Volume xx (*200y*), *Number z*, *pp. 1–10*

3D Line Textures and the Visualization of Confidence in Architecture UUCS-07-005

Kristin Potter, Amy Gooch†, Bruce Gooch†, Peter Willemsen‡, Joe Kniss*, Richard Riesenfeld and Peter Shirley

University of Utah † University of Victoria ‡ University of Minnesota Duluth *University of New Mexico

Abstract

This work introduces a technique for interactive walkthroughs of non-photorealistically rendered (NPR) scenes using 3D line primitives to define architectural features of the model, as well as indicate textural qualities. Line primitives are not typically used in this manner in favor of texture mapping techniques which can encapsulate a great deal of information in a single texture map, and take advantage of GPU optimizations for accelerated rendering. However, texture mapped images may not maintain the visual quality or aesthetic appeal that is possible when using 3D lines to simulate NPR scenes such as hand-drawn illustrations or architectural renderings. In addition, line textures can be modified interactively, for instance changing the sketchy quality of the lines, and can be exported as vectors to allow the automatic generation of illustrations and further modification in vector-based graphics programs. The technique introduced here extracts feature edges from a model, and using these edges, generates a reduced set of line textures which indicate material properties while maintaining interactive frame rates. A clipping algorithm is presented to enable 3D lines to reside only in the interior of the 3D model without exposing the underlying triangulated mesh. The resulting system produces interactive illustrations with high visual quality that are free from animation artifacts.

1. Introduction

Presentation graphics are communicative illustrations often employed by architects and other design professionals to express not only the features of a design or model, but also informative aspects such as material property, confidence or completion levels, and other important characteristics. These illustrations typically avoid the factual connotations associated with realistic imagery and instead use rendering styles that align better with the conceptual ideas being expressed. Images that resemble photographs are often interpreted as complete and unchangeable, while loose, sketchy illustrations can express malleable characteristics of a model or design. The degree of looseness of rendered lines is often associated with the variation of characteristics within a model. This line "sketchiness" can vary within a single image, visually revealing the change of a model attribute, and quickly expressing a great amount of information in a concise and compact way. An example of such an illustration can be seen in the archaeological reconstruction shown in Figure 1 which uses our technique to automatically place feature and texture lines on the model, and allows a user to modify the line characteristics while walking through the scene at interactive frame rates. In this illustration, the sketchiness of the feature edges and material property lines is modified based on the confidence of the specific areas of the reconstruction. Thus, the base of the Mayan Temple is rendered with straight, clean lines because this area of the model has the highest confidence level, while the hut at the top has a very low level of confidence and is rendered with loose, sketchy lines, and does not have any textural detail. The methods presented in this paper use 3D line primitives rather than traditional texture mapping in order to maintain high visual line quality, allow real-time modification of line characteristics and automatically generate vector illustrations from within an interactive walkthrough of the scene.

When creating an interactive line drawing system, the following important characteristics must be considered:

1. high visual quality of individual frames;
2. animation free of dynamic artifacts, such as popping;
3. high frame rate; and

4. ability to directly create manipulatable 2D illustrations.

The first two items suggest using 3D line primitives, as they can be anti-aliased in screen space, thus producing high visual quality. In addition, line primitives do not need level-of-detail management to maintain constant width or brightness in screen space. However, it is natural to think that interior lines should be rendered using texture mapping for efficiency. Indeed, texture mapping has been used effectively to accomplish interior line rendering by others [FMS01], who used careful generation of MIP mapped line textures to avoid dynamic artifacts. Unfortunately, this technique makes the line textures static, so line sketchiness cannot be varied at runtime. Also, texture mapping does not allow for the direct creation of 2D illustrations. Our method directly generates postscript illustrations in which each texture and feature line can be manipulated after creation.

The question remains whether 3D line primitives can be used while maintaining an interactive frame rate. Although lines are not used in most graphics programs, they are highly optimized by hardware makers because of the CAD market. Using lines directly has several advantages over texture mapping:

- line primitives can be anti-aliased without a multi-pass algorithm
- line primitives can have their sketchiness varied at runtime by perturbing vertices in a hardware vertex program
- line primitives preserve their width in screen space even for extreme close-ups.

The last item could be viewed as an advantage or a disadvantage depending on one's priorities; having constant width lines in screen-space makes for a clean drawing reminiscent of the type drawn by human draftsmen, but exchanges line-width depth cues for line-density depth cues, which can be visually distracting.

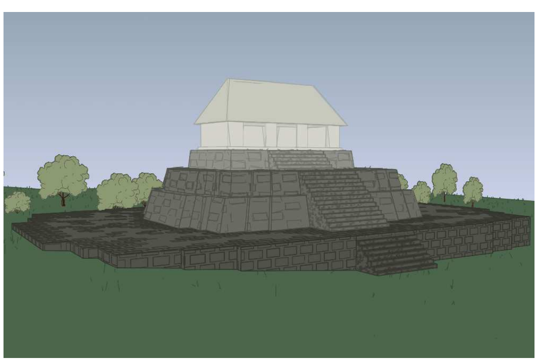

Figure 1: *An example illustration from an interactive session of an archaeological reconstruction with the sketchiness of the line primitives varied according to the level of confidence in the data. The base of the structure is rendered using clean lines, the midsection is rendered with a sketchy line texture, and the "top hut" is a matter of conjecture and is rendered without texture and very sketchy feature lines.*

This paper's main contributions are creating a system that allows for the automatic placement of texture across a model, the interactive manipulation of viewpoint in a 3D scene, and the creation of 2D vector illustrations. The success of the system is demonstrating that high frame rates can be achieved using line primitives in scenes of realistic complexity. An algorithm is provided to automatically place line textures on objects in order to perform material property "indication," i.e., a small number of texture elements indicates the material properties of the entire surface. Finally, we show that an interactive system using 3D lines is relatively simple to design and build, making line primitives a practical alternative to texture mapping with respect to software engineering as well as efficiency issues.

2. Background

The style of illustration used to create an image has a profound effect on the interpretation of that image [SPR*94]. Photographic images, like the ones traditionally generated by computers, give the sense that the scene is complete and unchangeable, not inviting discussions about design, or indicating the actual confidence level of the underlying model. Conversely, hand-drawn illustrations have a transient quality, the viewer is more comfortable talking about modifications to the scene [SSRL96], and is often more actively engaged with the image. Because of the difference in the effect of rendering style, determining which style to use is very important. It is also possible to use multiple styles within a single image, allowing for differing interpretations across the image [SMI99].

In addition to decisions on rendering style, specific features of a scene must be identified in order to create a communicative illustration. Feature lines such as silhouettes, creases, boundary and contour edges aid in the understanding of geometric shape and should be accented [ST90, SFWS03]. Likewise, how these lines are accented, as well as how other lines in the illustration appear is critical. For instance, in line drawings, the type of line, such as dashed, dotted, thick or thin, can express direction, distance and location, and end conditions can convey the relation of the line with respect to other lines and surfaces in the scene [DC90]. Such conventions are standard in hand-drawn illustrations, and can be adopted for use in computer generated imagery.

Translating artistic techniques from the human hand to a computer process requires that inherent characteristics of the traditional media be explicity defined. There has been much work in simulating charcoal, ink, watercolor, and other artistic mediums on the computer [GG01, SS02]. These methods simulate the variations in thickness, waviness, weight, and direction of the marks left by the media, as well as their interaction with paper or canvas. Typically, the strokes left by artistic techniques are texture mapped onto the model. Alternatively, actual geometry can be used in the form of graftals, a method which randomly places particles across a surface, from which artistic details can "grow" [KMN*99, KGC00, MMK*00]. While the physical simulation of the physical properties of artistic media is challenging, a harder problem is simulating the artistic hand.

It is very difficult to completely automate where an artist places strokes, and most systems rely on human interaction to aid in the generation of illustrations. For example, in work by Salisbury et al. [SABS94, SWHS97], the user controls where collections of strokes are placed on a model by "painting" on areas of interest. The Piranesi system [Sch96, RS95] allows users to paint artistic effects onto CAD models, creating images that are closer to the images handmade by architectural designers. The SKETCH system [ZHH96] uses gestural input as a method for modeling, simulating the artist sketching out a preliminary design on paper. Dollner and Walther [DW03] create a system that uses not only NPR techniques, but also cartographic and cognitive principles to render 3D city models with enhanced feature edges, two or three-tone shading, texture mapped details and simulated shadows. These types of systems aid in creating images that maintaining a human quality and allow for computer specific additions such as walkthroughs and interactive revisions.

Many interactive NPR algorithms suffer from a lack of frame-to-frame coherence in which the texture or geometry used to express the artistic detail pops in or out of the scene, or the strokes do not match up from one frame to another. This problem arises from the fact that the artistic techniques being simulated are meant to be seen as a single instance, rather than in repetition. Solutions to this problem involve fading detail into the background [MMK*00], or variations on texture and MIP mapping techniques. Texture mapping hardware can be used to maintain tone and detail via hatch and ink maps [FMS01], as well as to add fine tone control and reduce aliasing artifacts [WPFH02]. Maintaining coherence in an image based approach can be done by using art maps which maintain the width of NPR strokes in the scene, and rip maps eliminate artifacts that occur at oblique viewing angles [KLK*00].

The approach presented in this paper is novel in its use of three dimensional line primitives as an alternative to texture mapping. Line primitives have an advantage in that they maintain constant screen space throughout an interactive session, and require no special algorithms to preserve frame-to-frame coherence. Feature edges and texture are expressed through the placement of the 3D lines which is done automatically. Similarly to the approach by [WS94], texture is placed densely so as to accent feature edges, and sparsely across the rest of the model to "indicate" the texture. To create a hand-drawn quality and describe confidence levels in the underlying model, the endpoints of the lines can be interactively perturbed using graphics hardware, a feature that is difficult to achieve using texture mapping. Another benefit of using 3D line primitives to express model boundaries, features and texture is the ability to output 2D vector illustrations, allowing for scale, resolution, and display independent reproduction [SALS96].

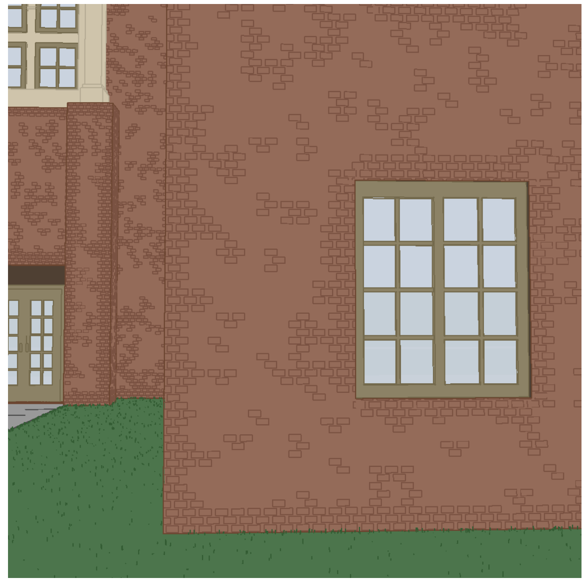

Figure 2: *Texture is increased around crease and boundary edges to enhance the features of the model.*

3. Algorithm

The algorithm presented here to create renderings with texture indicated by 3D lines consists of preprocessing the model to find important edges, determining the placement of and laying the texture, and finally clipping the texture to the model. New techniques include using 3D lines as an alternative to texture mapping, automating the indication of texture such that texture is minimized and placed sparsely across the model as well as along important edges, and creating sketchy lines through vertex programs. The system uses simple algorithms and currently supports a variety of textures such as bricks, stone work, shingles, thatch, stucco, and siding, while other materials could easily be supported [Yes79, Miy90].

3.1. Feature Edge Detection

An important cue to understanding the shape of an architectural model is emphasizing feature edges [ST90]. These edges include outlines separating the model from space, creases distinguishing interior features, and boundaries between materials. To find feature edges in the triangulated mesh used to represent the model, the model is first divided

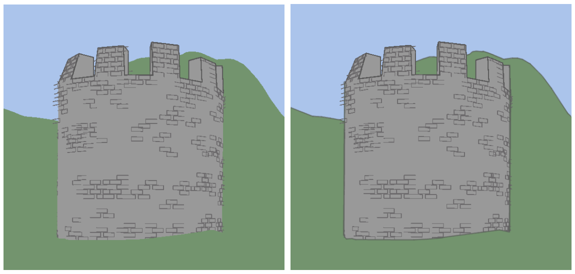

Figure 3: *A model with silhouette feature edges. Since the texturing algorithm is a pre-process, texture is neither populated nor clipped against silhouette edges (left). Adding silhouette edges aides in the understanding of the model, without needing to modify the texture algorithm.*

into material groups such as bricks or grass. The test for outlines and material boundaries is then simplified into a brute force search for edges contained in only a single polygon. The search for crease edges identifies edges that adjoin two polygons whose surface normals have a dihedral angle greater than θ, for some threshold value of θ. The feature edges are then explicitly drawn with a 3D line and texture is placed at a higher density close to these important areas as illustrated in Figure 2.

Because the feature edge and texture generation algorithms are designed as pre-processes, view-dependent silhouette edges (i.e., edges between front facing and back facing polygons) are not accented with texture. This raises problems for models, such as the one shown in Figure 3 that have border edges that can only be defined as silhouettes. In such models, texture is not populated or clipped about silhouette feature edges, which does not emphasize the feature edge and texture can possibly lie outside of the model. Moving the texture generation process to runtime would allow silhouette lines to be treated as stationary feature edges, however, the texture lines would need to be recalculated each time the viewpoint changed, leading to disturbing artifacts between frames. While there is no obvious solution to the texturing problem, 3D silhouette edges can be added as a hardware process at runtime, as shown in Figure 3, right, allowing such models to be displayed in our system.

3.2. Texture Generation

Line texture synthesis across interior surfaces is carried out according to a heuristic that thresholds Perlin solid noise [Per85] to place clusters of texture. An atomic texture element (e.g., a single brick or blade of grass) is placed on the triangle if the function:

$$\frac{1+3\times\ N(kx,ky,kz)}{2},$$

is above a threshold, (where N is the Perlin function). This function gives a uniformly random distribution of texture without excessive accumulations or concentrations. The threshold can be changed to allow more or less texture on any portion of a model.

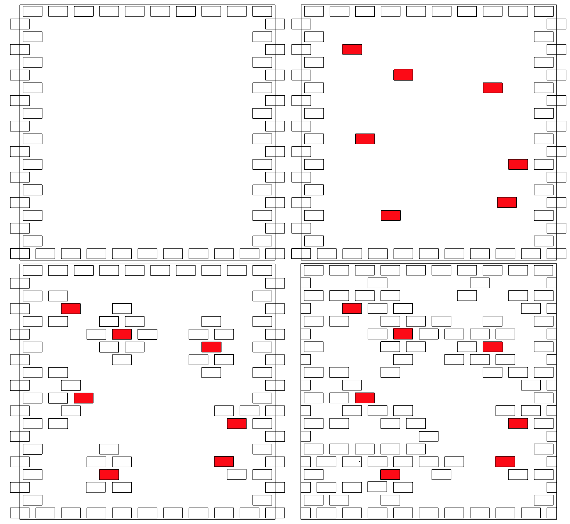

Figure 4: *The texture generation process. Top left: texture is placed along feature edges. Top right: atomic texture elements are placed based on Perlin noise. Bottom left: texture is clustered about the atomic elements. Bottom right: a complete textured cube with clipping (note: due to random selection of texture library elements, actual texture element placement varies.)*

The algorithm for texture placement iterates through the texture space of the model, calculating the noise function at each location where texture would be placed for full coverage of texture on the model. If the noise function returns a value indicating that the location should be textured, the texture lines corresponding to an atomic texture element are transformed into model space. Feature edges are embedded into the noise function such that the return value always indicates that these areas should be textured. The top two images in Figure 4 demonstrate this initial texture placement. On the left, texture is placed along the feature edges; on the right atomic texture elements (in red) are placed throughout the interior of the model.

Once an atomic texture element is placed, a texture cluster is populated around it. A texture cluster is a pleasing group of texture elements. The aesthetic quality of these groupings is critical to creating a good image. While automatic generation is possible, we have found that the criteria for what makes a good texture cluster are not obvious. Thus, we created a library of human generated clusters that are reused. The bottom images in Figure 4 show the final stages of texture placement. On the left, texture is clustered around atomic elements, and on the right is the final, clipped result, with enough texture added to pleasingly cover the surface.

A side-effect of the texture library is the repetition of texture elements, especially in very regular textures such as brick. This occurs when the texture clusters generated by the library place texture elements on top of already placed texture. The drawback is that this increases the line count, and may effect performance, however this does not occur on random textures such as the grass, which make up the bulk of texture lines. While these repetitive lines can easily be removed through a pre-process sort, we have chosen to keep the duplicate lines, since they are perturbed independently when the sketchiness is increased. This results in a quality that more closely resembles hand-drawn, since an artist will often draw repetitive lines when sketching.

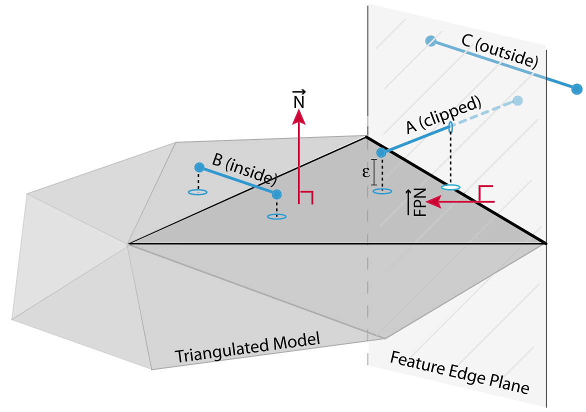

Figure 5: *The Clipping Algorithm. The texture lines A, B, and C are clipped against the current triangle (outlined in black) with normal N. C is not clipped because it does not project onto the triangle. B projects onto the triangle within ε distance, but is not clipped because it does not intersect with the feature edge plane (FP). A projects onto the triangle, and intersects FP; the line is split by the intersection point into two segments. The line segment that has a positive dot product with the feature plane normal (FPN) is kept as the new texture line.*

4. Texture Clipping

Following the placement of the texture lines on the model, all lines are clipped against the crease and boundary edges of the model. Clipping against the feature edges maintains the unity of the model, while clipping against single triangles would reveal the underlying triangulation. Thus, clipping is performed after all texture has been generated. However, clipping 3D lines against the 3D feature edges is not a trivial task.

The algorithm to clip texture along feature edges can be broken into two steps. First, all texture lines that are completely outside of the model are removed. Second, the remaining texture lines are clipped against the feature edges.

4.1. Outside Line Removal

The main goal of the texturing algorithm is to maintain visually appealing texture across the model. To hide the triangulation of the underlying representation, texture lines extend across the boundaries of the generating triangle and are often drawn completely outside of the model. These exterior lines will not be removed by the clipping algorithm since they do not intersect any feature edges. Instead, every texture line must be tested against every triangle in the model to determine if either endpoint is contained by the model.

The first step in removing outside edges is to find the distance, d, from the texture line, l, to the current triangle with a vertex tx, and normal N:

$$d = min(\frac{(l_x - t_x)\cdot N}{||N||}, \frac{(l_y - t_x)\cdot N}{||N||}). \quad (1)$$

If the distance is close to zero, the endpoints of the texture line are projected onto the plane of the triangle, by first creating two directed lines from the endpoints of the texture line pointing in the direction of the triangle normal. Each of these directed lines, w, are intersected with the plane of the triangle described by the triangle normal, N, and a point in the triangle, p:

$$u = \frac{(p - w_x)\cdot N}{(w_y - w_x)\cdot N}.$$

The u value is the parametric location of the intersection point on the line, w_i. The actual intersection point can be found by solving the line equation using the u value:

$$w_i = w_x + u(w_y - w_x).$$

If either of the intersection points are found to lie within the triangle, the texture line is considered to be contained within the model and the test stops. Otherwise, the test continues with the rest of the triangles of the model, and if the texture line does not lie within any triangle, it is discarded.

4.2. 3D Clipping

Once all of the exterior texture lines are removed, the algorithm tests each texture line against each triangle that contains a crease edge. As shown by Figure 5, the texture line (a,b) is clipped against the crease line (y,z) that is contained by the triangle (x,y,z). The main steps of the algorithm are as follows: 1) find the distance from the texture line to the triangle (similar to the test above), 2) for texture reasonably close to the current triangle, intersect the texture line with the "feature edge plane", 3) test that this intersection point lies between the endpoints of the feature edge, and finally, 4) keep the segment of the texture line that lies inside the triangle.

4.2.1. Distance Test

The first step in 3D clipping is to determine the distance from the texture line to the current triangle using Equation 1. Texture lines will only be clipped by feature edges that are spatially close (i.e. the distance function is within some threshold). This eliminates "ghosting" effects that occur when texture lines are clipped by distant feature edges.

4.2.2. Find Feature Edge Plane

The next step in the clipping algorithm is to find the "feature edge plane" (FP). This is the plane that contains the feature edge, and is perpendicular to the triangle. It is defined by taking the cross product of the feature edge, f, and the triangle normal, N, (i.e. $FP = f \times N$). The normal to the feature edge plane should point into the triangle, making the dot product of the feature plane normal with an edge of the triangle positive. In the case in which it is not, the feature edge plane normal is flipped.

4.2.3. Intersect Feature Edge Plane

The next step is to intersect the texture line with the feature edge plane. This intersection point will be the new endpoint of the texture line, however we must test that this intersection line actually lies between the endpoints of the feature edge. To do this, we project the intersection point onto the triangle plane by intersecting the line formed using this intersection point and the triangle normal, with the triangle plane. This gives a new intersection point that lies along the intersection line of the crease plane and triangle plane. To determine if the intersection point lies inside the crease edge, two vectors are formed using each crease line endpoint, and the intersection point that lies on the triangle plane. The dot product of these two vectors is taken, and if the dot product is zero, then we know the intersection point is within the crease line.

Once it has been found that the texture line should be clipped by the crease plane, which segment of the texture line to discard must be determined. This is done by forming two new vectors, using the endpoints of the texture line and the new crease plane intersection point. These vectors will point in opposite directions. The dot product of the crease plane normal and the two new vectors is computed, with the vector resulting in a positive dot product kept as the new texture line, and the rest of the line thrown out.

4.3. Vertex Perturbation

Once all of the lines of a texture have been placed and clipped, it is possible to adjust the sketchiness of the lines. To achieve "sketchiness" the endpoints of the texture and feature edge lines are randomly perturbed. A sketchy quality of the lines adds to the hand-drawn look of the imagery, and can be modified independently in different areas of the model, allowing each area to have a unique sketchy quality and maintain the unity of the scene. It is hard to determine the amount of sketchiness desired for the model, so allowing the user to modify the sketch quality parameter is desirable. Examples of varying levels of sketchiness are shown in Figure 6.

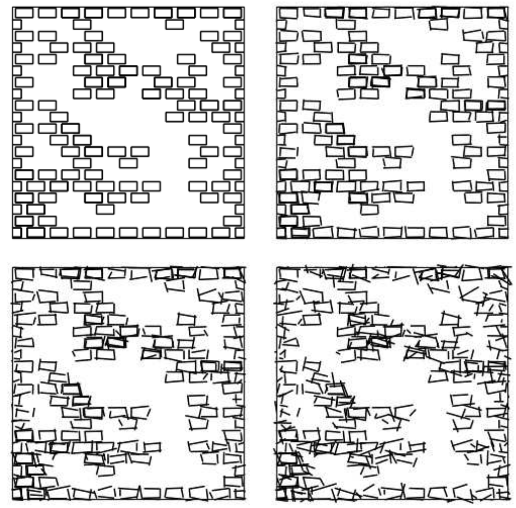

Figure 6: *Four levels of sketchiness.*

Using a hardware vertex program coded with the OpenGL Shading Language, the modification of the original line texture to sketchy lines can be done interactively. The goal of our vertex program is to maintain the basic line texture structure while adding a slight random offset to the original vertices. For each line vertex, the vertex program generates a perturbed line vertex coordinate by adding a perturbation vector to the original vertex. This 3D vector is randomly generated and stored in the multi-texture coordinate vertex attribute during the creation of the line texture vertex buffers. We also set the normal vector for each vertex to be the normal of the face on which the line texture is being rendered. These are the only vertex attributes that need to be set for the vertex program to execute.

Once running, the vertex program projects the random perturbation vector onto the plane determined by the normal, and then uses this projected vector to slightly change the location of the original vertex. The projection computation forces the modified vertices to remain in the plane of the polygon in which the line texture resides. To remove z-fighting, we also add a small offset in the direction of the face normal to lift the vertex off the surface slightly. We do not perform additional clipping at this stage to stop sketchy lines from extruding outside of the polygon face. The OpenGL Shading Language code to modify the vertices of our line textures can be found in the appendix. The fragment program used in our implementation simply sets the input color as the output fragment color.

4.4. Creating Two Dimensional Illustrations

Representing feature edges and texture as 3D line primitives allows the direct creation of 2D vector illustrations. This is especially useful for fine manipulation of the illustration after creation. The creation of a two dimensional illustration is straightforward. Once the user chooses a suitable viewpoint in the 3D scene, GL2PS [Geu] is used to convert the line primitives to vectors.

5. Discussion

Using only the indication of texture in which the material properties are hinted at rather than fully illustrated is intriguing for the viewer. Large amounts of line textures are distracting, and the imagination of the viewer is engaged to fill in the texture where it is omitted. In addition, the number of lines used to suggest texture is reduced, which helps maintain performance. Implementing indication automatically is a difficult problem because it is hard for artists to describe the process of deciding where to place texture. In previous implementations, systems have relied on the artist to input areas of the model that should be enhanced by texture [WS94]. Looking at the images created by these systems, it seems that the feature edges are common areas to receive more texture. Although feature edges may not be the only such areas to receive texture indication, this method enhances feature edges because they are so often enhanced by artists. It has also been shown that the enhancement of these edges aids in the understanding of the model. Additionally, material boundaries are typically enhanced by more texture indication, another phenomenon captured by this implementation. The remaining areas of the model receive sparse texture to reduce clutter, increase efficiency and maintain a clean look [Lin97].

An interesting feature of our texture placement algorithm is maintaining the density of the texture with distance. This approach allows the tone of the image to vary with depth so objects farther away will have a higher texture density and thus a darker tone, which can be seen in Figure 7, left. The tone of the image can be thought of as the ratio of black ink to white paper. Allowing the tone to vary is a method often adopted by artists to create the illusion of depth in the image. Traditionally computer graphics techniques decimate texture density with distance to maintain tone across an image. Thus, the density of texture on an object in the foreground would be at a level equal to the density of texture on a distant object. This is done to preserve the visual appeal of the image because the texture in the background can quickly become too dense, creating a very dark tone that is distracting and unappealing. However, removing texture can be perceptually confusing and lead to misperceived distance, results that conflict with the goals of this system. Thus, the implementation presented herein does not eliminate texture in the distance. A benefit of using indication is that in the distance, the texture density is still lower than if the entire surface was textured. This keeps the tone at an appealing level throughout the scene while preserving the property that the tone is darker in the distance. A possible drawback to this approach is that the size of the environments is limited. The assumption that partial textures will maintain appeal at far distances may not hold when the model becomes very large and extend a long way into the distance. The lines used to convey the texture are colored lines which are not as distracting as black lines when grouped tightly, and may not be as unappealing in the background, as shown in Figure 7, right. Artists, instead of removing texture in the distance, will use a lighter line when drawing texture in the background. Thus, a possible solution for a texture density that is too high in the distance is to fade with distance the lines that make up the texture.

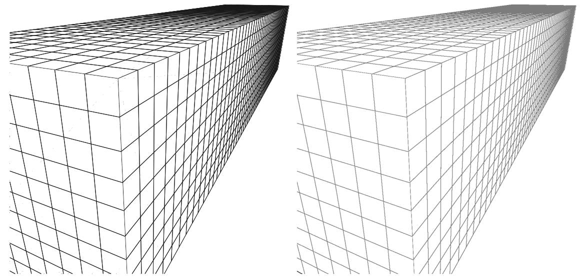

Figure 7: *An example of a fully textured wall whose tone gets too dark in the distance. While this implementation uses indication to lessen the amount of texture, and thus reduce the problem of dark tone in the distance, we also use colored lines which eases the problem of tone in the distance.*

Another addition to the system to aid in aesthetic appeal is entourage. Entourage refers to the use of familiar objects of known size in the scene and is used to create the effect of scale [Bur95]. The system uses hand-generated people and trees, which are placed in non-distracting areas of the scene such as near a corner of a building or in the background. To maintain the overall look, the trees resemble the pen and ink trees of Deussen [DS00].

6. Implementation

Scene	Polygons	Line Count	frames/sec
Mayan Temple	1,259	58,276	149.1
Castle	9,734	27,910	102.4
Olympic Village	20,467	1,054,559	11.0

Table 1: *Polygon count, line count, and frame display times averaged across 125 frames for interactive walkthroughs using our system at an image resolution of 1000x1000.*

Our system runs on a dual-core, 3.0GHz Intel Pentium D machine with 2GB RAM and an NVIDIA GeForce 7800

GTX graphics co-processor. The implementation is fairly straightforward, requiring approximately 3000 lines of code. The code is written in C++ using OpenGL libraries. The system's only drawing primitives are 3D constant-colored lines and 3D constant-colored polygons. Table 1 gives the polygon and line count as well as frame rates for a 1000x1000 pixel image. Figure 8 shows screen shots of an interactive session of our system using a model of the Salt Lake City Olympic Village.

7. Future Research

We have shown that interactive frame rates can be achieved using line primitives on scenes of realistic complexity. The approach presented is a nice alternative to texture mapping in that it is easy to implement, automates the indication of texture, allows for the runtime manipulation of the sketchy quality of the lines, and can be used as vector graphics to automatically produce 2D illustrations.

There are several further directions of research stemming from our system. First, it may be possible to use fewer line elements to indicate the material properties of a surface. Such an investigation is likely to involve perceptual psychologists. Alternative methods for texture placement could be investigated; perceptually or artistically based noise functions could be used instead of Perlin noise. The entourage elements are static 2D billboards. Depending on the application, 3D models or animated billboards could be used, or moved about in the scene. Finally, we have avoided the LOD management issue by limiting ourselves to medium-scale environments. For very large sized environments, some LOD management system may be needed.

8. Conclusion

When creating renderings that simulate hand-drawn illustrations, line quality and aesthetic appeal are important. These characteristics are difficult to achieve using texture mapping techniques. 3D line primitives, however, maintain high visual quality independent of viewpoint, suggest the aesthetic nature of hand-drawn illustration, are free of dynamic artifacts and can be manipulated in an interactive setting.

9. Acknowledgments

This work was supported in part by NSF grant (03-12479). All opinions, ndings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reect the views of the sponsoring agencies.

References

[Bur95] BURDEN E. E.: *Entourage: A Tracing File for Architects and Interior Design*. McGraw-Hill Professional Publishing, 1995. 7

[DC90] DOOLEY D., COHEN M. F.: Automatic illustration of 3d geometric models: Lines. In *Proceedings of the Symposium on Interactive 3D Graphics* (1990), pp. 77–82. 2

[DS00] DEUSSEN O., STROTHOTTE T.: Computer-generated pen-and-ink illustration of trees. In *Proceedings of ACM SIGGRAPH 2000* (2000), pp. 13–18. 7

[DW03] DOLLNER J., WALTHER M.: Real-time expressive rendering of city models. In *Proceedings of Information Visualization 2003* (2003), pp. 245–250. 3

[FMS01] FREUDENBERG B., MASUCH M., STROTHOTTE T.: Walk-through illustrations: Frame-coherent pen-and-ink style in a game engine. *Computer Graphics Forum: Proceedings of Eurographics 2001 20*, 3 (2001), 184–191. 2, 3

[Geu] GEUZAINE C.: Gl2ps: an opengl to postscript printing library. http://www.geuz.org/gl2ps/. 7

[GG01] GOOCH B., GOOCH A.: *Non-Photorealistic Rendering*. A.K. Peters, 2001. 2

[KGC00] KAPLAN M., GOOCH B., COHEN E.: Interactive artistic rendering. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering* (2000), pp. 67–74. 3

[KLK*00] KLEIN A. W., LI W. W., KAZHDAN M. M., CORREA W. T., FINKELSTEIN A., FUNKHOUSER T. A.: Non-photorealistic virtual environments. In *Proceedings ACM SIGGRAPH 2000* (2000), pp. 527–534. 3

[KMN*99] KOWALSKI M. A., MARKOSIAN L., NORTHRUP J., BOURDEVY L., BARZEL R., HOLDEN L. S., HUGHES J. F.: Art-based rendering of fur, grass, and trees. In *Proceedings of ACM SIGGRAPH 99* (1999), pp. 433–438. 3

[Lin97] LIN M. W.: *Drawing and Designing With Confidence : A Step-By-Step Guide*. John Wiley & Sons, 1997. 7

[Miy90] MIYATA K.: A method of generating stone wall patterns. In *Proceedings of ACM SIGGRAPH 90* (1990), pp. 387–394. 3

[MMK*00] MARKOSIAN L., MEIER B. J., KOWALSKI M. A., HOLDEN L. S., NORTHRUP J. D., HUGHES J. F.: Art-based rendering with continuous levels of detail. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)* (2000), pp. 59–66. 3

Figure 8: *Screen shot of an interactive session using the Olympic Village model*

[Per85] PERLIN K.: An image synthesizer. In *Computer Graphics, Vol. 19, No. 3.* (1985), pp. 287–296. 4

[RS95] RICHENS P., SCHOFIELD S.: Interactive computer rendering. *Architectural Research Quarterly 1*, 1 (1995), 1–18. 3

[SABS94] SALISBURY M. P., ANDERSON S. E., BARZEL R., SALESIN D. H.: Interactive pen-and-ink illustration. In *Proceedings of ACM SIGGRAPH 94* (1994), pp. 101–108. 3

[SALS96] SALISBURY M., ANDERSON C., LISCHINSKI D., SALESIN D. H.: Scale-dependent reproduction of pen-and-ink illustrations. In *Proceedings of ACM SIGGRAPH 96* (1996), pp. 461–468. 3

[Sch96] SCHOFIELD S.: Piranesi: A 3-d paint system. In *Eurographics UK 96 Conference Proceedings* (1996). 3

[SFWS03] SOUSA M. C., FOSTER K., WYVILL B., SAMAVATI F.: Precise ink drawing of 3d models. *Computer Graphics Forum 22*, 3 (2003), 369–379. 2

[SMI99] STROTHOTTE T., MASUCH M., ISENBERG T.: Visualizing knowledge about virtual reconstructions of ancient architecture. In *Proceedings of Computer Graphics International* (June 1999), pp. 36–43. 2

[SPR*94] STROTHOTTE T., PREIM B., RAAB A., SCHUMANN J., FORSEY D. R.: How to render frames and influence people. *Computer Graphics Forum, Proceedings of EuroGraphics 1994 13*, 3 (1994), 455–466. 2

[SS02] STROTHOTTE T., SCHLECHTWEG S.: *Non-Photorealistic Computer Graphics. Modelling, Animation, and Rendering*. Morgan Kaufmann, 2002. 2

[SSRL96] SCHUMANN J., STROTHOTTE T., RAAB A., LASER S.: Assessing the effect of non-photorealistic rendered images in cad. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Common Ground* (1996), pp. 35–41. 2

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-d shapes. *Proceedings of ACM SIGGRAPH 90 24*, 4 (1990), 197–206. 2, 3

[SWHS97] SALISBURY M. P., WONG M. T., HUGHES J. F., SALESIN D. H.: Orientable textures for image-based pen-and-ink illustration. In *Proceedings of ACM SIGGRAPH 97* (1997), pp. 401–406. 3

[WPFH02] WEBB M., PRAUN E., FINKELSTEIN A., HOPPE H.: Fine tone control in hardware hatching. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002)* (2002), pp. 53–58. 3

[WS94] WINKENBACH G., SALESIN D. H.: Computer-generated pen-and-ink illustration. In *Proceedings of ACM SIGGRAPH 94* (1994), pp. 91–100. 3, 7

[Yes79] YESSIOS C. I.: Computer drafting of stones, wood, plant and ground materials. 190–198. 3

[ZHH96] ZELEZNIK R. C., HERNDON K. P., HUGHES J. F.: Sketch: An interface for sketching 3d scenes. In *Proceedings of ACM SIGGRAPH 96* (1996), pp. 163–170. 3

Appendix: OpenGL Shading Language Vertex Program

```
uniform float k;
void main(void)
{
    vec3 face_normal = vec3(gl_Normal);
    normalize(face_normal);

    vec3 perturb = vec3(gl_MultiTexCoord1.x,
            gl_MultiTexCoord1.y, gl_MultiTexCoord1.z);
    normalize(perturb);

    vec3 v = cross(face_normal, perturb);
    vec3 u = cross(face_normal, v);
    mat3 plane = mat3(u, v, face_normal);
    mat3 plane_inv = mat3(u.x, v.x, face_normal.x,
            u.y, v.y, face_normal.y,
            u.z, v.z, face_normal.z);
    mat3 ortho_proj = mat3(1.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0);

    perturb = plane_ori * ortho_proj * plane * perturb;
    normalize(perturb);

    vec3 mod_v = vec3(gl_Vertex) + k * (perturb * 0.1) +
            (face_normal * 0.01);
    gl_Position = gl_ModelViewProjectionMatrix *
            vec4(mod_v, 1.0);

    gl_FrontColor = gl_Color;
}
```