. **Extent** refers to how many pixels will be affected as a result of a soft error. Qualitatively, this axis ranges from *unnoticeable* to *whole screen*. For example, our framebuffer example posited an error affecting one pixel in the final image, which is probably unnoticeable extent, while an error in a coefficient of the modelview matrix could easily have a whole screen extent.
2. **Magnitude** describes the severity of the error across the affected region of the final image. In principle, magnitude is a complex perceptual function; in practice we approximate magnitude using the L^2 error in RGB color space across affected pixels. Qualitatively, the magnitude axis ranges from *unnoticeable* to *insufferable*. A change to the low-order bit of a color channel would probably be unnoticeable, a clipping error that clipped away all geometry at affected pixels would probably be considered insuf-

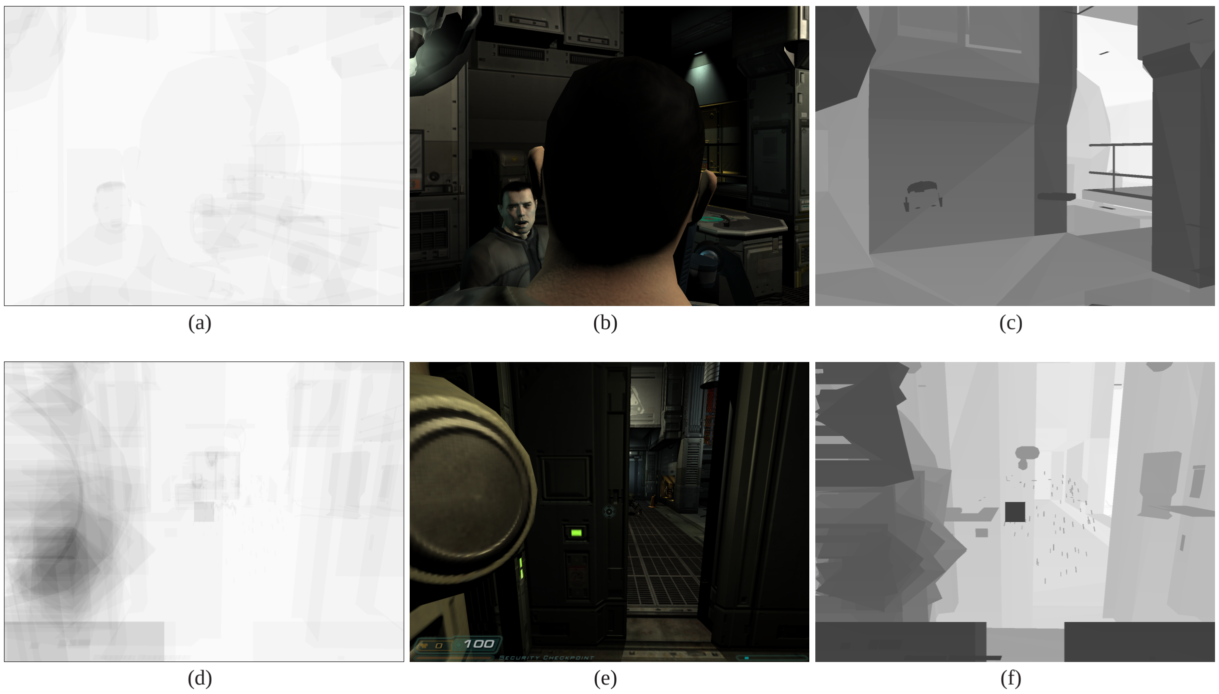


Figure 2: *(a) and (d) are depth complexity maps of the frames displayed in (b) and (e) respectively. (c) and (f) are the respective AVF maps. In the depth maps, white corresponds to a depth complexity of zero, while black represents a depth complexity of 51—the highest in our 589 frame sampling. In the AVF map, white represents an AVF of 1, while black is an AVF of 0. In (a), the depth complexity is reasonably consistent across the frame, and so using any of mean or median will give reasonable estimates of AVF. In (d), most of the frame has little complexity. The majority of the fragments are generated by a particle simulation that is entirely occluded from the vantage point of this image—note that the complexity is not due to the foreground object in (e)!—which artificially drives up the AVF of the frame. The bottom frame has a depth complexity of 4.80 and an AVF of 0.65, while the top has 2.24 and 0.48 respectively, even though, as is apparent from the depth maps, both of these frames have similar complexity save the particle simulation.*

ferable, and an error changing global anti-aliasing state might fall somewhere in between.

3. **Persistence** refers to how long the effect of a soft error will remain active. Persistence is measured in frames, typically 16-33 ms in real-time rendering applications such as video games. Qualitatively, persistence ranges from *transient*—the effect disappears after a single frame—to *indefinite*. An error in depth or color buffer would be transient, since these are cleared every frame, while an error in the value of a vertex buffer cached in on-card memory could last an indefinite number of frames. A hard error in our taxonomy would simply have a persistence of *permanent*.

While many subtleties of soft error impact are not directly captured in these three axes—for example, the severity of an error that corrupts a vertex buffer which is rendered repeatedly throughout a frame depends in part on the exact intra-frame timing of the error—we argue that the Visual Vulnerability Spectrum provides a sufficiently rich characterization of the range of soft error effects to allow graphics architects to usefully analyze soft error impact and protection schemes.

2.1. Application

To illustrate such an analysis, we have applied our taxonomy to the OpenGL 2.0 state vector [SA04] and identified a short list of state structures as high-priority candidates for soft error protection. Of course, there is no direct mapping between the OpenGL API and actual hardware structures, but suitable architectural details are not usually publicly available and the GL state is sufficient to delineate the different categories of vulnerability and indicate what types of hardware protection, if any, might be justified.

- **Matrix stack:** Almost any error in the matrix stack can produce errors in the final image with large extent and magnitude. A matrix stack error at the base of the stack could persist for several frames.
- **Scissor, depth, and alpha test enable bits and functions:** For example, if the programmer enables the depth test but a transient fault disables it, all subsequent geometry will write to the framebuffer. For simple applications, this effect could potentially persist until the application is terminated.
- **Viewport function coefficients:** Errors in the viewport will affect how much of the scene is displayed or how large the scene appears in the available viewport.
- **Depth range:** With errors in the depth range state, order-dependent occlusion errors could appear in the output image. This piece of state is often set during the initialization phase of an application or game and never modified again, so an error here will likely persist until the application completes.
- **Clip plane function coefficients:** Arbitrary errors in the clipping planes can cause clipping of geometry that should appear in the final image. Like the depth range, this state will often be unmodified by the programmer after initialization.

- **Lighting enable bits:** Enabling lighting in a scene with no lights will yield a black screen, while disabling it will eliminate most lighting effects—assuming that fixed-function OpenGL lighting is being used rather than programmable shading.
- **Culling enable bits:** Toggling the state of back- or front-face culling enable bits will change which geometry is allowed to change the image.
- **Polygon state including offset, stippling, and fill modes:** Changes to these state can drastically affect the appearance of rendered polygons.
- **Texture enable, active texture, and current texture unit:** Errors here will change which, if any, texture is applied to geometry.
- **Individual texture state:** Though individual texels are not high priority, errors that corrupt the associated texture state—texture dimensions, the format of the texels, clamping or wrapping mode, etc.—can have high extent, magnitude, and persistence.
- **Current drawbuffer:** Advanced applications making heavy use of render-to-texture often change render targets, but simpler applications may never modify the render target, so that an error corrupting the render target could easily result in a black screen until the application terminates.
- **Uniform and control-related shader state:** This includes the compiled program store, instruction counter, and uniform registers shared by all invocations of a shader. An error to a uniform register could affect all vertices or pixels processed by the shader for the remainder of the frame, and possibly (for simple applications using only one shader) persist for many frames. Errors in control state such as shader instruction counters could potentially crash the GPU.

Structures of intermediate importance include:

- **Vertex array enable bit, size, type, stride, pointer, high-order bits of vertex array elements, and the entire contents of all index arrays:** Some experiments and results with this state are discussed in Section 3.
- **Vertex attribute arrays:** Similarly for normal, fog, color, edge, index, and texture coordinate arrays.
- **High levels of the hierarchical z-pyramid:** An incorrect depth test due to an error in the z-pyramid will have an extent of $x^n \times x^n$ pixels, where x is typically 8 pixels and n is the level of the pyramid where the error occurred. Hardware using more than a single-level hierarchical depth test might want to protect those levels with $n >= 2$. However, such errors only persist for one frame.
- **Texture contents:** A corrupted bit in a texel could occasionally have large extent, for example if the texel is magnified or wrapped across many pixels. While the texture is cached on the card, the error may have a persistence of many frames. However, most errors will have relatively low magnitude.

Here are a few examples of items that are not important in most rendering applications:

- **The framebuffer:** A single bit error in the framebuffer will probably only affect one pixel. While this will potentially fall very high on the magnitude axis, it will have a very low extent (1 pixel) and persistence (1 frame). Note that if the framebuffer stores something other than raw frame data—like compressed data or context switch objects—then it is more vulnerable at least in terms of extent and perhaps in other dimensions.
- **Shader data registers:** Unlike the uniform registers mentioned above, the input and temporary registers that vary with each pixel or vertex being processed will have a persistence of no more than one frame, and in the case of pixel shaders, an extent of no more than one pixel.

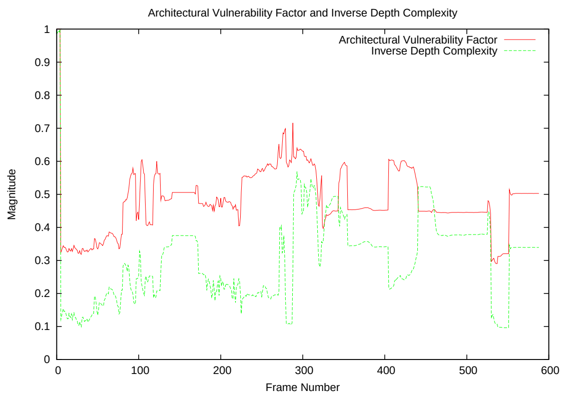


Figure 3: *AVF in the framebuffer tends to move with the inverse of depth complexity, but is more complicated since it accounts for the exact timing of when the nearest value at each pixel is written.*

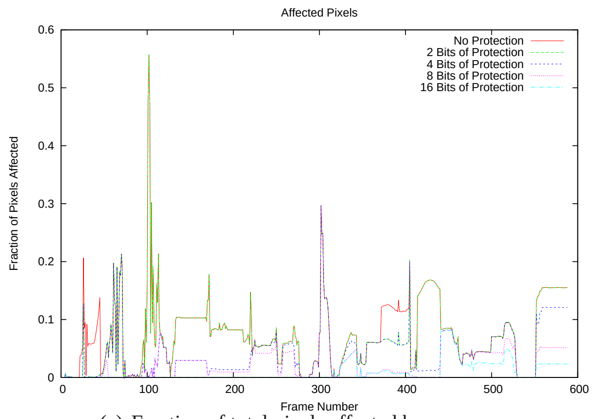
- **Antialiasing state:** An error in antialiasing state will likely persist for the remaining lifetime of the application, yielding a high persistence value, but extent will be low, since the errors will only show up only at high-frequency edges, as will magnitude which will be determined by a weighted average of colors that correctly occur in or near any given sample.
- **Various non-array vertex attribute state:** Including color, texture coordinates, normals, and generic shader attributes. This state can drastically affect all polygons using the corrupt vertex, but will not persist.

3. Characterization and Results

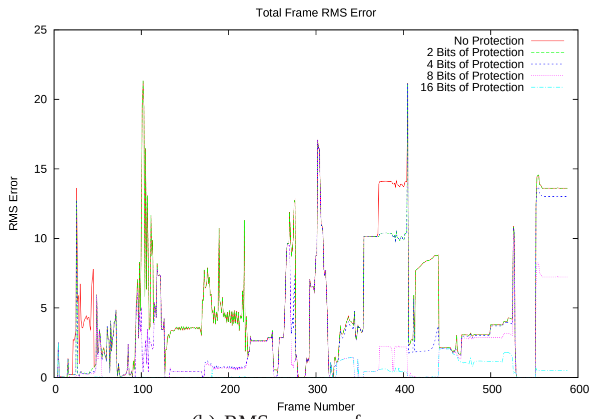
We describe two example experiments to illustrate the concepts in this paper. The first calculates a traditional AVF for the depth buffer; the second illustrates how the VVS could be characterized for a particular structure, in this case the vertex buffer.

3.1. Calculating AVF of GPU Structures

We instrumented Mesa-6.4.1 [P*06] to count depth buffer reads and writes and to dump this data, per frame, to disk. We implemented Equation 1, AVF $= \frac{\sum_{b \in B} t_b}{|B| \times \Delta t}$, in Mesa by placing a framebuffer sized—1600×1200—matrix, containing: arrays of 51 sequence numbers and ACE bit counters; an index into those arrays; a depth (so that we can easily perform a depth complexity analysis); and read and write counters. From a previous, simpler experiment, we know that no frame in the 589 frame `demo1` sequence—a demo path that ships with Doom 3—has a depth complexity of greater than 51. Each time a new fragment is tested, a global sequence number is incremented and stored in the sequence number element indexed by the current index for that sample, and 32 depth tests are performed–Mesa uses a 32 bit depth buffer–one for each bit in the depth value XORed with 2^{bit}. At the end of each frame, we compute per pixel AVFs by, for each element in the matrix: subtracting from each sequence number number in the sample's array the sequence number before it (subtracting zero from the first element); multiplying that difference by the corresponding ACE bit count; and dividing by 32 times the final sequence number (32 bits of data over *sequence number* units of time). Of course this assumes

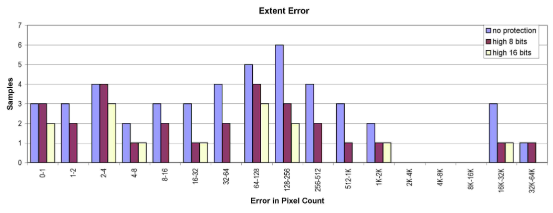


(a) Fraction of total pixels affected by an error.

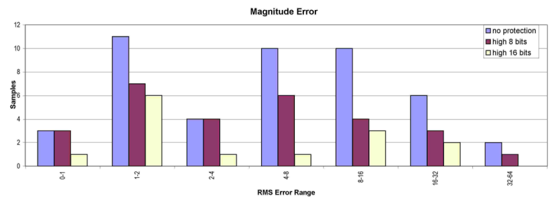


(b) RMS error per frame.

Figure 5: *These traces show error per frame from a set of reference images. Both graphs are based on the same set of faults in vertex array data and the same reference images. Further, this data is based on the assumption that the vertices have been downloaded to GPU memory but are not updated over the course of these 589 frames—each individual error persists for the duration of the simulation. Note that while frame RMS error tends to move with extent, or affected pixels, they are not directly proportional since frame RMS error also incorporates the magnitude of the error.*



(a) Errors on the extent axis



(b) Errors on the magnitude axis

Figure 4: *Histograms showing soft error extent and magnitude.*

that fragments are tested at constant intervals, but this is not a poor assumption. A global sum is used to calculate the per-frame AVF.

3.2. Vertex Fault Injection

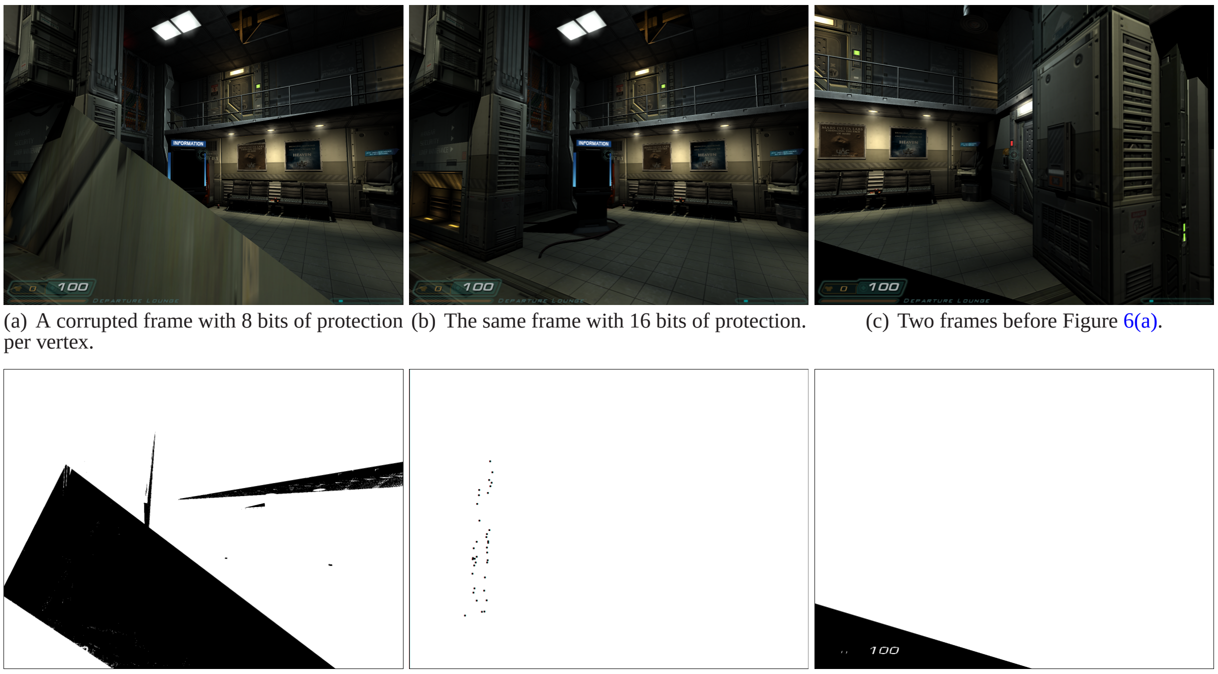
To illustrate how a specific class of errors can be analyzed with the VVS framework, we implemented a Chromium [HHN*02] *Stream Processing Unit* (SPU) to simulate transient faults in graphics memory by injecting errors into vertex position arrays. These faults are injected under various constraints to simulate different memory management configurations and transient fault protection techniques. We then analyze the extent, magnitude, and persistence of the resulting errors.

Cosmic and terrestrial ray flux is uniformly distributed over small areas and over time; transient faults due to other causes (crosstalk, voltage droop, etc.) may be less uniform but are impossible to model accurately without detailed hardware knowledge. We therefore assume that all vertices are equally likely to be corrupted, and inject faults into vertices being processed after random intervals averaging 1 fault per 100,000 vertices†. After some transformations in Chromium, all vertices come to our fault injection SPU in the form of `glVertex3fv()` calls. We randomly choose a vertex and a flip a random bit from that vertex's position values. We pass the new, corrupt value on to the renderer, and perhaps back into memory. Implementing the high-order bit memory protection scheme discussed in Section 4 is as simple as not performing bit flips in the protected high-order bits.

Figure 4 analyzes the extent and magnitude of errors caused by individual vertex faults. Figure 4(a) shows a histogram of errors classified by the number of affected pixels. Most errors affect zero pixels, such as when the corrupted vertex is occluded or off-screen, or when the corruption is too small to visibly affect the geometry. However, about 10% of the errors have nonzero extent. Note that the histogram bucket size increases exponentially: each column represents errors affecting twice as many pixels on average as the previous column. Thus errors captured in the right side of the histogram are much more severe than errors on the left side. The figure compares the severity of the errors resulting from protecting the high-order 8 and 16 bits of the 32-bit position values to the errors that occur with no protection. These results show that protecting even a fraction of the bits significantly reduces the number of errors with severe extent.

Figure 4(b) uses a similar histogram to show the distribution of error magnitudes, measured as RMS difference of affected pixels in RGB color space. Again, protecting a

† This corresponds with approximately one corruption per frame in our Doom 3 trace—much higher than would be expected from cosmic radiation, but not unreasonable for an aggressively overclocked GPU built on a near-future semiconductor technology node.



(a) A corrupted frame with 8 bits of protection per vertex. (b) The same frame with 16 bits of protection. (c) Two frames before Figure 6(a).

(d) Corrupted pixel map of (a). (e) Corrupted pixel map of (b) (f) Corrupted pixel map of (c).

Figure 6: *Images (a) and (b) show the same frame from two different vertex array fault injection sequences in Doom 3. The sequence that produced (a) protected the high-order 8 bits of each vertex, while the sequence that produced (b) protected the 16 highest-order bits. (d) and (e) are corrupted pixel maps of (a) and (b) respectively. The reference image is not included since, while (b) is not a perfect image, it is indistinguishable from the reference. The triangle that has been affected by the error in (b) moved such a small distance that most of the fragments it generates map to exactly the same color and screen-space position. (c) and (f) show the same error as the other images, but two frames earlier, in the frame in which the error actually occurred. In the sequence that generated these images, vertices were assumed to be downloaded each frame, so errors do not persist between frames, therefore this error no longer exists in this sequence during the frame in (a) and (b), and in fact, in this sequence, the frame in (a) and (b) is identical to the reference image.*

fraction of the bits significantly reduces error magnitude. In other words, not only are fewer pixels affected by errors, but the effect of the errors on those fewer pixels is also reduced.

When analyzing the persistence of soft error corruption to vertex data, we must consider two scenarios: streaming versus cached vertex buffers. In the first scenario, vertex data is streamed from system main memory, downloaded to the graphics card each frame. Thus all errors will have a persistence of one frame. To implement, we modify only the data that is passed downstream in the Chromuim SPU chain to ensure the resulting errors only affect the current frame.

The more interesting analysis of persistence occurs in the second scenario of cached vertex buffers, which once downloaded to graphics memory remain there and are not modified for some time. Persistence is now a function of when and for how long the corrupted geometry appears on the screen, which in turn depends on the scale of the error and the motion of the player. To implement, we modify the vertex array data directly, causing the errors to persist for many frames. Figure 5 shows a plot of total error over the course of the 589-frame trace sequence, measured both as extent (5(a)) and total frame RMS error (5(b)), which essentially represents extent times magnitude. In these plots, soft error persistence manifests as the tendency of errors to remain constant for several frames. Figure 5 also illustrates the effect of protecting varying numbers of bits in the input. Once again, protecting only a few bits greatly reduces the total effect of most errors.

In truth, Doom 3 falls somewhere between these two models: while some objects are downloaded for rendering every frame, most of the vertex data is stored in VBOs and is only downloaded by Doom 3 once. VBOs are managed by the graphics driver, which generally attempts to cache them on-card whenever possible for optimal performance.

4. Transient Fault Protection Schemes for GPUs

The fact that typical consumer applications for graphics hardware can tolerate some errors allows novel, low-cost error protection that still successfully limits severity of soft errors. As mentioned, this differs from general-purpose hardware, where any error to ACE state should be corrected. Here we propose some initial possibilities for graphics-specific protection and hope that future work will explore additional techniques.

As discussed in Section 2.1, only a small amount of state requires any protection, especially if visually significant errors can be tolerated for short periods of time—on the order of one to a few frames. The frame buffer can be left completely unprotected, because single errors will likely affect only one pixel. Even the z-buffer can be left unprotected, because errors will only persist for at most one frame. Most of the remaining large objects in memory, such as vertex storage, textures, etc. will benefit from protection, because they may not be reloaded every frame. But this protection need only detect errors, assuming all this state can be reloaded;

and detection need only operate on approximately a once per frame basis. A small amount of state that is rarely modified, such as various enable bits, coefficients, etc. is more important and should be fully protected. Other persistent state, such as vertices and shader code, requires only periodic error detection, just to prevent persistent errors.

4.1. Full Protection

Specifically, we propose full protection for the various enable bits, array state (not array data), viewport and clip plane coefficients, depth range, polygon state, current drawbuffer ID, uniform and control-related shader state and the matrix stack. The likelihood that these small state elements will be corrupted by radiation strikes is infinitesimally small, but the effects could be dramatic and lasting. More importantly, this state is vulnerable to errors from non-ideal circuit behavior and stability due to deep-submicron scaling.

Since these items are all small, the overhead of this protection is also small. Simple choices include upsizing the devices used to implement this state, hence increasing their critical charge Q_{crit} (the size of the disruption needed to change a transistor's state), or ECC. Full redundancy is a possibility but requires an XOR to compare the two copies and detect an error, as well as some mechanism to recover from the error. Triple redundancy allows correction but requires even more overhead.

4.2. Simple Parity Protection

The shader program store at each shader unit may not be updated for long periods of time, especially if organized as a cache. Errors in the shader code could be catastrophic and persistent. Simple detection suffices, because the shader code can be reloaded and re-initialized. This suggests conventional parity protection on a per-line basis for the program store. This may be more important for vertex shaders, because errors in a vertex can affect a large extent, while errors in a fragment are limited to a pixel. In a unified shader model, all shader stores will require this capability.

4.3. Periodic Error Detection

The dominant graphics memory today is Samsung and ATI's GDDR3. This is high-bandwidth, double data-rate VRAM is engineered specifically for graphics, with neither error detection nor correction [Sam05]. This memory is the primary store for most off-chip objects, including vertices, shader code, textures, and the z-buffer.

We can take advantage of the fact that most of the off-chip graphics state we are concerned with, including vertices, shaders, textures, normals, texture coordinates, and other such data, is replicated in the CPU-side driver space and hence resides in the CPU's main memory. In a modern, fault-tolerant system, it is safe to assume that this is protected with ECC.

Even in persistent state such as vertices and shaders, errors disrupting a few frames are usually tolerable. What we wish to protect against are persistent errors that would not go away without added hardware protection. To achieve this, we need only implement a low-cost, low-frequency detection mechanism, using the driver's copy of data to replace erroneous graphics data whenever an error is detected. We call this technique *Periodic Error Detection* and present two ways of implementing it here. The key is to observe that we need only detect errors over large objects, and this computation is off the critical path, allowing hardware implementations optimized to avoid any impact on access latency or bandwidth. Depending on the anticipated error rate, a single parity bit per object may suffice, or a slightly more sophisticated check may be needed, such as a checksum. Of course, any solution of this nature requires driver support, the implementation of which may be non-trivial.

The advantage of this graphics-specific approach is that only a few error checks are needed, the state to be stored is small (e.g., one parity bit per row in the RAM), and the detection need only be performed on a relatively infrequent basis. Conventional DRAM, on the other hand, provides parity on a per-byte basis, or ECC on a per-word basis; and every DRAM access requires error detection/correction.

The one problem with the graphics-specific approach is that, if we only detect errors across an entire row, it requires the ability to perform error detection across a large quantity of data. Reading all this data off the GDDR just for parity checks is undesirable. There are two ways to achieve this functionality with minimal overhead. The first piggybacks on the existing refresh mechanism inside the GDDR. The second piggybacks on existing streaming accesses, such as accessing a vertex array.

4.3.1. Refresh-based techniques.

DRAM cells do not maintain a connection to the power supply and hence cannot maintain their contents. The charge stored in a cell gradually leaks away over time. SRAM, in contrast, maintains a connection to the power supply, at the expense of additional transistors, inferior area efficiency, and higher power.

This means that DRAM requires periodic *refresh*, in which a row of the DRAM is read out of the data array and immediately written back. A typical retention time for data is on the order of 10s of milliseconds; the datasheet for the GDDR [Sam05] specifies that data must be refreshed every 32ms, which conveniently is about the time for processing 1–2 frames. Since the chip contains 4K rows, this means that a row must be accessed for refresh every 7.8μs. These refreshes are mandatory, and while the bits are available in the buffer before writeback, an error detection such as parity can be computed. The error detection is therefore performed one row at a time, as each row is accessed. This requires at most one element of check state per row (e.g. 1 parity bit per 16 Mbit row), and at least one element of check state per bank (e.g. 1 parity bit per 64 Mbit bank). This requires only a simple error-checking circuit on the DRAM chip that is capable of completing the computation for a single row in 7.8μs, and possibly the ability to combine that with results from a prior row if the error detection is aggregated instead of being performed on a per-row basis.

When an error is detected, the driver must be notified to reload appropriate state. A brute-force solution is for the driver to cause the entire state of the graphics computation to be reloaded. While the expense of this is considerable, it may be tolerable if errors are relatively rare.

The cost of error recovery can be reduced if the driver can attribute errors to specific objects that must be reloaded, such as vertex arrays. This presupposes driver-level data structures capable of identifying which, if any, memory objects (textures, vertex buffers, depth or color render targets, etc.) are resident at the given address, so that those objects can be reloaded if necessary when a fault occurs at that address. A simple ordered list of all objects requiring protection would suffice. Only objects requiring protection need to be entered into this list. Then if a row does not match an object in the list, the error can be ignored.

4.3.2. Demand error checking.

In demand error checking, the graphics driver calculates check state for each object of interest and associates that bit with the data structure that it downloads to the GPU. Each time that object is accessed, error detection is performed. Since the error detection is only done in conjunction with an access that would be performed anyway, and since it is not time-critical, the overhead should be small, for example a parity checker that operates while data streams off the GDDR. Buffering the data allows for a simpler, lower-cost error checker that can lag behind the data streaming off the memory, performing the check computations as the main graphics operations proceed. The error check is not in the critical path and only imposes the cost of the fanout. On detecting an error, the check circuitry alerts the graphics driver to download a new copy of the data.

Demand parity is best used with large blocks of state that will be read in a repeatable manner; vertex and index arrays are an obvious target for this technique. For objects that are accessed with different ranges, such as index arrays, each possible range requires a check value. This requires a table, probably stored in graphics memory as well.

4.4. High-Order Bit Protection

Unlike general-purpose computations, high-order bits are more important for graphics than low-order bits. Section 3.2 demonstrates this for vertex array data. This motivates a protection scheme in which only the *x* highest order bits of each array element are protected. Based on the results in Section 3.2, 16 bits seems to be sufficient here, protecting the sign, the exponent, and the 7 highest-order bits of the mantissa in IEEE single-precision floating point. This idea can be implemented in conjunction with one of the previous two techniques: either placing the highest order bytes of all array data in protected memory or implementing error detection only across the highest order two bytes and leaving the remaining bytes unchecked.

5. Conclusions and Future Work

We have raised the problem of transient faults in graphics architecture. We provide the first attempt to characterize the impact and nature of soft errors in the graphics workload. Transient errors have been an important concern in general purpose architecture for nearly three decades. They have not yet posed a problem for graphics processors, but as technology processes continue to shrink, the soft error problem will soon be relevant in the graphics domain. We believe that the time to start thinking about it is now.

Our specific contributions include:

1. We have shown that AVF is a poor metric for vulnerability in GPUs, because it treats all errors that affect the final result as equally intolerable. GPU workloads are more forgiving—many errors can be ignored or corrected after the fact—but also more complex. To illustrate these points, we have performed experiments (using a test trace from a state-of-the-art game) to analyze AVF and investigate fault injection results.
2. We have proposed the Visual Vulnerability Spectrum to classify transient faults on the GPU. The VVS extends and generalizes the traditional AVF, characterizing faults on three orthogonal axes: Extent, Magnitude, and Persistence. State that falls high on two or three of these axes is important and may require protection.
3. We have presented several simple protection and recovery schemes for some graphics memory. The novel, graphics-specific solutions piggyback on existing mechanisms in the graphics workload. These schemes exploit the tolerance of the graphics workload by performing error checking off of the critical path, imposing little to no overhead. Since most GPU state is backed up in reliable storage on the other side of the PCI-E bus, these schemes also take advantage of reliable backing store for recovery.

Our analysis identified critical graphics state based on the application of the Visual Vulnerability Spectrum to the OpenGL state vector. Real GPUs do not directly implement OpenGL, as the driver translates the calls into hardware-specific instructions. While most of the OpenGL state we identified requires analogous state on the GPU, there is likely additional state that is critical as well.

Our treatment of the VVS has been mostly qualitative, with the goal of arguing for a graphics-specific treatment of soft-error protection. Future work needs to better identify important state and quantitatively evaluate its importance using the VVS, including detailed fault injection studies.

The protection techniques we proposed can probably be improved upon: more effective and efficient protection techniques are another important area for future work. Cost-effective ways to support full protection are also needed for GPGPU and other applications that do not offer the visual latitude assumed here. Application of ideas from the Redundant Multithreading [MKR02] literature look like viable solutions in such domains, especially when it becomes important that combinational logic, in addition to state, be protected.

6. Acknowledgments

This work was supported in part by NSF grants CCF-0429765, CCR-0306404, the Army Research Office under grant no. W911NF-04-1-0288, a research grant from Intel MRL, and an ATI graduate fellowship. We would like to extend out sincere thanks to the anonymous reviewers for their detailed and helpful comments.

References

[Bor05] BORKAR S.: Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro 25*, 6 (Nov./Dec. 2005), 10–16.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., AHERN S., FRANK R., KIRCHNER P., KLOSOWSKI J. T.: Chromium: A stream processing framework for interactive graphics on clusters of workstations. *ACM Transactions on Graphics 21*, 3 (July 2002), 693–702.

[HKM*03] HAZUCHA P., KARNIK T., MAIZ J., WALSTRA S., BLOECHEL B., TSCHANZ J., DERMER G., HARELAND S., ARMSTRONG P., BORKAR S.: Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25-μm to 90-nm generation. In *IEEE International Electron Devices Meeting 2003 Technical Digest* (Dec. 2003), IEEE, pp. 523–526.

[MER05] MUKHERJEE S. S., EMER J. S., REINHARDT S. K.: The soft error problem: An architectural perspective. In *HPCA* (2005), IEEE, IEEE Computer Society, pp. 243–247.

[MKR02] MUKHERJEE S. S., KONTZ M., REINHARDT S. K.: Detailed design and evaluation of redundant multithreading alternatives. In *ISCA* (2002), IEEE, IEEE Computer Society, pp. 99–110.

[Nor96] NORMAND E.: Single event upsets at ground level. *IEEE Transactions on Nuclear Science 43*, 6 (Dec. 1996).

[P*06] PAUL B., ET AL.: The Mesa 3-D graphics library, 1993–2006. http://www.mesa3d.org/.

[SA04] SEGAL M., AKELEY K. (Eds.): *The OpenGL Graphics Systen: A Specification (Version 2.0 - October 22, 2004)*. Silicon Graphics Inc., Oct. 2004.

[SABR04] SRINIVASAN J., ADVE S. V., BOSE P., RIVERS J. A.: The impact of technology scaling on lifetime reliability. In *DSN* (2004), IEEE, IEEE Computer Society, pp. 177–.

[Sam05] SAMSUNG ELECTRONICS: 256Mbit GDDR3 SDRAM: Revision 1.8, April 2005.

[ZCM*96] ZIEGLER J. F., CURTIS H. W., MUHLFELD H. P., MONTROSE C. J., CHIN B.: IBM experiments in soft fails in computer electronics (1978–1994). *IBM J. Res. Dev. 40*, 1 (1996), 3–18.

[Zie96] ZIEGLER J. F.: Terrestrial cosmic rays. *IBM J. Res. Dev. 40*, 1 (1996), 19–39.