

Texture Mapping with Vector Graphics: A Nested Mipmapping Solution

Wei Zhang
Dept. of Computer Science
Prairie View A&M University
Texas 77446

Yonggao Yang
Dept. of Computer Science
Prairie View A&M University
Texas 77446

Song Xing
Dept. of Information Systems
California State University, LA
CA 90032

ABSTRACT

Texture mapping with vector graphics, rather than raster graphics generates better rendering quality. This paper discusses a fully developed approach of texture mapping 3D objects with vector graphics. First, the vector graphics is rendered to generate the corresponding raster graphics, which is temporarily stored in the back framebuffer. Then, using the newly generated raster graphics, a pyramid of mipmap (LOD) images are dynamically generated and maintained. Finally, a “nested dynamic mipmapping” mechanism is applied to pick the right image to achieve the best resolution during rendering. VTexturer, was developed to test and evaluate the proposed approach. The system runs on the regular PCs with the MS Windows operating system, and is capable of using vector graphics in .wmf or .emf format to texture map grid or TIN based 3D objects. The experimental results show that our approach maintains good rendering performance and yields very satisfied results.

KEYWORDS: *Texture mapping, vector graphics, raster graphics, mipmapping, LOD.*

1 Introduction

Texture mapping is a fundamental and must-have graphics primitive in modern computer graphics systems. It allows us to achieve realistic effects with relatively lower cost of computation. However, the traditional raster graphics based texture mapping has its limitation on rendering “resolution” because of its non-scalable

feature. As vector graphics increases its popularity recently, it becomes necessary to investigate applying vector graphics to texture mapping. This paper presents a fully developed and efficient method of using scalable vector graphics to perform texture mapping in 3D graphics. The approach, nested dynamic mipmapping, can be categorized as a LOD solution, where a textured object may have an inconstant number of textures at different LODs.

Vector graphics based texture mapping, or vector texture mapping, has a distinct advantage, that is, the textured 3D objects suffer no texture blurry and deformation. Unfortunately, this topic has not been fully studied so far. Here we briefly summarize the existing work related to this topic. Lance Williams first coined the term “mipmap” in 1983 when he presented his novel idea to solve shimmering and flashing artifacts in regular texture mapping [1]. Later Tanner etc. proposed a dynamic texture representation, named *clipmap*, to perform texture mapping with texture images in huge size [2]. Recently, Ray introduced vector texture maps, where the discontinuity filtering algorithm relies on a manually designed texturing hierarchy [3]. Haddon and Stephenson reported their vector texturing rendering method, which has certain similarity to our approach [4]. However, their approach does not allow the rasterized texture image in arbitrary size. Other related existing works include silhouette maps [5, 6], sharp embedded boundaries [7], and discontinuity meshing related algorithms [8].

The solution presented in this paper extends the traditional raster image based mipmapping to support

vector texture mapping. Our implementation shows satisfactory rendering efficiency and quality.

The remainder of this paper is organized as follows. Section 2 elaborates on our approach, including the nested dynamic mipmapping mechanism and a set of related implementation details. Section 3 describes the experimental system, system performance, and evaluation results. Finally, Section 4 concludes this work and points out the future work directions.

2 Nested Dynamic Mipmapping

In this section, we introduce the *nested dynamic mipmapping* approach to texture map 3D objects with vector graphics in real time. A child level of mipmaps for geometric patches is generated from its parent level of mipmap, and they are nested, in this fashion, recursively to support textures in arbitrary size.

2.1 Overview

With the geometric models and textures specified for 3D objects, texture mapping takes as inputs a series of additional settings to produce the final rendering result. These settings include texture repeating and clamping, texture external and internal color formats, environment mode, and filtering settings. Among them, our nested dynamic mipmapping approach works on the filtering operations that are executed to magnify or minify the texture. Our solution to vector texture mapping is able to completely eliminate the artifact of texture stretch and thus produce the best resolution. In addition to the main approach, a cache mechanism is devised to adjust texture memory allocation. Implemented by aging algorithms, this mechanism ensures that 3D objects, mapped with vector graphics, are rendered in real time. Table 1 shows the comparison of the filtering methods between raster texture mapping and vector texture mapping.

Table 1: Comparison of filtering methods

	Raster textures	Vector textures
Magnifying textures	Linear interpolation	Nested dynamic mipmapping
	Prevents sudden color changes on the mapped surface; Cannot avoid texture stretch artifacts.	Provides the best screen resolution on mapped surface.
	Linear convolution; Mipmapping	Nested dynamic mipmapping with cache mechanism

Minifying textures	Prevents shimmering and flashing effects during real-time rendering.	Inherits the merits of mipmapping; dynamically allocates memory to hold textures for real-time rendering.
--------------------	--	---

2.2 Polygon Tessellation and Division

Polygon tessellation is to divide the polygons of an object surface into smaller ones. There are two main purposes of performing tessellation: (a) produce realistic lighting effects; and (b) provide the base level of texture patches for dynamic texture division during rendering.

The texture mapping process is a set of per-fragment operations. In our approach, when vertex coordinates and texture coordinates are specified, a mesh of model patches is implicitly set up and the texture is divided into corresponding texture patches. LOD modeling decomposes texture patches to more detailed levels in order to represent objects better when they are closer to the viewing point. A texture patch and its corresponding geometric patch are the base of the next level of the texture, as shown in Figure 1.



Figure 1: Texture patch levels

Grid based and triangulated irregular network (TIN) based models are the two most popular methods of representing 3D objects. Each geometric patch is a quadrangle or triangle in a grid based or TIN based geometric model. Since a texture image contains a rectangular array of data, grid based models can directly use the input vertex and texture coordinates at any level of detail. However, when a triangular geometric patch is detached into a more detailed level, the texture patch needs to be recalculated and generated as a rectangular texture, which involves extra memory to hold unused data. To continue the division of texture patches to a further LOD, we simply draw virtual lines by connecting the midpoints of the edges, thus obtain four quadrangles or four triangles.

2.3 Dynamic Mipmaps

Mipmap management is the core of our nested dynamic mipmapping approach. The objective is to provide vector texture in certain sizes to prevent texture stretch, that is, the top level mipmap image is large enough to be mapped onto the model surface without being magnified.

Rasterized images from the original vector graphics are obtained from the back framebuffer, which is more efficient than being loaded from the main memory since texture data is transferred directly from the framebuffer to texture memory.

Here we present an efficient approach to manage mipmaps dynamically and provide a set of criteria for nested mipmaps in our graphics application. In a typical rendering pipeline of graphics systems, such as OpenGL, a series of 4×4 matrices are used to generalize vertex calculations. A homogenous coordinate (x, y, z, w) undergoes a series of calculations to obtain the screen coordinates (x_w, y_w) .

Let M represent the model view matrix, P the projection matrix, and the viewport size be w by h pixels. The projected coordinate (x_p, y_p, z_p, w_p) is:

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ w_p \end{pmatrix} = P \cdot M \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

We then calculate the normalized device coordinate (x_{nd}, y_{nd}) as $x_{nd} = x_p / z_p$, $y_{nd} = y_p / z_p$. In the final step, considering the window device context has its origin at the left bottom corner, we obtain the screen coordinate (x_w, y_w) corresponding to the dimension of the viewport:

$$x_w = (x_{nd} + 1) \left(\frac{w}{2} \right) + x$$

$$y_w = (1 - y_{nd}) \left(\frac{h}{2} \right) + x$$

With the above formulas, we calculate screen coordinates for any vertices in 3D space. Figure 2 illustrates the dynamic mipmaps.

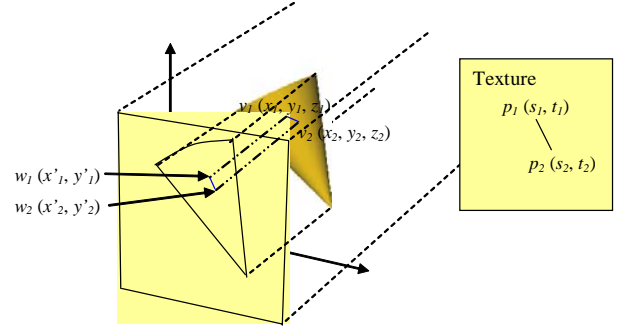


Figure 2: Dynamic mipmaps

Given a geometric patch G and its corresponding texture patch T , we examine first two adjacent space vertices $v_1(x_1, y_1, z_1)$ and $v_2(x_2, y_2, z_2)$ along the border of G . The projected screen coordinates w_1 and w_2 can be computed from space coordinates v_1 and v_2 , respectively. The length of the line segment w_1w_2 in 2D screen pixels is designated d_w .

Texture coordinates for v_1 and v_2 correspond to p_1 and p_2 in texture space. Since vector textures have no fixed dimension, the distance between p_1 and p_2 might vary according to

- The current size of the vector texture
- The ratio of texture width to texture height

Each vector graphics is drawn on a blank pattern, whose size is called the original size of the vector graphics. Another size can be calculated and defined as the initial size of the vector graphics in which each dimension is the power of 2 closest to the original dimension.

Assuming the initial width of the vector graphics is w_I and the initial height h_I , we define a ratio of w_I to h_I as r . In a mipmap, either for the original texture or a texture patch, the base level in the texture mipmap always has the highest resolution, which we define as w_T by h_T . Since the mipmaps are dynamically generated, the mipmap images at the base level also changes. For texture patch T , we call the size of the base level mipmap image the current size of the texture (w_T by h_T). Mipmap images should have a consistent look within a mipmap; the ratio of w_T to h_T should always be equal to r .

The distance d_p between p_1 and p_2 in texture coordinates is calculated. Usually texture coordinates are between 0.0 and 1.0, but we need to unify the s and t coordinates to one direction before calculating d_p , assuming we use the s coordinate to calculate d_p .

For texture coordinates $p_1(s_1, t_1)$ and $p_2(s_2, t_2)$, since s coordinates and t coordinates are proportional to the initial width w_T and the initial height h_T , we obtain d_p as follows:

$$d_p = \sqrt{(s_2 - s_1)^2 + ((t_2 - t_1) \times r)^2}$$

where $r = w_T / h_T$. When texture coordinates (s_1, t_1) and (s_2, t_2) are set to relate texture patch T with geometric patch G , d_p is calculated at that time. Since it remains the same as long as texture coordinates are not reassigned, we keep d_p in a structure for T or G .

Based on d_p , in the next step, we obtain the actual number of texels d_T on the line between p_1 and p_2 . Since we use the s coordinate in calculating d_p , d_T is thus related to w_T :

$$d_T = d_p \times w_T$$

In previous steps, we already know the distance between w_1 and w_2 (d_w) in screen pixels. We compare d_w with d_T to determine whether we need to generate texture mipmap images with higher resolutions to prevent them from being magnified. The two possible situations are:

- (a) $d_w \leq d_T$: The screen display of the texture is smaller than the largest mipmap image. Graphics systems will automatically select proper mipmap levels to texture map the 3D object.
- (b) $d_w > d_T$: The base level mipmap image is not large enough to cover the whole mapped model surface without being stretched. One or more detailed mipmap images need to be generated. The number of those new images is defined as n_M , which can be calculated by

$$n_M = \lceil \log_2(d_w / d_T) \rceil$$

For example, when the viewing point is moving toward the texture mapped object and at a certain point, d_w goes a little bigger than d_T but still smaller than $2d_T$. However, in order to avoid stretching the texture, the mipmap for the texture needs an additional image (since $n_M = 1$) to the top of it. This way, the system keeps track of d_w and d_T for updating the mipmaps for all texture patches.

The above calculation of n_M is only for a single line segment in v_1v_2 and its corresponding line segment w_1w_2 is displayed within the viewport. To consider a geometric patch and its texture patch, we need to know this n_M value for the other sides that encompass the patch.

The other three sides in a grid based model or the other two sides in a TIN based model are calculated. For each side of geometric patches on a model surface, d_w / d_T is only calculated once, since we save all of the ratios to avoid unnecessary calculations. Then we select the largest d_w / d_T and then get the largest n_M .

In practice, we obtain the values of w_1, w_2 from v_1, v_2 . We check if the line segment is within the viewport; if not, this segment should not count as a criterion to compute how many more detailed images need to be added to the object's mipmap. Since we have checked all of the line segments for a geometric patch G , we know if G is within the viewing volume. If G is clipped out of the viewing volume, we don't render it at all.

2.4 Nested Patch Mipmaps

In the preceding subsection, we use the ratio of d_w to d_T to determine whether more detailed mipmap images need to be added to the object's mipmap. For a 3D object, we obtain the largest n_M for all its texture patches. Considering texture size limit, for example, 1024 pixels, if a dimension of a visible texture patch reaches 1024 and it still cannot satisfy the resolution requirement, we check to see if G already has a separate mipmap. If not, we "detach" G 's texture patch T and create a new mipmap from T and bind it to G . Instead of using the original texture mipmap of the object, we select the detached mipmap for future rendering of G . The procedures of building a nested patch mipmap, M_G , are as follows:

- 1) Find the offsets and dimensions in s and t directions of T in the object's original texture.
- 2) Use the largest mipmap image of the object to get the actual size of T , (w_T, h_T) .
- 3) Calculate the initial dimensions of M_p 's top level mipmap image.
- 4) Generate the nested mipmap for G .
- 5) Keep the number of levels n_{PM} in which mipmap images are larger than T .
- 6) Calculate a new set of texture coordinates for G 's vertices using M_G 's texture maps.

G 's mipmap is updated when G is currently using a detached mipmap and the object demands more detailed mipmap images.

2.5 Cache Mechanism

Without a cache mechanism, vector textures might consume all the texture memory, and even the main memory. On the other hand, unnecessary mipmaps waste a huge amount of computation time. We propose a set of aging algorithm to cache texture mipmap among texture memory, main memory, and hard disk.

At any moment, there are three types of mipmap memories that the aging algorithms work on:

- An object's mipmap images which are not currently in use;
- A patch's mipmap images which are not currently in use;
- A patch's mipmap which is no longer in need to prevent texture stretch.

For an object's mipmap, this base mipmap structure should always be kept in order to render geometric patches that do not have detached mipmaps and to provide the basis for dividing texture patches and generating detached mipmaps. We set up a *bottom level* of image size in an object's mipmap, if a mipmap image is above that level and not in need, we can consider assigning ages for that image. If an image exceeds a certain age, it will be released from the object's mipmap. Similar rules apply to a patch's mipmap images. Note the bottom level image size for a patch mipmap is equal to the size of the largest corresponding texture patch. For the third type, if the object is not using the largest texture image or the patch does not even need to use the largest texture patch from the object's texture, the patch's mipmap structure will be aged. If its age goes beyond a certain threshold, the whole mipmap will be destroyed.

A complete execution flow of the aging algorithm first increases ages for the objects that are not visible in the viewport, and then ages are added to those objects' mipmaps and their patch's mipmaps, if applicable. Every time a mipmap or a mipmap image is aged, we check if it is old enough to be released from memory.

During every cycle we look for the largest ratio of d_w to d_T (suppose this ratio is r), the aging algorithm undergoes the following steps:

(1) Case 1: $r \leq 0.5$: In this situation, the next or even a lower level of mipmap image can satisfy the resolution demand. Counting down from the top of the mipmap, the

actual number of levels n_{AL} that needs adding ages is calculated as:

$$n_{AL} = \min(\lceil -\log r \rceil, n_L)$$

The mipmaps of object's patches also need aging if the patch has a nested mipmap and either it is not currently in sight or its largest ratio of d_w to d_T is smaller than 1.

(2) Case 2: $0.5 \leq r \leq 1$: We reset the ages of the object's mipmap images to 0 because all of these images are being used during rendering in order to prevent texture stretch.

(3) Case 3: $r > 1$: First, we use the same method as stated in Case 1 to check and add age for every patch's mipmap. For each patch, we calculate n_M as described using the formula $n_M = \lceil \log_2(d_w / d_T) \rceil$ to update the patch's mipmap. Again, n_M is the number of mipmap images whose resolutions are higher than the object's texture patch. If n_M is less than n_{PM} , the number of levels already added, we calculate n_{AL} based on n_L , n_M , and n_{PM} by $n_{AL} = \min(n_{PM} - n_M, n_L)$. Then we apply the aging algorithm to the n_{AL} images, that is, the number of mipmap images from the top of the patch's mipmap.

With the aging algorithm, we keep track of the amount of memory that has been allocated to hold mipmaps and mipmap images. If this value reaches the upper limit, for example, 128MB, mipmaps or mipmap images with the largest age are released.

Besides the main execution flow of the cache mechanism, we may consider some other factors to design the aging algorithm:

- If a mipmap or a mipmap image has not been used for a certain period of time, its age increases.
- We set up a limit for the amount of memory that can be allocated to hold mipmaps. If mipmaps grow beyond that limit, the least recently used mipmap image is released.

3 An Experimental System

An experimental system, *VTexturer*, has been developed on Windows platform, using OpenGL and VC++.Net. Besides the nested dynamic mipmapping and aging algorithms, *VTexturer* has the following features:

- Supports various vector graphics formats: wmf, emf, and raster graphics formats: bmp, gif, jpg.

- Supports both grid-based (quadrangular patch) and TIN-based (triangular patch) object models.
- Allows creating, loading, editing, and saving 3D object models in VTexturer scene file format.
- Allows navigating freely in the virtual scene.
- Allows tracking internal data structures and keeping record of their dynamic changes.

We conducted the performance evaluation on laptop with a Pentium4 2.2GHz processor, ATI Radeon 7500 graphics card, and 256MB main memory. The testing result may vary on different hardware platform and software settings. We used a 34KB wmf vector graphics to texture-map 100 to 1089 quadrangles. We tested them from a distance that we can view all objects clearly.

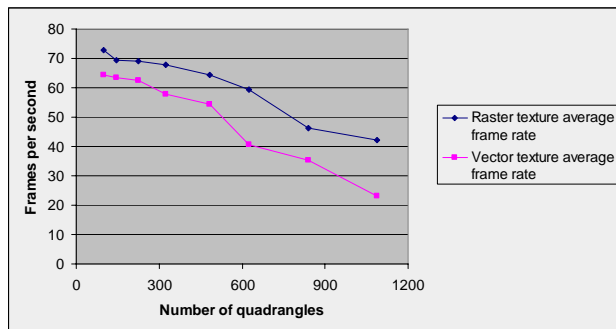
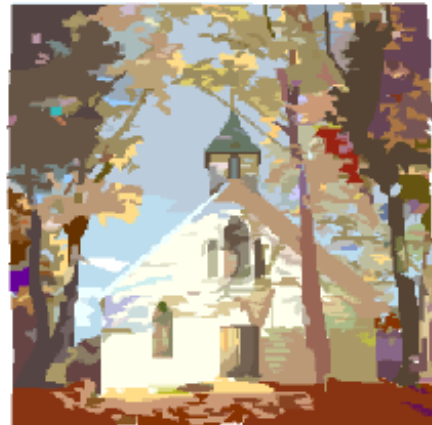


Figure 3: Rendering performance comparison

Figure 3 shows the performance comparison between raster texture mapping and vector texture mapping. The raster image and the vector graphics are in the same size and they look the same. The test result shows that the vector texture mapping is slightly slower than the traditional raster texture mapping. The frame rate of the vector texture mapping remains above 20 fps when more than 1000 quadrangles of object models are visible.

Figure 4 shows the example discussed in the preceding section. The object surfaces, texture-mapped with the same graphics but one in raster format and the other in vector format, look the same as Figure 4(a) if viewed from a far distance. When the viewing point is moved closer to the object, the surface mapped with the raster texture appears blurred, as shown in Figure 4(b), while the surface mapped with the vector texture always has the best screen resolution, as shown in Figure 4(c).



(a) Original vector graphics image



(b) Raster graphics based texture mapping



(c) Vector graphics based texture mapping

Figure 4: Texture mapping comparison

Figure 5 illustrates a TIN object model texture-mapped with a vector graphics. There are two strings of text on the vector texture, which are not clearly visible due to the far distance. When the viewing point is moved closer to the object model, the text is getting clearer.

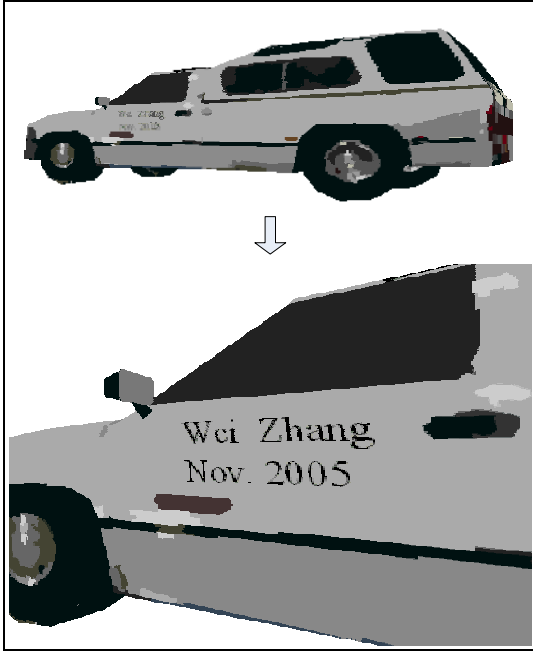


Figure 5: Texture map a TIN model with vector graphics

In the last example, we set a two-level tessellation for an object surface that has up to 100^2 quadrangular patches. If the maximum texture size for the OpenGL system is selected for the largest size of a mipmap image, the entire texture map will be in the size of 200k by 200k pixels in the rasterized format.

4 Conclusion

This paper presents a novel mipmap based solution to extend the traditional raster graphics image based texture mapping to vector texture mapping. The nested dynamic mipmapping approach, composed of two level-of-detail phases, is devised to realize vector graphics based texture mapping in real time. Vector textures are first rasterized in the back framebuffer and tessellated or further divided when building nested mipmaps. Besides the texture patches visible in the viewport, the nested mipmap structure also keeps texture data of lower or higher levels of mipmap images and, if applicable, texture data of neighboring detached texture patches, which are generated during mipmap updating in every rendering cycle. A cache mechanism, implemented with a set of aging algorithms, accelerates the real-time rendering and also limits the amount of texture memory and main memory allocated for the dynamic mipmaps. The experimental system, *VTexturer*,

shows satisfactory results of real-time rendering and memory usage. When objects are textured with vector graphics and rendered on the screen, they exhibit high rendering quality as they are mapped with dynamically generated raster textures that are large enough to meet the screen resolution demand.

The nested dynamic mipmapping solution is currently implemented at software level. The mipmap updating takes most of computation time. It involves a large amount of computation, such as matrix transformation, rasterized texture generation, and mipmap image settings. All these tasks can be implemented and accelerated at hardware level, and thus improving the general performance.

References

- [1] L. Williams. 1983. "Pyramidal Parametrics". *ACM SIGGRAPH '83*. vol.17, Issue 3, pp.1– 11.
- [2] C. Tanner, C. Migdal, and M. Jones. 1998. "The Clipmap: a Virtual Mipmap". *Proc of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. pp.151 – 158.
- [3] N. Ray, X. Cavin, and Bruno Levy. 2005. "Vector Texture Maps on the GPU". www.loria.fr/~levy/publications/papers/2005/VTM/vtm.pdf
- [4] J. Haddon and I. Stephenson. 2001. "Implementing Vector-based Texturing in RenderMan". DCT Systems. www.dctsystems.co.uk/Text/haddon.pdf
- [5] P. Sen, M. Cammarano, and P. Hanrahan. 2003. "Shadow Silhouette Maps". *ACM Transactions on Graphics*. vol.22, Issue 3. pp.521 – 526.
- [6] P. Sen. 2004. "Silhouette Maps for Improved Texture Magnification". *EUROGRAPHICS Workshop on Graphics Hardware*. pp.65 - 73.
- [7] J. Tumblin and P. Choudury. 2004. "Bixels: Picture Samples with Sharp Embedded Boundaries". *Proc of the Eurographics Symposium on Rendering*.
- [8] P. Heckbert. 1992. "Discontinuity Meshing for Radiosity". *Eurographics Workshop on Rendering*. pp. 203–226.