TEXTURE CACHING WITH MODIFIED QUAD-TREES

PAUL NETTLE
FLUID STUDIOS, INC.
HTTP://WWW.FLUIDSTUDIOS.COM/
AUGUST 6, 1999

OVERVIEW

This document describes a texture caching method that attempts to maximize the use of texture RAM. By utilizing a dynamic caching mechanism, texture space is shared between textures of varying sizes, allowing a single texture to replace multiple smaller textures when needed, in an efficient manner. This algorithm minimizes wasted space and includes an efficient method for finding the most optimal texture (or group of textures) to replace, with the lowest risk of overwriting an important texture.

Though generic enough to be used in a variety of applications, this algorithm has proven to be efficient. It can also be easily adapted to any of the standard 3D APIs (OpenGL, Glide & Direct3D) since it does not rely on texture allocations, except during the initialization phase. This also improves performance, as texture memory allocation tends to be inefficient for real-time use.

DESCRIPTION: TEXTURE STORAGE

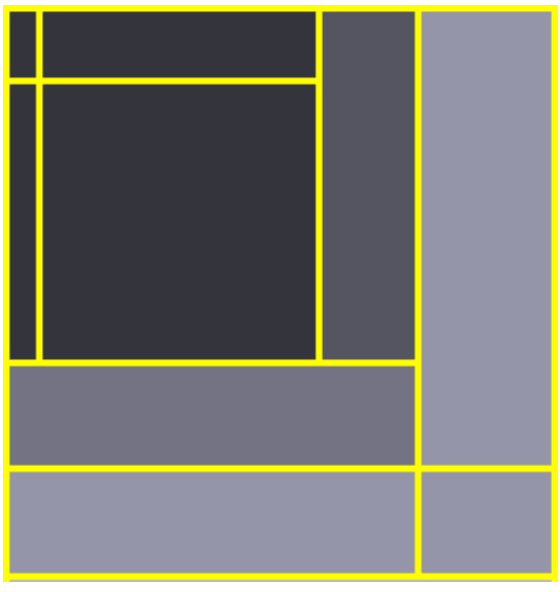
The algorithm pre-allocates a set number of surfaces (including all mip-levels) from the 3D API. These surfaces (called "Cache nodes") are only allocated once. They are used and re-used throughout the duration of the application. This avoids any overhead of texture space allocation, which can be costly. This also adds a layer of abstraction from the algorithm and the underlying 3D API.

The application may decide to allocate all of the available texture space to cache nodes. Each cache node can have its own individual size. In this document, we'll assume that our maximum texture size is 256x256. We'll also assume that each cache node is allocated at the maximum texture size, allowing any texture that the application might need to fit into any cache node. This gives us a texture-space balance across the cache.

Each cache node is capable of storing NxM textures (where N is the width of the cache node, and M is the height of the cache node.) The organization of textures within cache nodes will be managed by the use of a modified quad-tree.

DESCRIPTION: MODIFIED QUAD-TREE

The algorithm employs a modified quad-tree. This tree can be split in one of three ways: Horizontally (two children - stacked vertically), Vertically (two children - horizontally opposed) or the standard quad-tree split resulting in four children. Another modification to the tree is that the subdivisions are not always centered in the node, but may be offset in any direction (see figure 1.)



By storing the textures into a structure of this type, we're given enough freedom to store textures of any size with high efficiency. Any unused space remains available for future texture additions to the cache.

Figure 1: An example of the modified quad-tree, showing each of the possible subdivisions. Deeper levels into the hierarchy are noted with a darker hue

DESCRIPTION: CACHE INSERTIONS

Remember that the cache is comprised of several cache nodes, and each node represents a hierarchical data structure in which the actual textures are stored.

Insertions into this cache require visiting each cache node and traversing the quad-tree to locate a quad-tree node that is large enough to hold the new texture. If, during this process, we find an empty quad-tree node, we simply stop and use that node.

However, with more use, the cache will become more balanced, forcing situations where it will be necessary to replace a quad-tree node. Note that each quad-tree node may comprise multiple children, and hence, multiple textures. So replacing a quad-tree node might mean replacing a group of textures. To maximize efficiency, we'll want to minimize any replacements of newer textures. Or, more specifically, we want to replace the oldest quad-tree node (across the entire cache) that is just large enough to house the texture being inserted. Any excess space will then be marked as unused and will be made available for future cache inserts.

Each quad-tree node will have an age associated with it (this might simply be the cache's access-count at the time the quad-tree node was created.) By the nature of the hierarchical structure of the quad-tree, a parent's age will always be older than its oldest child's age. This gives us very useful information when scanning through the list of cache nodes.

As we scan cache nodes, traversing into each, we keep track of a 'best fit' quad-tree node and its age. As we continue to visit other cache nodes, we can quickly minimize traversal into these nodes by stopping when we reach a node that is newer than the current 'best fit'. This is the case, because traversing deeper into the tree will only uncover younger quad-tree nodes. Therefore, the more cache nodes visited the lower the probability for having to traverse deep into future cache nodes, increasing our performance with each visit to a cache node. To improve this probability, we choose cache nodes in random order during visitation, which balances the distribution of textures into the overall cache.

TEXTURE MAPPING FROM THE CACHE

Since multiple textures are stored in each cache node (or, texture surface to the 3D APIs) we'll need to modify the texture U/Vs when performing texture mapping, to account for the offset of the physical texture within the cache node.

PROBLEMATIC AREA: TEXTURE WRAPPING

Because we're sharing a single texture surface with multiple textures, there arises a problem when wrapping textures. For those applications requiring wrapping, they have a couple options.

First, limit their wrapped texture to the full size of the cache node. This prevents the possibility of sharing that cache node with any other texture.

Another option is to mark cache nodes as "wrapping" and only insert textures that fit perfectly into those nodes. The efficiency of the cache is reduced by a proportional ratio of wrapped cache nodes to non-wrapped cache nodes.

PROBLEMATIC AREA: MIP MAPPING CACHE NODES

When storing a texture into a quad-tree node, the mip-maps will also require storage. This leads itself into situations where a texture may cross boundaries with other textures in smaller

mip-levels. At the lowest level of a 1x1 mip-map, all textures in a cache node will be blended together in a single pixel. So a minimum mip-level must be chosen.

The number of pixels in the smallest mip-map determines the number of unique textures a cache node can store. Also, the minimum size of a texture is determined by dividing the dimensions of the cache node resolution by the resolution of the smallest mip-map.

This creates a trade-off situation. The more mip-levels, the fewer unique textures that a cache node can store. This is a trade off between mip-mapping effectiveness and cache resolution. An example would be a mip-map count of 5 (smallest mip-map of 16x16). In most applications, this mip-map would be small enough, while allowing a minimum texture size of 16x16 (anything smaller will result in wasted space.)

This completely avoids the problem of unique textures blending together in smaller mip-maps.