Texture Guided Realtime Painterly Rendering of Geometric Models

Shiben Bhattacharjee[1]* Neeharika Adabala[2]

[1] CVIT, International Institute of Information Technology Hyderabad
[2] Microsoft Research India

Abstract. We present a real-time painterly rendering technique for geometric models. The painterly appearance and the impression of geometric detail is created by effectively rendering several brush strokes. Unlike existing techniques, we use the textures of the models to come up with the features and the positions of strokes in 3D object space. The strokes have fixed locations on the surfaces of the models during animation, this enables frame to frame coherence. We use vertex and fragment shaders to render strokes for real-time performance. The strokes are rendered as sprites in two-dimensions, analogous to the way artists paint on canvas. While animating, strokes may get cluttered since they are closely located on screen. Existing techniques ignore this issue; we address it by developing a level of detail scheme that maintains a uniform stroke density in screen space. We achieve painterly rendering in real-time with a combination of object space positioning and image space rendering of strokes. We also maintain consistency of rendering between frames . We illustrate our method with images and performance results.
Keywords: Non-Photo-realistic Rendering, Real-time Painterly Rendering, Stroke Based Rendering, Texture Guided Strokes, Levels of Detail of Strokes.

1 Introduction

Paintings are often used to depict ideas. The aesthetics and expressiveness of paintings enables effective capture of the intentions of the artist. Animations are therefore often created in painterly style. In recent times computers are often used to generate the environments in cartoon based entertainment. (Eg. Titan A.E., Transformers etc.) Use of computers saves artists from the tedious need to create various views of the same static environment, but it leads to a visual disparity between the hand drawn objects and the environment as computer generated images appear synthetic and lack abstraction. Painterly rendering, a non-photorealistic rendering technique, can harmonize the composition of hand drawn elements and the computer modeled environment. Therefore, painterly rendering has been the focus of several graphics researchers.

More recently games depicting cartoon like appearance based upon cartoon serials/movies (Eg. Teenage Mutant Ninja Turtles, 2004) have been made. These

* This work was done while being an intern in Microsoft Research India

games could benefit from real-time painterly rendering. When painterly rendering is applied in gaming scenarios one has to address two key issues namely, frame to frame coherence, and level of detail management. In this paper we present a real-time painterly rendering algorithm that addresses theses issues. Existing painterly rendering techniques for animations employ geometry alone for placement of strokes and ignore textures that are a crucial part of models. They also do not address the issue of cluttered strokes. We present a painterly rendering technique that uses texture guided stroke placement on models and handles problems due to cluttering of strokes.

The organization of the rest of the paper is as follows: We briefly describe related work in the following section. We outline our technique in the section 3 and give details on stroke position computation, classification of strokes and rendering of strokes. We also describe a technique to address problem of stroke cluttering. Illustrations of our results and the performance of our system are discussed in section 4. We conclude with a discussion on the aesthetic considerations and technical aspects in section 5.

2 Related Work

Abstract representation of still images was introduced by Haeberli [1], he uses image color gradient and user interactivity for painting. Hertzmann [2] places curved brush strokes of multiple sizes on images for painterly rendering. The technique fills color by using big strokes in the middle of a region and uses progressively smaller strokes as one approaches the edges of the region. Shiraishi and Yamaguchi [3] improves the performance of above method by approximating the continuous strokes by placement of rectangular strokes discreetly along the edges to create painterly appearance. Santella and DeCarlo[4] used eye tracking data to get points of focus on images and create painterly rendering with focus information. All these techniques work well on single images but they usually involve iterative (optimization) techniques that make them cumbersome for real-time applications (see [5]). Also if they are applied on each frame of an animation independently, it often leads to flickering of strokes due to incoherence of strokes between frames.

Painterly rendering for animation was introduced in Meier's work [6] which focuses on eliminating shower door effect and achieve frame to frame coherence. Non existence of programmable graphics hardware, however, made the technique non-realtime. Also the method was limited to fetch stroke properties from geometry. Klein et al. [7] used realtime creation of painterly textures for painterly rendering using image based rendering algorithms for simple geometric models. Their algorithm lacks frame to frame coherence. Haller and Sperl [8] describes a realtime painterly process inspired by Meier [6]. Their approach makes the painterly rendering process work in real-time with the help of programmable graphics hardware. The method extracts stroke properties from geometry alone, and it does not address the problem of cluttering of strokes with changes in viewpoint.

3 Our Approach

A painting is created by placing several brush strokes of various shapes at specific locations on the canvas. In our approach we use the textures of models to enable us to select the number, location and shape of strokes to render. The position of the strokes are defined by the image space coordinates of a pixel in a texture and property of the stroke is stored as the pixel value at that location. We call the resulting image as a **feature image**. The stroke locations are in the image/texture space; we transform the 2D positions of stroke locations to 3D object space coordinates for painterly rendering.

The outline of the algorithm is as follows:

```
Start:
  Load various stroke textures;
  Load Model Information;
  Extract features from Model's textures;
  Transform features from Image to 3D space;
  For each frame:
    PASS1:
      Draw the object;
      Save the screen as a texture;
      Save the depth information as a texture;
    PASS2:
      Draw Edge strokes;
      Draw Filling strokes;
      Draw Feature strokes;
End
```

In each frame we render the object/scene and save it as a **reference image**. We also save the depth information as a **depth texture**. In the next pass, we render sprites at the stroke locations using vertex and fragment shaders. These brush-strokes are categorized based on the details given in section 3.1. The sprites are texture mapped with brush stroke textures, alpha blended and associated with color information from the reference image. The depth texture is used to decide the visibility of sprites/strokes. When a face of the object occupies less area in the screen space, the strokes in that region become cluttered and overlap. We use levels of detail to overcome the cluttering. The level of detail scheme, however, has a popping artifact during transitions. We develop an approach for smooth level of detail change, the details of which are presented in section 3.3.

3.1 Feature Extraction from Textures

This section describes the technique we use to obtain features from textures. When painting, one has to decide locations of strokes, number of strokes, shapes of strokes and orientations of strokes. These properties are gathered from the textures associated with the model. For each texture, example figure 1(a), we use a simple Sobel's edge detector [9] to get the edges and store them in the gray-scale feature image. Then this edge map is used to get another set of edges running parallel along the detected edges. These parallel edges store the angle along which the edge is oriented at each pixel as the pixel value as shown in figure 1(b). The angles in the range $[0, \pi]$, are discretized and scaled in the range $[100, 200]$ as the pixel value in the feature image. We then stipple the lines by running a mask on the lines and nullifying a number of surrounding pixels. The size of the mask is a parameter that decides the concentration of strokes while rendering since each non-null pixel location represents a stroke. A smaller size implies a greater concentration of stroke locations, which in turn implies a larger number of strokes. The generated points on the original edges have pixel value 255. Strokes at these locations are called **edge strokes**. Strokes on the points, which are on the lines parallel to the edges, are called **feature strokes**. In the remaining empty area we distribute points with a pixel value 64, with random spacing as shown in figure 1(c). These strokes are called **filling strokes**.

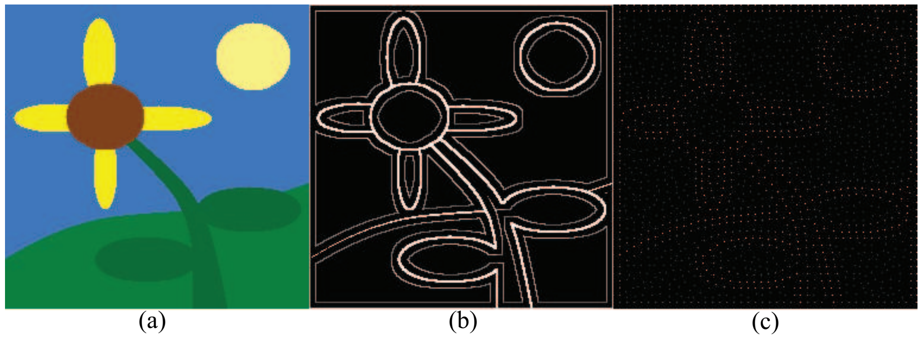

Fig. 1. (a) Example texture; (b) Detected Edges and parallel edges storing the orientation; (c) Final **feature image** giving stroke locations; here different pixel values indicate whether the strokes are edge, feature or filling strokes.

3.2 Stroke Location Transfer from Image to Object Space

As a pre-processing step we transform the positions of pixels in the feature image to object space depending upon which face the strokes are stuck to. We use simple geometric transformation equations to solve this issue. We find the 3D points (x, y, z) for any pixel (X, Y) as

$$aX + bY + k = x$$

$$cX + dY + k = y$$

$$eX + fY + k = z$$

For any 3 pixels in the texture $(X_1, Y_1), (X_2, Y_2), (X_3, Y_3)$, if we know 3 object coordinates $(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)$ (Eg. 3 corners of a triangular face), we can solve the equations for a, b, c, d, e, f. We save stroke positions and their properties for each face.

3.3 Rendering

Rendering consists of two passes; the first pass renders the object and saves it on a texture as the reference image. The second pass involves calculation of a Level of Detail (LOD) factor and a blending factor for each face depending upon its screen occupancy. Less the occupancy, lower the level of detail associated with the face during rendering. The order of rendering strokes is based on their category. Rendering of some strokes is skipped based on the LOD.

Creation of Reference Image We render the textured geometric model, and copy the output to a texture example figure 2(a). The depth information is copied to a depth texture.

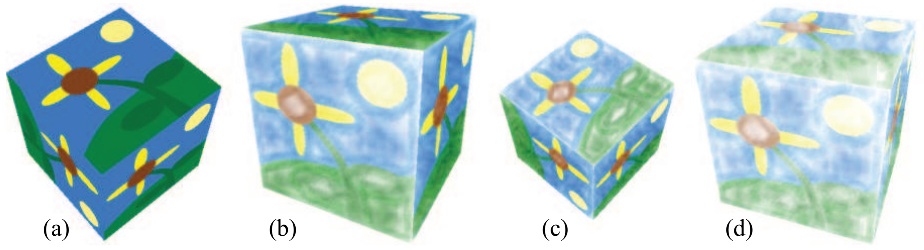

Fig. 2. (a) Reference Image; (b) Cluttered strokes of oblique faces; (c) Cluttered strokes when far; (d) No clutter when with LOD scheme

Calculation of Levels of Detail We see a cluttering of strokes, in figure 2(c), when the geometric model is distant. We calculate the distance of the camera from each face. This distance d is used to calculate LOD indicator l_d which is the level of detail due to distance, as $N(d - min)/(max - min)$ where max and min are the maximum and minimum distance respectively the object travels from the eye position, and N is the number of LODs available. When the face is at min distance to camera, l_d equals 0 and when the face is at max distance to camera, l_d equals N.

However, faces also cover less screen space when they are nearly parallel to the viewing direction as shown in figure 2(b). To address this issue, we calculate another LOD indicator, level of detail due to orientation l_o, as $l_o = (1 - |n.v|)N$, where n is the normal of the face and v is the unit view vector. As per dot product's nature, $l_o = N$ when n and v are perpendicular to each other i.e. face is completely out of view, and $l_o = 0$ when n and v are equal i.e. face completely faces you.

Thus we use $l_d = 0, l_o = 0$ as the highest LOD indicators and $l_d = N, l_o = N$ as the lowest LOD indicators. We take the weighted mean of the two and use the value for assigning an LOD factor l and blending factor α to the face as follows:

$$l = [w_o l_o + w_d l_d], \alpha = 1 - \{w_o l_o + w_d l_d\}$$

where w_d and w_o are user decided weights for farness and orientation respectively. We skip 2^l number of strokes while drawing stokes for a face. For a stroke with index i in a face, if the expression $mod(i, 2^{(l+1)})$ returns a non null value, means that this stroke is skipped when LOD changes for this face. We multiply α with this stroke's opacity, so that it gradually becomes transparent as the face approaches the next LOD transition. When the face shifts to the next LOD, this stroke is dropped but we do not see any popping artifact since it gradually becomes totally transparent. Ours strokes do not clutter, example figure 2(d). The calculations involving assignment of blending factor is done on the GPU with the help of vertex shaders explained in more detail in the next section.

Rendering of Strokes We render the edge strokes first. For each edge stroke we pass on the edge stroke location to the vertex shader 4 times with 4 texture coordinates of a randomly chosen perturber texture. Sample edge stroke textures are shown in figure 3(a). This randomness is pre-computed to avoid

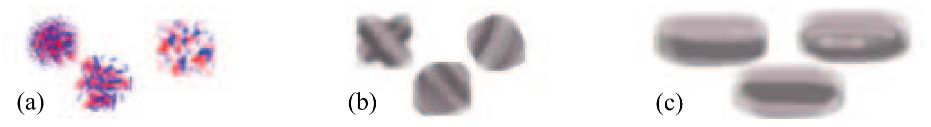

Fig. 3. (a) Edge Stroke textures; (b) Filling Stroke textures; (c) Feature Stroke textures

inconsistency between frames. A vertex shader, which we call **VS1** for future reference, calculates the sprite coordinates using the texture coordinates. We calculate the sprite coordinates after we have applied the model-view transformation to the stroke location. To maintain constant sprites size, we calculate the sprite coordinates after we have projected the stroke location. A fragment shader, which we call **FS1** for future reference, picks color information from the edge stroke texture; uses the red stream as the amount of perturbation in x axis direction and blue stream as the amount of perturbation in y axis direction, of the location of pixel of reference image (see figure 4(b)).

$$p_x = 2C_r - 1, p_y = 2C_b - 1$$

$$O_c = I_c T_{x+kp_x, y+kp_y}$$

where p is the disturbance with a scale k in the reference texture's T coordinates x, y at that fragment location, C is the color of the stroke texture at that fragment location and I_c is the optional input color for the whole stroke. O_c is output fragment color of the **FS1**. Also we use the blending factor and multiply it with the opacity of the stroke as explained in the previous section. Now using the filling stroke coordinates and filling stroke textures randomly chosen from available ones as show in figure 3(b). We render the filling strokes as sprites. We use the same vertex shader **VS1** but a different fragment shader, which

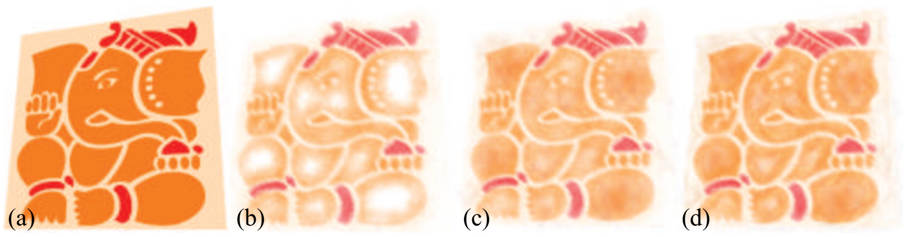

Fig. 4. Lord Ganesha; Using (a) Reference texture, (b) Rendering of edge strokes followed by (c) Rendering of filling strokes and then (d) Finally rendering of feature strokes to get the final output

we call **FS2** for future reference, since we want to perturb the color picked up from the reference texture. **FS2** uses the filling stroke texture's color streams to change the color of the background reference texture at that fragment location (see figure 4(c)).

$$p = 2C - 1$$

$$O_c = I_c(T_{x,y} + kp)$$

where the notations mean the same as explained earlier. **FS2** does the same job as **FS1** regarding the blending factor. Next we use feature stroke coordinates and feature stroke textures randomly chosen from available ones to render the feature strokes as sprites. Sample feature strokes are shown in figure 3(c). We use a different vertex shader, which we call **VS2** for future reference. **VS2** incorporates not only calculation of sprite coordinates but also rotation of the sprite in the image space according to feature information stored along with the stroke coordinate. The rotated strokes are rotated by another angle which is due to the animation of the model. We use **FS2** for the later part of the processing of this stroke. Figure 4(d), is a example of the output when all the strokes are rendered.

The strokes are alpha blended, therefore order of blending is important. This

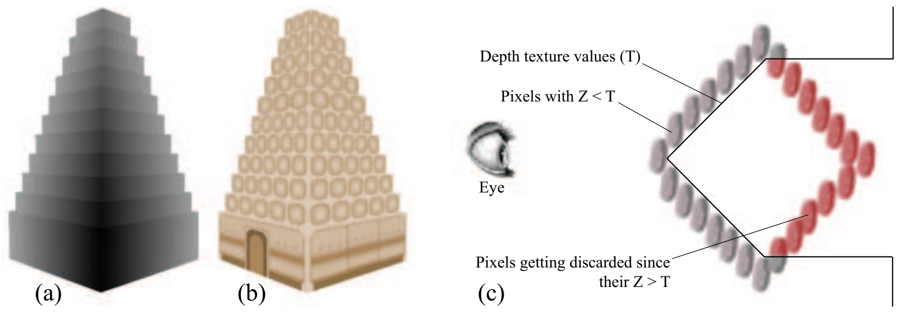

Fig. 5. (a) A sample depth texture for (b) A reference image; (c) Only pixels of strokes with depth less than depth texture value pass.

is where depth texture comes in picture. An example depth texture is shown in figure 5(a). In **FS1** and **FS2** we test the depth of the pixel to be less than or equal to the depth value at that location in the depth texture, if the pixel does not pass the condition, it is discarded as illustrated in figure 5(c). We use a small offset when testing since our strokes are front facing sprites with constant depth. All strokes drawn have a maximum opacity less than 1, so that the rendering is relative to the background color. This is consistent with an artist using water colors, the painting has a tone of the color of the paper being used.

4 Results

We use a system with the following specifications: Intel Pentium 4 3.4 GHz, 2.00 GB RAM, nVIDIA 6800 Ultra. We implement the algorithm in C++, with libraries OpenGL, SDL and CGgl along with Nvidia CG for shaders. We render simple models: cube with 6 faces, building with 15 faces, a South-Indian Style temple with 55 faces and a tall building with 1000 faces. The results are given

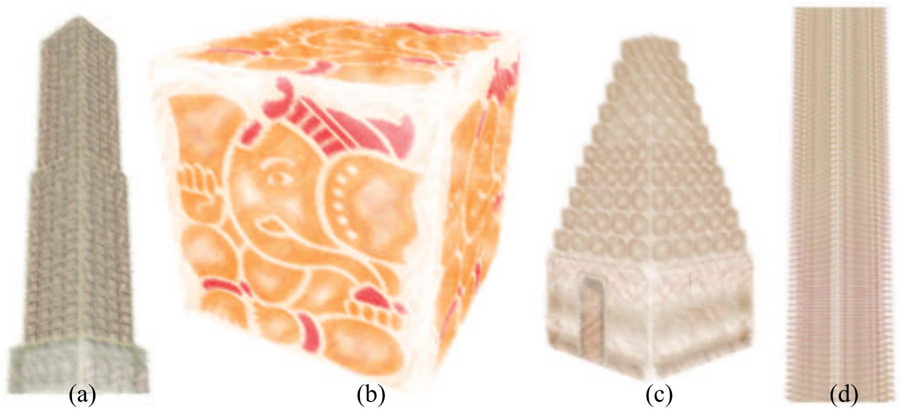

Fig. 6. Various Painterly Rendered Objects: (a) Building; (b) Cube; (c) Temple; (d) Tall building

in table 1 and outputs are shown in figure 6. The cube with LOD system gives a frame rate of 150 to 200 as the model oscillates between the near and far plane of the camera respectively. The performance of the system is dependent on the type of strokes that are rendered as the shaders have different calculations for different category of strokes. The speed of the system is mainly influenced by the number of strokes. Example as given in table 1, the cube with 6 faces and 4266 stroke count gives similar frame rates as the Tall Building with 1000 faces and 4400 stroke count. We do most of the calculations on the GPU. Only the calculations for LOD are done on the CPU as they decide the primitives that are getting rendered rather than computations that are performed on the primitives.

Table 1. Frame Rates for Model: Building

Model	No. of faces	No. of Strokes	FPS
Building	15	7455	107
Building	15	3727	185
Building	15	1863	357
Temple	55	7170	89
Temple	55	3585	170
Temple	55	1797	402
Cube	6	8532	60
Cube	6	4266	120
Cube	6	2133	232
Tall Building	1000	51200	16
Tall Building	1000	25600	30
Tall Building	1000	12800	50
Tall Building	1000	17600	51
Tall Building	1000	8800	81
Tall Building	1000	4400	140

5 Conclusions and Future Work

We presented a system which produces a painterly rendering of simple geometric models. Its a combination of stroke based rendering of still 2D images and painterly rendering in 3D. The visual appearance depends on the number of strokes used, the stroke textures, the size of strokes. In some scenes, when less strokes are used, it gives a nice visual appearance of a light water color drawing. Large strokes bring abstract effect whereas small strokes bring accuracy to the object. Stroke texture used should have a smooth gradient content, high frequency stroke textures create discreteness between adjacent strokes and spoil the hand drawn appearance.

As future work, we will explore making technical improvements to our implementation at various places. Copying the scene and depth texture after PASS1 as explained in section 3.3 are done by the `glCopyTexImage2D()` function. We can improve the implementation by rendering directly to textures with the help of `pbuffers`. The visibility testing of strokes is done on a fragment shader, i.e. on all of it's pixels. This can be done even more efficiently if we can access the depth texture at the vertex shader level (we want a stroke to be visible as a whole or not). Vertex texture fetch is a possibility, however vertex textures are slow and are limited to vendor and specific data types. We are studying vertex texture fetch improvements.

References

1. Haeberli, P.: Paint by numbers: abstract image representations. In: SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM Press (1990) 207–214

2. Hertzmann, A.: Painterly rendering with curved brush strokes of multiple sizes. Computer Graphics **32** (1998) 453–460
3. Shiraishi, M., Yamaguchi, Y.: An algorithm for automatic painterly rendering based on local source image approximation. In: NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering, New York, NY, USA, ACM Press (2000) 53–58
4. Santella, A., DeCarlo, D.: Abstracted painterly renderings using eye-tracking data. In: NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering, New York, NY, USA, ACM Press (2002) 75–ff
5. Hertzmann, A.: Tutorial: A survey of stroke-based rendering. IEEE Comput. Graph. Appl. **23** (2003) 70–81
6. Meier, B.J.: Painterly rendering for animation. Computer Graphics **30** (1996) 477–484
7. Klein, A.W., Li, W.W., Kazhdan, M.M., Correa, W.T., Finkelstein, A., Funkhouser, T.A.: Non-photorealistic virtual environments. In Akeley, K., ed.: Siggraph 2000, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman (2000) 527–534
8. Haller, M., Sperl, D.: Real-time painterly rendering for mr applications. In: GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, New York, NY, USA, ACM Press (2004) 30–38
9. Gonzalez, R.C., Woods, R.: Digital Image processing. (Addison-Wesley)
10. Hertzmann, A., Perlin, K.: Painterly rendering for video and interaction. (2000)
11. Hertzmann, A., Jacobs, C.E., Oliver, N., Curless, B., Salesin, D.H.: Image analogies. In Fiume, E., ed.: SIGGRAPH 2001, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH (2001) 327–340
12. Hertzmann, A.: Fast paint texture (2002)
13. Strothotte, T., Masuch, M., Isenberg, T.: Visualizing Knowledge about Virtual Reconstructions of Ancient Architecture. In: Proceedings Computer Graphics International, The Computer Graphics Society, IEEE Computer Society (1999) 36–43
14. Freudenberg, B., Masuch, M., Strothotte, T.: (Walk-through illustrations: Frame-coherent pen-and-ink style in a game engine)
15. Lee, H., Kwon, S., Lee, S.: Real-time pencil rendering. In: NPAR '06: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering, New York, NY, USA, ACM Press (2006) 37–45
16. DeCarlo, D., Santella, A.: Stylization and abstraction of photographs. In: SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM Press (2002) 769–776
17. Santella, A., DeCarlo, D.: Visual interest and npr: an evaluation and manifesto. In: NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering, New York, NY, USA, ACM Press (2004) 71–150
18. Nienhaus, M., Dollner, J.: Sketchy drawings. In: AFRIGRAPH '04: Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa, New York, NY, USA, ACM Press (2004) 73–81