The KAGE Surface Caching Mechanism

Last modified: December 13, 1997

INTRODUCTION

A surface cache, as many are probably already aware, is used to speed up the rendering of, typically, illumination maps blended with textures. Typically, each polygon in the scene is assigned a texture (which may be shared with other polygons in the scene) and an illumination map (one for every polygon in the scene.)

By pre-calculating static illumination maps for each polygon in a scene, and storing them as low resolution textures, they can be blended with the textures and stored in the surface cache for fast texturing during the rendering process.

The need for separate textures and illumination maps is due to memory constraints. Storing a lower resolution illumination map for each unique polygon in the scene with shared textures requires much less memory than storing a full resolution blended texture for each unique polygon.

The blending process is not a fast process, especially for true-color illumination maps (as in the case of KAGE.) So the need to cache the blended surfaces becomes necessary for software renderers that are blending more and more complex textures with higher resolution light maps.

Many techniques used in surface caching today rely on the fact that the cached surfaces are on power-of-two boundaries. This is very common in realtime graphics programming since it simplifies many aspects of the 3D systems. However, given the nature of KAGE's generic dataset (our environments come from modellers like 3DS Max at the moment) a power-of-two assumption isn't necessarily the best assumption. This should become clear in the sections to come.

To solve this, a somewhat dynamically sized surface cache had to be allocated to take advantage of these odd-sized textures.

DEFINITIONS

A texture is a 2-dimensional bitmap of a given bit depth.

A surface refers to a polygon as it appears in UV space. So in the same way a polygon appears in 3-space by the given 3D points, a polygon appears in UV space (as a "surface") by the given vertex UV points. To explain it in a different way, a surface could be viewed as if it were drawn onto the using the polygon's UV coordinates.

Finally, I'll be using the term "texel" quite a bit. This refers to a single texture pixel.

PREPARATION

Before we can even consider loading and displaying a scene that uses a surface cache, there is a bit of pre-calculation that needs to take place. The most obvious is the generation of the illumination maps. KAGE uses a high-end radiosity processor, written in-house, for this. However, any other popular lighting methods (such as ray tracing) may also be used.

Aside from that, the polygons must also go through a pre-processing step. The purpose of this pre-processing step is to prevent any polygon from occupying too much texture space. In other words, we want to limit the size of our surfaces, as they appear in UV space.

For example, if we were given a polygon that was 100 miles long, and 5 feet tall, then we'll end up with a long sliver of texture space that the surface occupies. This might translate to a sliver of texture space that is 520,000 texels long and 5 texels tall. This is quite unmanageable, and also quite unnecessary, since a surface of that size will probably not be completely visible at once. So we need to determine a good maximum limit for our surfaces and split up the polygons in the scene so that they do not occupy too much texture space. I won't be getting into any detail on this, but to fully understand the purpose of a surface cache, and how to successfully manage one, you need to understand a little bit about texel density...

A NOTE ON TEXEL DENSITY

Texel density refers to the way in which a polygon is textured. A low-density mapping gives polygons a blocky look when viewed up close. By increasing the texel density of a surface, texels become smaller and give the polygons more texture detail when viewed up close (see figure 1).

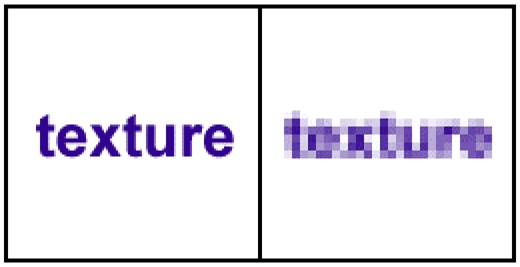


Figure 1. Two polygons mapped with differing texel densities. The polygon on the left has four 16 times the density than the polygon on the right.

I learned early on that control over the texel density of each surface in the scene is paramount to a successful surface cache. The texel density is directly proportional to the size of the surface cache for any given view of a scene (prior to mip mapping.)

Extending this, texel density is also paramount to keeping the polygon counts low. If the texel density is too high, then the polygons are split more often due to the 256x256 surface size limitation. I considered extending that limitation to 1024x1024 to reduce splitting, but that could cause problems when it came time to blend the large surfaces into the cache. The larger the surface in the cache, the longer it would take to blend into the cache, causing drop-outs in the frame rate, and preventing a smooth animation appearance. Alpha blending a 256x256 surface cache entry with bilinear filtering on the illumination map can be tricky enough, if you want a smooth frame rate.

Since KAGE's environments aren't built from convex polyhedra as many popular game editors (Quake, etc.) currently operate, but rather from a generic polygon mesh, we've had to use generic modellers. These modellers do not give us direct control over texel density in the same way that the typical editor does. Due to the existence of these editors, the concept of texel density may have been lost. But without an editor of that kind, texel density becomes an all-important factor. So the texel density across each surface must be adjusted.

Also, by adjusting the texel density for a scene, you can control the aspect ratio of the textures as they are applied to the surfaces. Keeping a square aspect ratio on the textures usually means a clean rendering.

With a square texel aspect ratio on all surfaces, we may get a clean rendering, but this is not necessarily what the artists intended when they created the scene. Some surfaces may require a higher density texture to be applied. This can be accomplished, but the higher the density, the more surface cache space is required. So all surfaces are assigned an ideal texel density as a reference point, then the artists are given the task of adjusting the textures by stretching (adjusting the density), shifting and rotating them, until the scene is pleasant to look at. This is also much easier than performing the texturing in a generic modeller.

OUR EXAMPLE

This text is based on the example of KAGE. We already know that for this example the surfaces are limited to 256x256 texels in size. For a little more clarity, I'll define a few other decision factors.

The typical texture in KAGE is 128x128. Since surfaces are 256x256 it can be derived that a texture may wrap as much as twice within a single surface.

The illumination maps used in KAGE are one fourth the width and height of the surface for any given polygon. So if a surface occupies 256x256 texels, then the illumination map for that surface would be 64x64. For comparison purposes, the illumination maps in Quake for a similar surface would be 16x16. By increasing the illumination map resolution, the resolution difference between the illumination maps and the textures becomes almost completely imperceptible in all cases, giving us excellent illumination detail. Add to this, the fact that the illumination maps are 16bpp.

This, however, causes us a lot of other problems. It costs a lot more, in terms of processor time to blend these surfaces into the surface cache. Also, it uses approximately 32 times the memory than a monochrome, 16x16 Quake illumination map, increasing the need for a very conservative surface cache mechanism.

By now, you should have a pretty good idea of what we're up against. We'll need a high-performance surface cache that maximizes its use of memory for surface cache entries of just about any size, up to the maximum of 256x256. Sounds like a simple task, but the random sized textures really increases the problem complexity exponentially if the surface cache is to be effective and waste as little memory as possible.

MIP MAPPING

Typically, without mip mapping, in a scene with a bunch of large surfaces, it is quite easy to over-extend the bounds of your surface cache with even the simplest scenes. Take for example a scene consisting of only 200 full-sized surfaces. With surfaces of 256x256, 16-bits each, we're talking about 25MB.

With the aid of mip mapping, however, that requirement can drop down to less than a megabyte, since the mip mapped version of a texture is cached, rather than the entire texture.

The mip mapping technique used in KAGE is an exact mip map level choice based upon the distance of a surface to the view, the texel density of that surface, the resolution, aspect ratio, the field of view, and some other parameters. This accomplishes the true mip map level at a given depth into the scene. This has the advantage of also being perspective correct. So a point travelling away from the viewer changes mip levels along the perspective curve. I only mention this because this is important for maximizing the use of surface cache RAM. This causes a very few number of surfaces to retain mip level 0 (full size) for any view. Surfaces quickly fall off to lower mip map levels in the scene, minimizing cache usage, and maximizing performance and visual quality.

ONLY CACHING WHAT IS NECESSARY

As mentioned above, the KAGE environment is simply a mesh of polygons. These polygons can be any size up to our maximum surface size of 256x256. With a relatively consistent texel density across all surfaces, it's not hard to realize that some surfaces don't use much texture space. Some small polygons may only use a 4x4 section of texture space, so that's really all we want to cache.

Herein lies a problem. We've just complicated the task of the caching mechanism ten-fold, by requiring that it cache a surface of literally any size up to 256x256. Is this really important? For KAGE, yes. I found that without this feature, it would take as much as three times the surface cache space to cache the same scene as it would with this feature.

At this point, it should be obvious that visibility algorithms also play a key roll in surface cache performance. The visibility algorithm's conservation (i.e. resulting in more polygons than are actually visible) will cause caching of more surfaces than are actually necessary. KAGE uses a realtime visibility set (RVS) that does not have this problem. So all statistics given will be dependent upon this factor.

DEFINING THE PROBLEM

We're almost ready to get into the surface cache. But before we do, let's get a recap of the requirements for our surface cache. The preprocessing steps must first split any polygons whose surfaces will extend through a 256x256 section of UV space. This, in turn, limits our surface cache entries to a maximum size of 256x256. Our surfaces may be any X by Y size, where X & Y refer to any number from 1 to 256.

In this example, all surfaces are 16-bits deep, and, for the sake of speedy texture mapping during the rendering process, each surface should be stored within a region of memory that is 256 pixels wide. This may mean that many surfaces are stored side-by-side.

Our inputs to the surface cache routine must also include the polygon's mip map level and U/V coordinates, so we only cache what is necessary, and nothing more.

THE SURFACE CACHE

The surface cache is primarily a collection of cache pages. Each page has the potential of containing one or more surfaces. Within KAGE, each page is typically 256 pixels wide and N pixels tall (where N can range from 1 to 256.) There may be one or more pages of a given height. Surfaces are stored in sorted order within their cache page via a singularly linked list. Surfaces are defined within the page by being part of the linked list (see figure 2.)

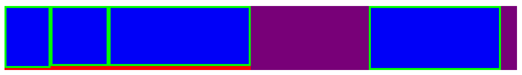


Figure 2. In this visual representation of a cache page, surfaces are outlined in green, red areas represent wasted space and purple represents available space.

Each list element (cached surface) contains two integer values, which depict the horizontal location within the page and the width of the element. Empty space within a cache page is denoted by the lack of an element covering that section of the page.

Surfaces are chosen for replacement by a simple LRU (Least Recently Used) algorithm. This is done by incrementing an access_count variable every time the cache is accessed. All surfaces in the cache have an MRU (Most Recently Used) value that is used to track their age. Each time a surface is accessed (either by being added to the cache or by being retrieved from the cache) the MRU is updated to the cache's current access_count.

As a surface is requested from the cache, a search is done on the cache to see if the surface currently exists. If so, the surface is returned. Locating surfaces in the cache is done quickly with a hash table. The hash table is maintained for every surface insertion into, or removal from, the cache.

When a surface is not found within the cache, the surface must be added. This could mean that an unused portion of a cache page is located and used, or a replacement of an existing surface (or set of surfaces) will occur.

The first step in this process is to locate the "potential" cache pages. These are pages that are closest in height to the new surface without being too small. The next step is to locate a "best fit" for the surface within the potential set of pages. "Best fit" may refer to a single surface, or group of contiguous surfaces. This is where the new surface will be stored.

During this search, unused portions of the pages are also examined. If an unused portion (gap) within a page is located that is large enough to contain the new surface, then the new surface is inserted at that point and returned to the user. Note that any gaps larger than the inserted surface will retain some free space, by the simple fact that the new element doesn't completely fill the space within the linked list of surfaces for the page.

The "best fit" search is a search for an existing surface, or group of contiguous surfaces (including gaps between them) that is large enough to contain the new surface, and which also has the oldest combined MRU value.

Once located, the "best fit" set is removed from the page, and replaced by the new surface. If the "best fit" is larger than the new surface, then the unused portion automatically forms a gap between the new surface and any surfaces following it. This gap is considered available memory and may be used by future surface allocations.

There is a potential for waste in this process since surfaces may only be placed side-by-side within a single cache page and not vertically. This can be calculated (in pixels) by:

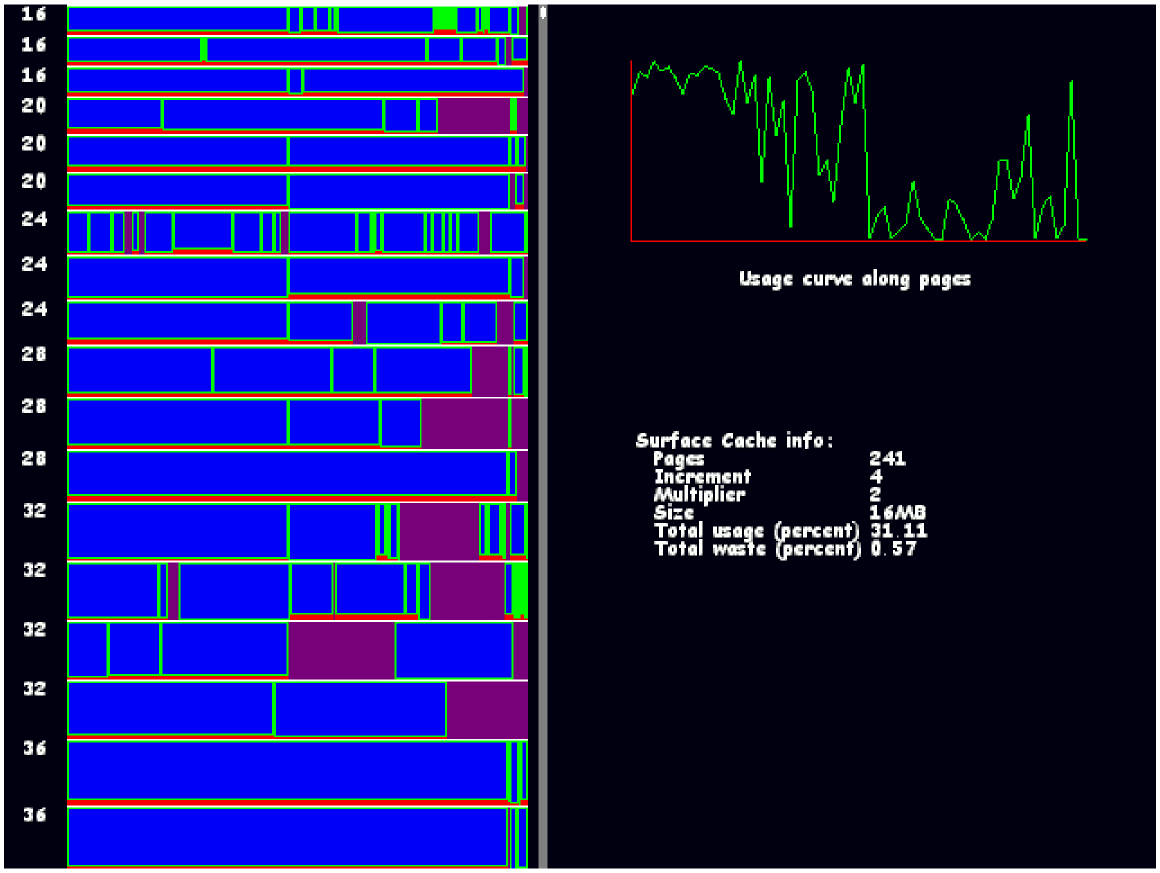
(page_height – surface_height) * surface_width

Note that if the surface is the same height as the page, there can be no waste. Any gaps between surfaces are not considered waste, since those gaps may be allocated by future surfaces without affecting their neighboring surfaces in the given cache page. By allowing surfaces to align on any pixel in the horizontal direction within a page, waste is minimized.

Extending this logic, it stands to reason that if there is one cache page for every possible height, the waste would effectively be zero in all cases. However, this has the drawbacks of lack of good cache usage and lots of storage space. I've found that, even with a difference of 8 pixels in size from cache page to cache page, the waste has proven very low. Smaller caches of 2MB tend to result in an average of 5 percent waste, while 8MB caches rarely see 2 percent, and 16MB caches are typically less than 0.5 percent.

This leads us to the allocation distribution algorithm, which is the key to a successful surface cache for this technique. The purpose of this algorithm is to intelligently pre-allocate the cache pages in such a way as to maximize the cache usage (see figure 3.) I have a few theories on this subject (each of which is valid within a given set of circumstances) but I'll cover the one technique used in KAGE.

Due to the effect of perspective correct mip mapping of the surfaces being cached, I've found that allocating surfaces based on a curve similar to the perspective curve to be very efficient. Very few full-sized pages are allocated (bottom of the curve) while many small pages are allocated (top of the curve.) This tends to give a relatively equal surface distribution throughout the range of surface cache page sizes.



```
Surface Cache info:
  Pages                  241
  Increment              4
  Multiplier             2
  Size                   16MB
  Total usage (percent)  31.11
  Total waste (percent)  0.57
```

Figure 3. Screen capture from the KAGE surface cache debugger displaying only a small sub-section of a full 16MB surface cache. Numbers to the left represent the height of each cache page, and the graph on the right represents the usage of cache pages from small pages (left) to large pages (right.) This graph looks unbalanced due to the short time in which the cache was in use, since cache performance improves over time. Note the very small amount of waste.

FUTURE WORK

There are many extensions to this algorithm, none of which I've taken advantage of, and will not take advantage of unless the need arises. The first of these extensions is the ability to dynamically re-allocate pages based on need within the cache. Pages receiving very few surfaces may be broken up into multiple smaller pages where needed (or multiple smaller pages combined to form a larger page as needed.)

Within the generic data sets typically given to KAGE, I often find long slivers of texture that tend to nicely fill the empty gaps within pages. However, there are two possible improvements to this.

Garbage collection could be done occasionally, shifting surfaces within a page to gather groups of small gaps into a single, larger empty slot at the end of the page.

Since there are many slivers of texture that fit nicely into the empty vertical gaps in the pages, it only makes sense that there will be about the same number of horizontal slivers. These slivers cause the most potential for waste within the smaller pages. By swapping UV values of the polygon, the surface could be stored vertically in the surface cache, reducing waste even more. I haven't bothered to implement this since the waste was so much less than I ever expected.

CONCLUSION

If you find this algorithm useful, or even better, have improvements for it please let me know. I can be contacted at: kage@terminalreality.com

Thanks for reading,
Paul Nettle
KAGE Developer