A Texture Cache

Jonathan Blow **jon@number-none.com**
Bolt Action Software
March 20, 1998

To appear in the Proceedings of the 1998 Computer Game Developers Conference

Abstract

This paper describes a texture caching system for a 3D action game; the purpose of the texture cache is to manage the texture RAM of a hardware accelerator. The specific implementation we discuss is designed for 3Dfx Voodoo-based accelerators, using the Glide API to manipulate the texture RAM. However, the design is applicable to any API that provides linear access to texture memory.

I: Introduction

This paper jumps straight to the point and describes a recently implemented texture manager for Voodoo-based accelerators. We assume that the reader understands the motivation for such a system and is familiar with all the concerns the system must address. Those wishing more background on the subject should see "Implementing a Texture Caching System", *Game Developer* magazine, April 1998. That article serves as a lengthy introduction to this paper; we will refer back to it from time to time.

The Goal

The goal of the caching system is to minimize the quantity of data downloaded to the accelerator's texture RAM. Furthermore, the downloads should be *leveled*, that is, occurring evenly distributed over time inasmuch as that is possible.

Texture downloading is a relatively expensive operation; so if downloads occur in an unlevel fashion, the game's frame rate will stutter, which is an ugly sight. A cache will cause wasteful downloads if its replacement policy does not fit the game's texture access patterns, and often these downloads will be pathologically clumped. We'll see a clear example of this in a moment; avoiding pathological cases is a big part of The Goal.

Design Parameters

The caching system detailed here was designed for a game called *Wulfram. Wulfram* is a multiplayer vehicle combat game that takes place in an outdoor environment. The game typically draws a landscape, a sky (which is a box of polygons around the landscape), entities (represented by BSP trees holding static polygon data and also by voxel grids) and billboards

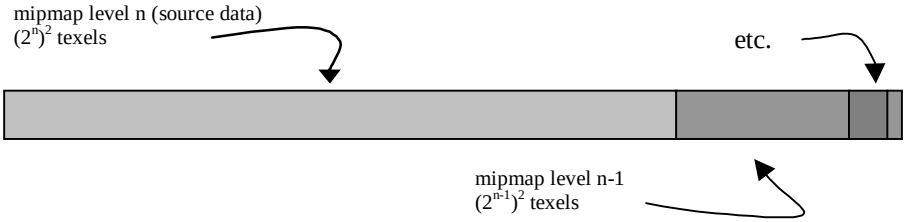
(semitransparent polygons that directly face the user and are animated, representing explosions, smoke, etc.) Screenshots of *Wulfram* can be found in the *Game Developer* article.

A *Wulfram* scene will contain a few hundred to a few thousand polygons, depending on the detail level chosen by the user. Most of these polygons (50%-90%, based on scene content and detail level) are landscape. Textures for the landscape are dynamically generated by the 3D engine, largely to facilitate detail reduction (if two polygons are combined into one, we need a "larger" texture to map onto the new polygon). Because of this dynamic generation and related factors, there are a large number of individual textures in each scene; the ratio of unique textures to displayed polygons is about 0.6 : 1.0.

There is no far clipping plane in *Wulfram;* players can see to "the edge of the world". This means that many polygons will be displayed at small mipmap levels, so the cache will be required to handle small textures in an efficient manner.

All textures used in the game are power-of-two and square, a fact which can greatly simplify the design of a cache (though, as it happens, neither power-of-twoness nor squareness were required by the system we ended up building.)

Most textures used in the game are 128x128 texels and 8 bits in depth; a few textures are 256x256 and a few are 64x64. For each texture, all mipmap levels down to 1x1 are computed using Floyd-Steinberg dithering and stored together with the highest resolution (the "source data"). (Figure 1).



Total size: $\sum_{i=0}^{n} 2^{2n} = \left(\frac{2^{2n+2} - 1}{3} \right)$

Figure 1: Arrangement of a single texture in RAM. Successive mipmap levels are stored contiguously.

Legacy System

When we first wrote the Glide code for *Wulfram* we were mainly interested in getting it working, without much regard for performance. On 3Dfx's developer web site was a simple code example that provided a primitive form of texture caching, so we wrote our own version of their algorithm and used that.

The algorithm was simple: treat texture memory as a wrap-around buffer. When you need to put a new texture into the cache, upload it at the current position in the buffer. Then advance the current position to the end of the texture. If the texture you're about to upload is going to overwrite any textures currently in the cache, throw them out. (Figure 2).

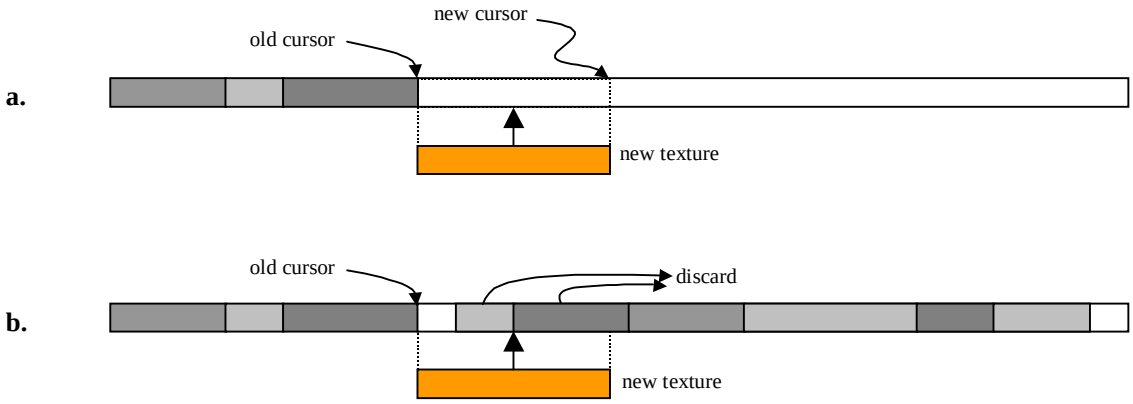


Figure 2: A simple texture caching method. In (a), cache memory is mostly empty. The "old cursor" points to the beginning of the free space. When we need to store a new texture, we store it at the old cursor, and update the cursor position. In (b), cache memory is full of textures. We store the incoming texture in the same way as (a), but we first kick out two textures to make enough free space.

This caching scheme is great because it is easy to understand and easy to implement, and it places no restrictions on textures size or geometry. At first glance it doesn't look like an excellent performer – clearly you would occasionally discard textures you were still using – but it's not too bad, right?

Let us examine the behavior of this cache algorithm in a typical game situation and bask in the wonder of its suckiness.

First we must ask: "What is a typical game situation anyway?" We consider a simple case, when the scene is mostly still. If the 3D engine draws a given object in the same position each frame, then unless it's a very weird engine, it will output the same polygons in the same order every time. If you think about this for a while, you may conclude the following: though it's certainly not true that successive frames of a 3D game contain the same polygons output in the same order, chances are quite high that, if we output some sequence of polygons in one frame, they'll appear in the next frame too, in roughly the same order. This is a specific instance of a very old idea in computer graphics, usually called "frame coherence" [FV90].

Because of frame coherence, if we output textures in the order { *a*, *b*, *c*, *d*, *e* } in one frame, we'll probably do it again in the next frame. Now let's look at the cache algorithm described above and see what happens when we output a texture *z*, causing *a* to be overwritten, right before we begin the *a*-*e* sequence in the next frame; in other words, now we're outputting { *z*, *a*, *b*, *c*, *d*, *e* } (Figure 3.)

Every new texture download will clobber the texture we're about to use for the next polygon. The result is that we end up downloading all of { *a*, *b*, *c*, *d*, *e* } even though we just had them all in cache and could clearly have made a much better choice.

Having shipped this algorithm to actual users (in a beta!), we can state with full force of experience that the above case is not just a pessimistic hypothetical situation; it happens *all the time* during routine operation. On a 2MB Diamond Monster 3D card using 8-bit textures, we would occasionally see spikes of 800kb-1200kb of texture downloads per frame, lasting for several frames; and if any motion was occurring at all, downloads of 200k-400k per frame were common. (This is already with plenty of code in place to ensure that we upload textures only at mipmap levels that are in use.)

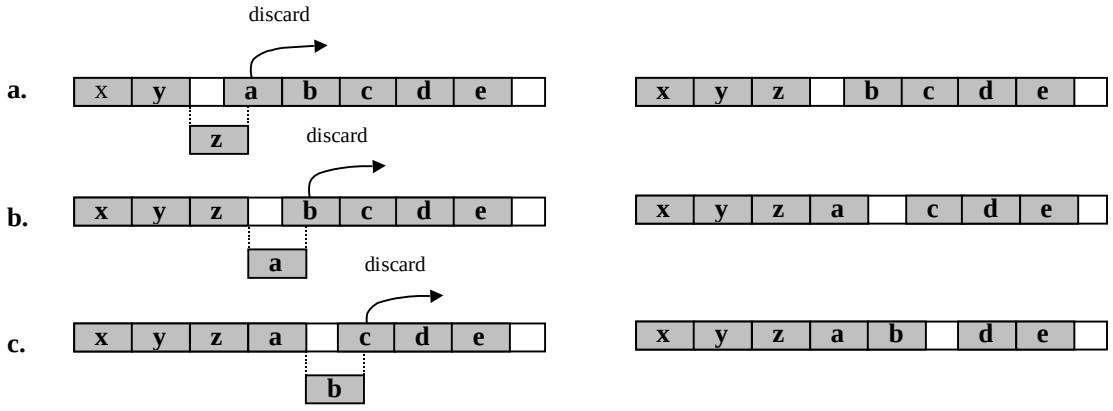


Figure 3: We output the sequence { z, a, b, c, d, e } at a very unfortunate time. In part (a), the output of *z* kicks *a* out of the cache. In (b), the subsequent output of *a* kicks *b* out of the cache. The pattern continues in (c).

None of this, of course, is a slight to 3Dfx. They never claimed that their code sample presented a *good* cache algorithm; mainly they provided it as an example of how to use the Glide texture downloading routines. They clearly expected developers to implement their own systems.

But there is one thing about this cache algorithm that we had damn well better notice before we go off and try to make something superior. Say we have a total of 26 textures in our game, *a*-*z*, which we output in sequential order each frame, and texture RAM is only big enough to fit 25 of them. Just like in figure 3, when the time comes to output *z* it will overwrite *a*, and then we will overwrite *b* with *a*, causing that same chain reaction. But because each frame consists of outputting *a*-*z*, then when it comes time to output *z*, texture *a* is the *least recently used* texture. [Silb91]. So if we built a cache that used the ever-popular LRU replacement policy, it would behave in exactly this way: our performance would suck to high hell.

This is a crucial example of "a little knowledge is a dangerous thing": most people who have read some stuff about caching "know" that LRU is a catch-all replacement policy that delivers good performance in most cases. But any good cache book will also provide

examples of how LRU (or any other deterministic algorithm) can generate pathological cases.

This section is growing long so we will baldly assert the next fact and leave the reader to think about it: LRU is a good replacement policy when we're considering textures that have been unused for more than one frame. But once we begin outputting more textures than can be held in RAM at once, LRU is an awful scheme, close to the worst we can formulate.

So with all that in our jumbled little heads, we set out to build a system for *Wulfram*.

Archetype System

The architecture of the texture caching system created by Outrage Entertainment for *Descent 3* seemed like a good fit for *Wulfram.* They had designed their cache for square, power-of-two textures, in a way that could handle small textures efficiently. So we decided to start by building our own version of Outrage's cache system and seeing how it worked. Their system is described in the *Game Developer* article but we present a quick recap here:

Texture RAM is divided into a number of separate spaces, which we will call "arenas" here. Each arena will hold textures of one mipmap level; so there will be one arena for 128x128 textures, one for 64x64 textures, and so on. The arenas are sorted in texture RAM, with the lowest-resolution arena "on the left" (occupying the lowest range of memory addresses) and the highest-resolution arena on the right. (Figure 4a.)

Because each arena stores only textures of the same size, an arena can be subdivided into "blocks" each of which holds exactly one texture. Because each block is the same size as the others in its arena, it is impossible to have memory fragmentation within an arena. This design decision is very attractive because it eliminates a lot of messy heap management issues.

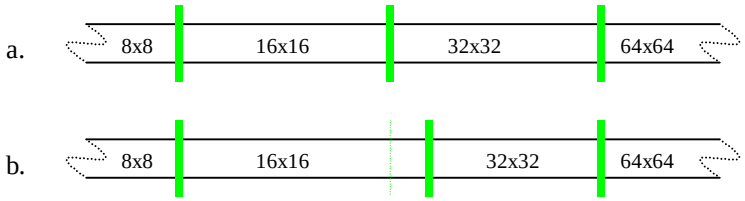


Figure 4: Descent 3's memory management scheme. **a:** Memory is partitioned into contiguous regions based on texture size. **b:** The 16x16 region has grown by taking space from the 32x32 region.

The walls between arenas are allowed to slide to the left and right to accommodate changing texture usage patterns. For example, suppose that the 32x32 arena is full. The arena can grow by stealing memory from the 16x16 arena to its left or from the 64x64 arena to its right. When stealing from the left, an arena must take enough memory for four of its neighbor's textures to hold one of its own. When stealing from the right, an arena may take only one of its neighbor's blocks, yielding enough RAM for four of its own. (Figure 4b.) The choice of

when to grow an arena, and in which direction to grow it, was made by a set of heuristics the details of which we did not know.

II. Our Cache Implementation

First Attempt

We set out to build a system similar to the cache made by Outrage, filling in the necessary details (such as arena growth heuristics). After about an hour of implementation, though, we stopped to reconsider.

We had planned to use the Glide facility of "texture multibase addressing" to implement the cache. Multibase addressing simply means that different mipmap levels of the same texture do not need to be stored contiguously in memory; they can be anywhere in texture RAM so long as their locations are registered. Multibase addressing is not the default mode of operation for Glide, but it can be enabled with **grTexMultibase()**. Because we knew that multibase addressing existed (though we had never used it) we figured there would not be a problem.

Of course there was a problem! Glide only permits specification of separate base addresses for the top 3 detail levels (256, 128, and 64 texels). All the rest (32 on down) must be lumped together into contiguous memory. This would destroy the neat organization of Outrage's memory scheme, which wanted each arena's block size to be four times as large as the block size below it. Also it would create many special cases in the texture caching code, which would probably take a while to debug and which would make further experimentation more complex and difficult.

A simple way around this problem would be to treat each mipmap level as a logically separate texture when using Glide. So, for example, instead of declaring a texture that has mipmap levels at 64x64 and 32x32 texels, we would declare two textures, one of size 32x32 and one of size 64x64, each of which would have only one mipmap level. Multibase addressing would not be required since each texture would be atomic.

This approach has serious drawbacks: it effectively disables some of the chipset's graphics features, such as per-pixel mipmapping and trilinear filtering. Both of these features require the hardware to access different mipmap levels of the same texture; obviously, if we tell it that there are no different mipmap levels for each texture, it cannot perform these features.

We didn't care too much about trilinear filtering, but we really wanted per-pixel mipmapping. So we retreated and redesigned.

Current Version

One nicety of Outrage's system was that, besides the fact that there could be no fragmentation within an arena, there could also be no fragmentation between arenas. This is because every block's size was four times the size below it; memory could be moved around in a way that guaranteed a perfect fit. This kind of "divide-by-four" cleanliness is the sort of thing we programmers find aesthetically pleasing and hard to let go.

After a quick bit of analysis, however, we decided that this design feature was an unnecessary frill of Outrage's system. If we allow block sizes to be arbitrary (though still of a constant size within each arena), then when an arena becomes overcrowded, we can just steal memory from the left or right until we have enough; afterward we will leave an area of temporarily "wasted" space between the two arenas, whose size is guaranteed to be less than the block size of the leftmost arena involved in the memory swap. This fragment space will be taken up again next time the arena barriers move. (Figure 5.)

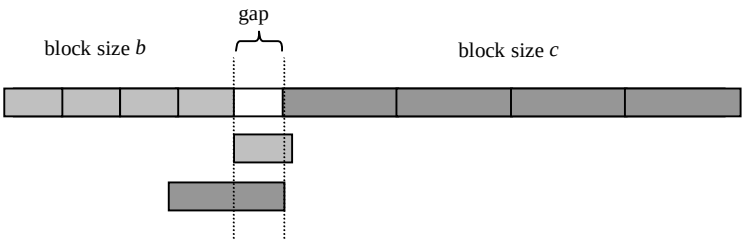


Figure 5: Between two arenas, one with block size *b* and one with block size *c*, there will be a gap. The gap must be smaller than *b*, otherwise it would be taken up by the arena on the left. It must also be smaller than *c*, otherwise it would be taken up by the arena on the right. Therefore the gap is smaller than Min(*b*, *c*).

In a system with 9 arenas, one for each mipmap level, where the size of a block of mipmap level *n* is $(2^n)^2$ (rounded up to the nearest 8 bytes because Voodoo hardware seems to enjoy addresses that are 8-byte aligned), then the total amount of "wasted" space at any given time must be less than or equal to:

$$\begin{aligned}&\left(2^7\right)^2+\left(2^6\right)^2+\left(2^5\right)^2+\left(2^4\right)^2+\left(2^3\right)^2+\left(2^2\right)^2+\left(2^1\right)^2+\left(2^0\right)^2+0\\&=16384+4096+1024+256+64+16+4+1\\&=21845\text{ bytes}\end{aligned}$$

The smallest amount of texture RAM a Voodoo card supports is 2 megabytes; so the worst fragmentation possible is (21845 / (2 * 1024 * 1024)) ≈ 1%, a perfectly acceptable quantity as an upper bound.

So we elected to use block sizes that were not multiples of each other. Specifically, we stored textures in the Glide **TEXFMT_P_8** format; when we downloaded a texture to the card, we would ensure that all mip levels from 0 to *n* were present (where *n* is the number of the arena we are downloading into). The size of each block is then given by the **SizePlus** function described in the *Game Developer* article (which is the same as the equation given in Figure 1), with the exception that all block sizes are rounded up to the next multiple of 8; a table of block size by mip level is given in Figure 6.

Using these block sizes, our max fragmentation will be (21848 + 5464 + 1368 + 344 + 88 + 24 + 8 + 8) – (8 * 8) = 29088 bytes, for a slightly higher max fragmentation than in the earlier case (almost 1.4%).

max mip level	texture size (bytes)	block size (bytes)
0	1	8
1	5	8
2	21	24
3	85	88
4	341	344
5	1365	1368
6	5461	5464
7	21845	21848
8	87381	87384

Figure 6: The block sizes used in the *Wulfram* cache. Each block size (third column) is the texture size (second column) rounded up to the next 8-byte boundary.

Growth Heuristics

To start with, we implemented a version of the cache which did not allow the arenas to resize themselves. By using a graphical texture download monitor we could see that already this system performed better than 3Dfx's sample code, except when an arena became full, in which case textures that were being used would always be kicked out of the cache, causing a vicious cycle. We toyed briefly with different partitionings of texture memory but no set of arena sizes seemed to avoid the problem. So, as we knew would be necessary, we set out to make the arenas grow and shrink in a reasonable way.

To come up with our heuristic, we used an analogy that is common in computer science: temperature. An arena that is "hot" wants to expand; an arena that is "cold" wants to contract. An arena that is too hot – one in which textures are constantly being thrown out prematurely – is "boiling".

For the "temperature" of a region, we measured the percentage of that arena that was occupied by active textures, represented as a floating-point number between 0 and 1. In the case when "boiling" occurs, we add an artificial bonus to the temperature each time a texture is wrongfully ejected from the cache. Therefore, the more a region is boiling, the higher its temperature will be above 1. (Figure 7.) This is an abuse of the term "boiling", but hey.

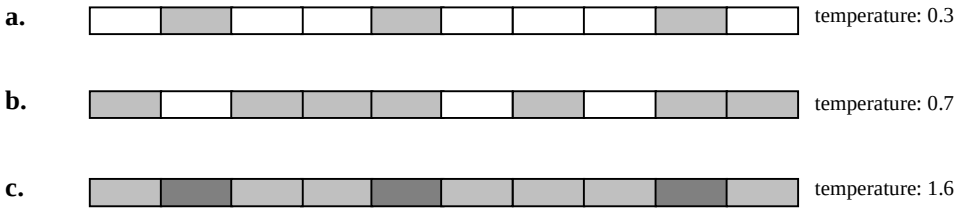


Figure 7: Arenas at different temperatures. The white cells represent unallocated blocks; the shaded cells represent occupied blocks. Arena (a) is coldest at 30% occupation; (b) is warmer at 70%. Arena (c) is "boiling"; the darkly shaded cells represent blocks where active textures had to be discarded to make room for new textures. This is bad and the badness is reflected in (c)'s very high temperature.

Because we wanted the cache system to remain stable over time, not to be perturbed too greatly by instantaneous oddities in texture usage statistics, we decided to use the above method to compute the *instantaneous temperature* of each arena; but that instantaneous temperature would not be used directly. Instead, we introduced the idea of *recent temperature*, which is a time-averaging of instantaneous temperatures. Specifically,

$$recent_0 = 0$$
$$recent_n = recent_{n-1} * 0.7 + instantaneous_n * 0.3$$

This way, temperature does not change too discontinuously, so the cache does not "thrash" when usage statistics change dramatically from frame to frame.

Once per frame, during the "cache update", we modify the sizes of the arenas. Each arena looks at the average temperature of everyone in the cache to its left; if that temperature is substantially lower than the arena's *recent temperature*, the arena expands to the left. The arena then does the same thing to the right.

Because the arenas consider temperatures that are averaged over the entire memory aperture instead of just their neighbors, the cache system will discard active textures in the short term in order to achieve longer-term stability. See Figure 8.

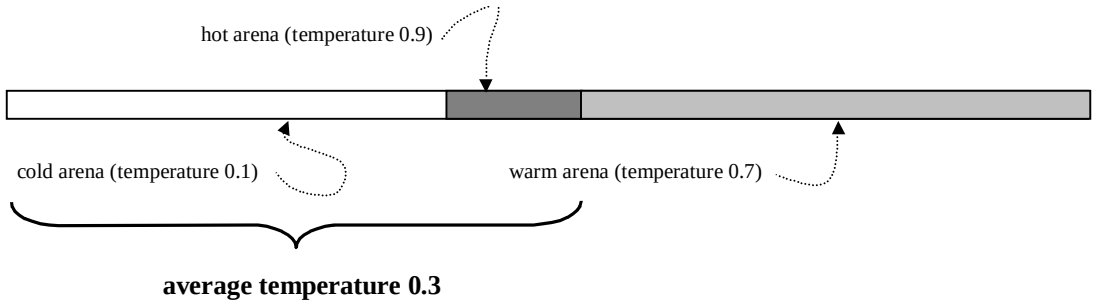


Figure 8: The arena on the right will expand to the left because the average temperature to its left is 0.3. In expanding it will take memory from an arena hotter than itself.

It is worth noting that this simple scheme is capable of preemptively sensing and responding to imbalances of memory allocation. For example, if an arena is very hot, it will begin expanding even though it has not yet filled; so, in most cases, emergencies are averted before they occur.

Placement and Replacement Policies

When choosing an empty block to store a new texture, it would be a bad idea to use a block that is about to be yielded to a hotter arena; the texture would just be knocked out of the cache and would need to be downloaded again, decreasing the cache's efficiency. To avoid this, each arena remembers the temperature measurements it took during the last cache update and prefers to allocate blocks that are closer to the cooler side of the cache. (Blocks which are on the absolute left and right of the memory aperture, having only one neighbor, always prefer to allocate on the side without a neighbor.) See Figure 9.

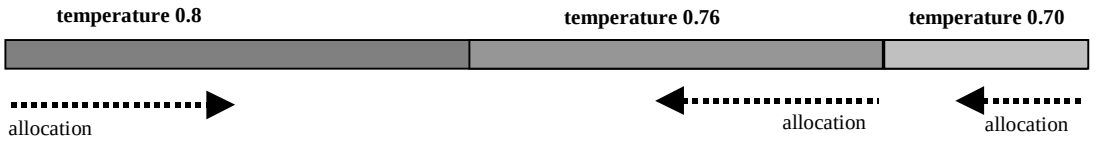


Figure 9: The arena on the left is at the left end of texture RAM, so it will prefer to allocate textures on its left side. The arena next to it (temperature 0.76) prefers to allocate from the right because texture memory is cooler on that side.

When we need to throw out a texture that is in use (i.e. an arena is boiling), we simply choose a random block within the arena and discard its texture. The "random" replacement policy performs nicely in this case, avoiding the problems caused by LRU.

III: Conclusions and Future Work

Overall Performance

For the current version of *Wulfram* the cache performs excellently. It is extremely efficient at allocating texture RAM and exhibits level performance statistics. We can change scene types abruptly (from, say, 300 polygons with high-resolution textures to 9000 polygons with small textures) and the cache will quickly adapt.

Because blocks within an arena are all the same size, allocation and deallocation of textures is a nearly instantaneous process.

Potential Performance Issues

One reason that this scheme works well for *Wulfram* is that we have only 9 possible texture sizes, so the overhead involved in maintaining the arena spaces is minimal. In a system that

required a much larger number of texture sizes, the usefulness of this type of cache would break down. In the extreme case, we can imagine a cache with hundreds of arenas, each of which contains only a few textures. Since there are now many arenas, there will now be an unacceptable amount of wasted space between the arenas. But even worse, in this case we must regard some unoccupied blocks within the arenas as fragmentation (since the mean time to block usage for any given block size will be very high), and suddenly our cache design is a very poor performer. For an engine that requires the use of arbitrary texture geometries (such as the KAGE engine outlined in the *Game Developer* article), this type of cache is wholly inappropriate, and indeed it is difficult to imagine how to apply the idea at all.

Furthermore, with any number of block sizes, it is possible for two arenas to "fight" each other, where one arena expands into another arena, which then expands back into the first

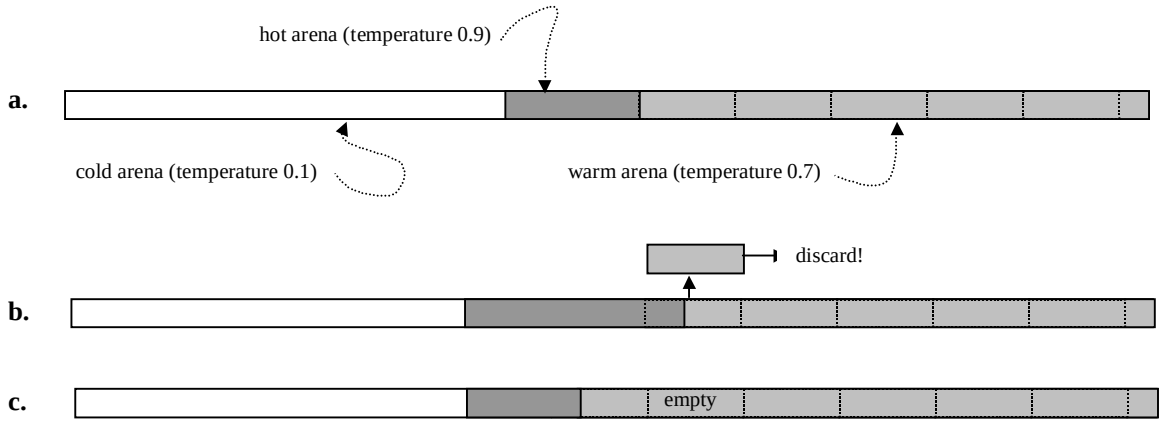


Figure 10: Looking back at the situation from Figure 8: If, during the course of our arena maintenance, we expand the "hot" arena (b) into the warm arena, which itself wants to expand, we may discard a texture in the warm arena that we shouldn't have. This kind of case is rare (the allocation preferences help avoid it). It can be handled with careful programming.

arena. This problem is not fatal because, provably, the cache algorithm reaches equilibrium in the steady state. Also, we have not seen this phenomenon occur in noticeable amounts during gameplay. However, it is probably responsible for a few extra texture discards whenever an arena boils (Figure 10).

Also, there are some clear cases when this algorithm is less efficient than the blind heap management given to us by higher-level APIs. Imagine that we start with an empty cache; then we output into Arena 5, say, twice as many textures as will fit there (though they would still fit into all of texture RAM taken as a unit). Even the much-maligned algorithm at the beginning of this paper will handle this case with no problem, but our algorithm will end up kicking some textures out of the cache while it adjusts. This is a bummer, but we console ourselves with the fact that this is a bizarre scenario unrepresentative of typical conditions.

Is this algorithm applicable to future hardware?

Earlier we have said that this algorithm requires linear addressing of texture memory, but that isn't strictly true. The algorithm uses block uploads to put textures into memory, and this is the abstraction used by most hardware with nonlinear addressing.

Within a texture block, byte order could be random for all we care. And since block sizes are arbitrary, we can choose a block size that works for particular hardware if such a size exists. (For example, given hardware where memory proceeds in 64-byte chunks, the interiors of which are arbitrarily ordered or unaddressible, we need merely ensure that our block sizes are all multiples of 64 bytes.)

Certainly we can concoct hardware for which this algorithm is inappropriate. Generally, though, it can be made to work, provided that the API gives us some form of addressing.

Is all this even a good idea?

We've got games to write here, and games are getting bigger and more complicated all the time. Is it justifiable to spend time and effort working on a custom texture cache, when driver writers can theoretically spend more effort and do a better job?

For now this may be a moot question. Developers using OpenGL (or inferior APIs such as Direct3D) do not have the ability to directly manage texture memory, so they cannot implement algorithms such as the one described here. Instead they must tell the API which textures to allocate and deallocate without knowledge of the effect such operations will have on the fragmentation of the heap. Of course, it is the driver writer's job to make sure that the driver is very smart about keeping heap fragmentation minimal; but this is a very hard problem (creating a good memory management system that works for anyone who might wish to use the driver) whereas the author of a game is confronted with a much easier problem (creating a good memory management system that works for one game's access patterns).

In any case, at this time it is clear that some developers prefer coding to a native API for a few major (market-dominating) cards, using a higher-level API such as OpenGL to cover the rest. In the case when coding to a native API, a good cache is probably not available, so making one is probably a good idea.

And

As it stands, the source code for our implementation uses some idioms of our own 3D engine, so it is not quite useful to the general public. However, if there is enough interest (expressed via e-mail), I will spend the effort required to abstract the code and release it to the public. The basic idea of this kind of cache could certainly be taken much further, and it would be interesting to see people experimenting with it.

Acknowledgements

Thanks once again to Jason Leighton of Outrage Entertainment for sharing the details of *Descent 3*'s cache system.

References

[Blow98] J. Blow, “Implementing a Texture Caching System,” *Game Developer* magazine, April 1998.

[FV90] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *“Computer Graphics Principles and Practice,”* 2nd ed., Addison-Wesley, 1990.

[Silb91] A. Silberschatz, J. Peterson, and P. Galvin, *“Operating System Concepts,”* 3rd ed., Addison-Wesley, 1991.