HYPERTEXTURING COMPLEX VOLUME OBJECTS

R. A. Satherley, M. W. Jones

Department of Computer Science
University of Wales, Swansea
Singleton Park, Swansea
United Kingdom
{csrich, m.w.jones}@swansea.ac.uk

ABSTRACT

This paper will examine hypertexture rendering techniques and will demonstrate how volume datasets may be adapted in order for hypertexture to be applied. Details are given of a process for the conversion of complex objects, such as CT scans, into accurate distance fields. Hypertexture is applied to these objects and example renderings include the UNC CThead, a chess piece, a dodecahedron and a tank. Additional information is given about soft objects, density modulation functions, ray marching and controlling hypertexture application.

CR Categories: I.3.7[Computer Graphics]:Three-Dimensional Graphics and Realism — Color, Shading, Shadowing and Texture
Keywords: Hypertexture, Distance Transform, Distance Field

1 INTRODUCTION

Traditional texture mapping techniques, such as Peachey [Peach85] and Perlin's [Perli85] *solid textures*, are limited to the reproduction of textures that have *simple* surface definitions. Many natural textures, such as fur, have surface definitions that are at best complex. Others, such as fire and smoke, have no well defined surface at all and are therefore unreproducible with such methods.

Hypertexture [Perli89], developed by Perlin and Hoffert, allows the reproduction of such complex textures through the manipulation of surface densities. This method is restricted to objects which can be defined using a modified form of implicit function ($f(x, y, z) = D(x, y, z)$), such as spheres and tori. Dischler and Ghazanfarpour [Disch95] produced hypertexture like effects on more general objects, with the use of *geometrical based textures*. A method which involves defining a *skeleton* for an object, from which densities are interpolated. Again this method is limited as it is only applicable to closed objects.

This paper aims to introduce a new method for the application of hypertexture to complex (open or closed) volume objects, such as CT scans and voxelised polygon meshes, with the use of *distance fields*. Distance fields are well suited to hypertexture application, as each point in the field gives the distance to its closest surface point, from which density is calculated easily.

Section 2 will briefly introduce the process of hypertexture application, summarising the concepts of *soft objects* (Section 2.1) and *density modulation functions* (Section 2.2). In Section 2.3 *ray marching* is introduced and examples of hypertexture effects are given in Section 2.4. Section 3 will develop the new idea of using distance fields as a basis for applying hypertexture to complex objects. Section 3.1 will demonstrate how scanned data, such as CT datasets, and previously voxelised (shell) datasets ([Jones96]) can be converted into distance fields using a *distance transform*. Section 3.2 will introduce a new faster and more accurate vector distance transform, the *vector-city vector distance transform (VCVDT)*. Example images will be given in Section 3.3. Finally, Section 4 introduces means of controlling the application of hypertexture.

2 HYPERTEXTURE

It has already been stated that Perlin and Hoffert developed hypertexture [Perli89] as a method for replicating natural phenomena, such as fire and fur, on implicit surfaces. This section will briefly summaries the main aspects of hypertexture, giving examples of the achievable effects.

2.1 Soft Objects

Before describing how to produce hypertexture effects, it is necessary to introduce *soft objects*. A soft object is an object where the thin surface boundary has been extended into a larger *soft region*, across which density varies from 1 (inside the object) to 0 (outside the object).

The soft region (and therefore the object) is modelled using an *object density function, $D(p)$*. An example of which, describing a sphere, is given in Eq. 1 and is illustrated in Fig. 1. Note that the outer surface has been cut in order to give a better view of the two surfaces.

$$D(p) = \begin{cases} 1 & \text{if } |p|^2 \leq {r_i}^2 \\ 0 & \text{if } |p|^2 \geq {r_o}^2 \\ \frac{{r_o}^2 - |p|^2}{{r_o}^2 - {r_i}^2} & \text{otherwise} \end{cases} \quad (1)$$

where $r_i =$ inner radius,
$r_o =$ outer radius
and $p \in \mathbb{R}^3$

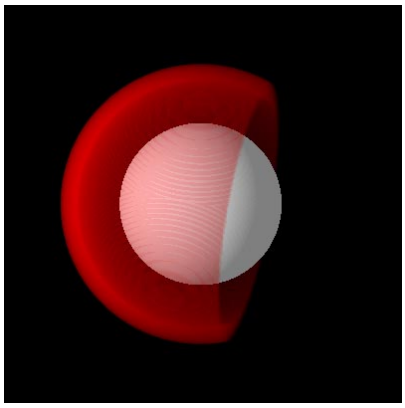


Figure 1: Illustration of Eq. 1.

Hypertexture effects can now be achieved by the repeated application of *Density Modulation Functions* (*DMF*, Section 2.2) to the soft region of $D(p)$, Eq. 2.

$$\begin{aligned} &H : \mathbb{R} \times \mathbb{R}^3 \rightarrow r, g, b, \alpha \\ &H(D(p), p) = DMF_n(\ldots(DMF_0(D(p)))) \end{aligned} \quad (2)$$

2.2 Density Modulation Functions

The density modulation functions can be separated into the three categories below.

- *Position dependent* – functions that are dependent on p.
- *Position independent* – scalar argumented functions.
- *Geometry dependent* – functions that are dependent on local geometry, e.g. surface normals.

Complex density modulation functions are constructed, as shown in Eq. 2, from the following basis functions, along with control functions such as *cosine* (*cf* Perlin's solid textures [Perli85]).

- *Noise* – used to approximate band-limited white noise, producing a pseudo-random value in the range $(-1, 1)$. Several implementations are given by Lewis [Lewis89].
- *Turbulence* – simulates the appearance of Brownian motion (turbulent flow) by the summation of noise at increasing frequencies (Eq. 3), introducing a self-similar $\frac{1}{f}$ pattern.

$$turb(p) = \sum_i abs\left(\frac{noise(2^i p)}{2^i}\right) \quad (3)$$

- *Bias* – defined by the power curve of Eq. 4, bias is used to shape the density across the soft region.

$$bias_b(D(p)) = D(p)^{\frac{ln(b)}{ln(\frac{1}{2})}} \quad (4)$$

- *Gain* – defined as a combination of two bias curves (Eq. 5), gain is used to alter the rate at which the density changes in the midrange of the soft region.

$$gain_g(D(p)) = \begin{cases} \frac{bias_{1-g}(2D(p))}{2} & \text{if } D(p) < \frac{1}{2} \\ 1 - \frac{bias_{1-g}(2-2D(p))}{2} & \text{otherwise} \end{cases} \quad (5)$$

2.3 Implementation

The absence of a well defined surface for the natural phenomena modelled by hypertexture, means that hypertextured objects will also have no well defined surface. This necessitates the use of a *ray marcher*, a simplified form of volume rendering ([Levoy88]), when creating hypertexture effects.

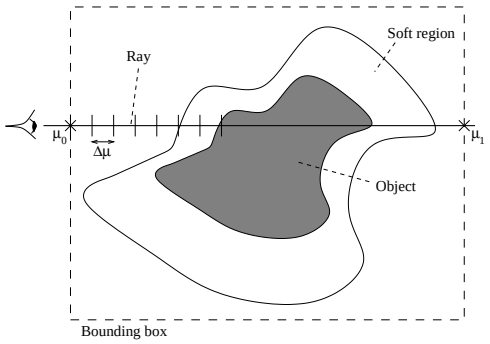


Figure 2: The ray marching process.

A simplified model of a ray marcher is given in Fig. 2. A ray, r, is fired from every pixel into the object space. If a ray intersects the object's bounding box, the entry (μ_0) and exit (μ_1) points for this ray are calculated. $D(p)$ is now evaluated at fixed points along the ray according to Eq. 6.

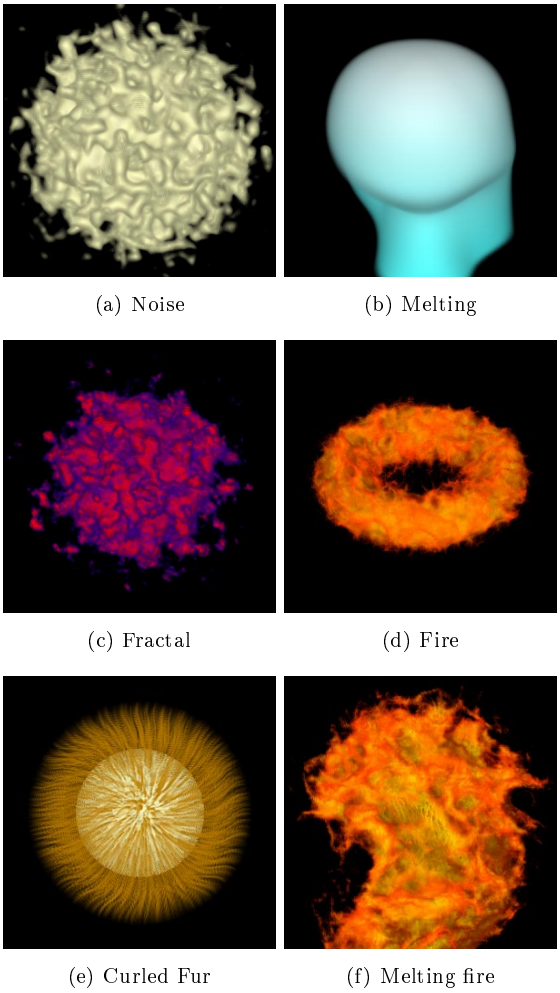
$$r(k) = \mu_0 + k\Delta\mu \quad \text{where } k \in \mathbb{N} \qquad (6)$$

If p lies within the soft region, the necessary DMF combination is applied, and a colour, C_λ ($\lambda = r, g, b$), and opacity, α, for this point of the ray are obtained. This process is repeated along the ray, accumulating colour and opacity, until one of following three criterion is met.

- *Ray termination* – $r(k) = \mu_0 + k\Delta\mu \geq \mu_1$
- *Inner surface reached* – $D(p) = 1$
- *Opaque surface* – $\alpha \geq 1$

2.4 Examples of Hypertexture

- *Unconditional noise* – noise is added to each component of p – Fig. 3(a).
- *Directed noise* – noise is added to only one of the components of p, creating a melting effect – Fig. 3(b).
- *Fractal noise* – noise of increasing frequency and decreasing amplitude is summed and added to p – Fig. 3(c).
- *Fire* – turbulence is added to each component of p – Fig. 3(d).
- *Fur* – fur filaments are *grown* from random positions – Fig. 3(e).
- *Melting fire* – a fire hypertexture has been applied to a previously melted sphere – Fig. 3(f).



(a) Noise (b) Melting

(c) Fractal (d) Fire

(e) Curled Fur (f) Melting fire

Figure 3: Examples of hypertexture effects.

3 HYPERTEXTURED VOLUME DATA

The fact hypertexture is implemented with a ray marcher (theoretically) allows it to operate on volume data. Problems arise when it is not possible to derive an implicit function for an object which is traditionally rendered using volume based methods, for example CT scan datasets.

Examination of the datasets used for the images of Fig. 3 and other mathematically generated datasets, shows that the voxels give values related

to the *distance* to some *feature point / surface*. Therefore, complex objects could be rendered with hypertexture if the dataset could be converted into a distance volume, D, where $D(p)$ is the distance from p to the closest point on the object's surface. For volume data, this can be achieved by measuring the distance between every background voxel and every surface voxel and then taking the minimum. This is extremely computationally expensive – taking over 62 hours for the UNC CThead. This can be improved to 8 hours by employing an octree and using a neighbour's closest voxel as a candidate to eliminate parts of the tree.

The run-time of the distance conversion process can be considerably improved, at the cost of reduced accuracy, with the use of a *distance transform (DT)* [Rosen66, Danie80, Borge86, Mulli92, Borge96] – a process used to approximate Euclidean distance calculations via *local distance propagation*. In Section 3.1 the distance transforms process will be introduced, followed by a description of the *vector-city vector distance transform* – a new more accurate distance transform, in Section 3.2. Example images will also be given.

3.1 Distance Transforms

Distance transforms can be separated into two categories, *chamfer distance transforms (CDT)* and *vector distance transforms (VDT)*. The difference being CDTs propagate distance by addition of known local neighbourhood distances, whereas vectorial information, from which distances are calculated, is propagated by VDTs. Both categories are implemented as a two stage process – surface extraction, followed by distance propagation.

The first stage of a DT is to *segment* the dataset, f, on a regular grid, via a thresholding operation, τ, to extract the surface of interest, S.

$$S = \{(x,y,z) : f(x,y,z) \geq \tau\} \quad \text{where } x,y,z \in \mathbb{Z} \tag{7}$$

3.1.1 Chamfer Distance Transforms

The preliminary distance field, for a chamfer distance transform, is constructed using Eq. 8.

$$D(p) = \begin{cases} 0 & \text{if } p \in S \text{ and} \exists\ q \in p_{26}, q \notin S \\ \infty & \text{otherwise} \end{cases} \tag{8}$$

where $p \in \mathbb{Z}^3$ and p_{26} is the set of voxels which are the 26 neighbours of p

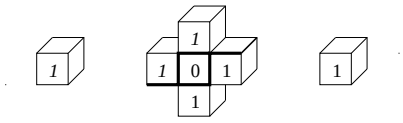
CDTs propagate distances with the use of two passes of the *distance matrix*, d_M, applying Eq. 9 to each voxel in turn, as demonstrated in the pseudo-code below.

$$D(p) = min\big(D(x+i,y+j,z+k)+d_M(i,j,k)\big) \quad \forall\ i,j,k \in d_M, \text{ where } p \in \mathbb{Z}^3 \text{ and } i,j,k \in \mathbb{Z} \tag{9}$$

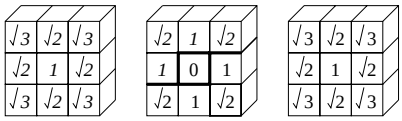
```
/* Forward Pass */
FOR(z = 0; z < f_z; z++)
  FOR(y = 0; y < f_y; y++)
    FOR(x = 0; x < f_x; x++)
      D(x,y,z) = Eq.9

/* Backward Pass */
FOR(z = f_z-1; z ≥ 0; z--)
  FOR(y = f_y-1; y ≥ 0; y--)
    FOR(x = f_x-1; x ≥ 0; x--)
      D(x,y,z) = Eq.9
```

Two examples of a chamfer distance matrix are given in Fig. 4 (see [Borge86] for more examples). Note that each matrix element gives the distance to the central element.



(a) City-block



(b) Complete CDT

Figure 4: Chamfer distance matrices.

Overall each voxel is considered twice, with its distance calculation depending purely upon addition of the matrix elements to its neighbours and taking the minimum.

3.1.2 Vector Distance Transforms

VDTs generally require more passes of the distance matrix. During each pass the vector components are added to the necessary vector position, the distance calculated, a decision made as to whether any of the new distances are minimal, and finally the minimal vector is stored.

To allow vector propagation Eqs. 8 and 9 are modified as shown in Eqs. 10 and 11 respectively.

$$\vec{V}(p) = \begin{cases} (0,0,0) & \text{if } p \in S \text{ and} \\ & \exists\, q \in p_{26}, q \notin S \\ (\infty,\infty,\infty) & \text{otherwise} \end{cases} \quad (10)$$

$$D(p) = min\left|\vec{V}(x+i,y+j,z+k) + d_M(i,j,k)\right| \quad \forall i,j,k \in d_M, \text{ where } d_M = (\vec{M}_x, \vec{M}_y, \vec{M}_z) \quad (11)$$

Fig. 5 shows the matrix passes employed by Mullikin's *efficient VDT (EVDT)* [Mulli92] – the best VDT represented in the literature.

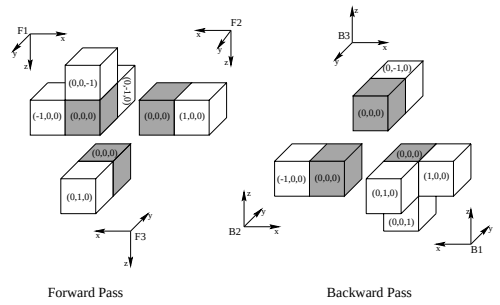


Figure 5: Mullikin's EVDT matrix passes

Overall each voxel attempts to update its distance once during each pass, with the number of calculations and comparison made each time dependent on the matrix used. For example the EVDT passes over the dataset six times, making a total of sixteen distance calculations and eleven comparison per voxel.

Table 1 summarises the findings of a comparison between the true Euclidean distance field, for the skull of the UNC CThead, and those generated using the city-block, complete CDT and EVDT distance matrices. From the table it can be seen that the EVDT is by far the superior of the implemented DTs, having an almost negligible average error per voxel.

3.2 The Vector-city VDT

The previous section concluded that vector distance transforms are far more accurate than chamfer distance transforms. In this section a new, more accurate, vector distance transform, based on the city-block CDT, will be introduced.

The city-block CDT is the most elementary distance transform. A fact which has led to it being the basis of the majority of VDTs. For example Danielsson's 4SED [Danie80] uses the same local neighbourhood and Mullikin's EVDT [Mulli92] uses the vector equivalent of the two city-block passes in its own. The new *vector-city vector distance transform (VCVDT)* extends this heuristic to include all four matrix passes. That is, the extra passes could also be implemented as part of the city-block CDT. Fig. 6 illustrates the matrix used during each pass. The dashed positions in Fig. 6 are omitted during the matrix passes as they represent redundant calculations, that is, they perform the same calculation as a previous position.

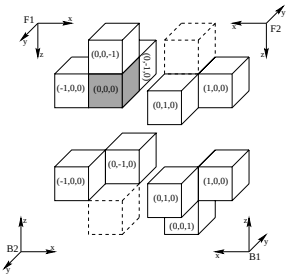


Figure 6: The VCVDT matrix passes

The vector-city vector distance transform is implemented in a manner similar to that of the EVDT. During each pass (*F1*, *F2*, *B1* and *B2*) the corresponding matrix segment is applied in the direction indicated by Fig. 6. At each voxel, its neighbours' vectors are altered according to the overlying matrix element and the minimal vector stored. If the situation arises where two or more of the distances are minimal, *a tie*, no preference is made as to which of the corresponding vectors is stored. This is also the case in the EVDT.

Unlike the EVDT, which only stores two vector slices, the VCVDT stores a complete vector copy of the distance volume. Storing only two vector slices was found to be the inferior of the two approaches, as vector information is lost as a pass progresses through the dataset.

Furthermore, by storing (after each pass) the minimal distance along with the minimum vector, the VCVDT is able to employ Leymarie and Levin's [Leyma92] optimisations. Thus removing the need to recalculate the distance for the central voxel, saving three distance calculations per voxel. Therefore, the VCVDT makes only eleven distance calculations and ten comparisons per voxel.

The accuracy of the VCVDT can be improved by increasing the number of passes made by the distance matrix from four to eight, as shown in

Distance matrix	Execution time (s)	Error range min ⇒ max	Incorrect voxels (%)	Average error per voxel
True Euclidean	≈28800.000	0.000 ⇒ 0.000	0.000	0.000000
City-block	1.320	-2.000 ⇒ 76.060	91.579	12.269071
Complete 3×3×3	4.842	-0.415 ⇒ 11.769	88.774	2.196785
EVDT	7.540	-0.518 ⇒ 2.533	3.449	0.004761

Table 1: Results of comparing the generated distance fields to the ture Euclidean distance field.

Fig. 7. As before the extra passes could be implemented as part of the city-block CDT, adding an extra eight distance calculations and comparisons per voxel. Thus, the eight pass VCVDT makes a total of nineteen distance calculations and eighteen comparisons per voxel.

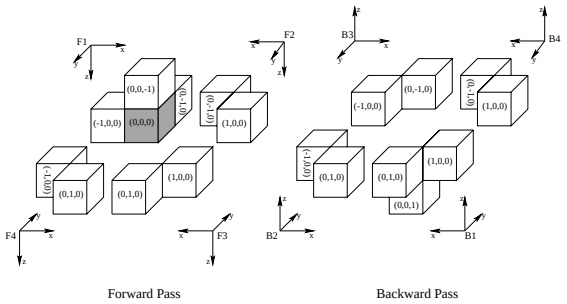


Figure 7: The 8VCVDT matrix passes

Table 2 shows the results of the comparison between the distance fields obtained from the four and eight pass VCVDT and the true Euclidean distance field. It can be seen from Tables 1 and 2 that the VCVDTs out perform the EVDT in both speed of execution and accuracy.

3.3 Hypertextured Distance Fields

Hypertexture can be easily added to a distance field by making a slight modification to the object density function (Eq 1), as shown in Eq. 12.

$$D(p) = \begin{cases} 1 & \text{if } \|p\| \le r_i \\ 0 & \text{if } \|p\| \ge r_o \\ \frac{r_o - \|p\|}{r_o - r_i} & \text{otherwise} \end{cases} \qquad (12)$$

where $r_i =$ inner radius,
$r_o =$ outer radius
$\|p\| =$ dataset value at p

That is, the value held at voxel p is used instead of the magnitude of p. Furthermore, as the dataset holds distance values, it is no longer necessary to square the inner and outer radii.

Fig. 8 demonstrates the application of hypertexture to distance fields of triangular mesh objects and the UNC CThead.

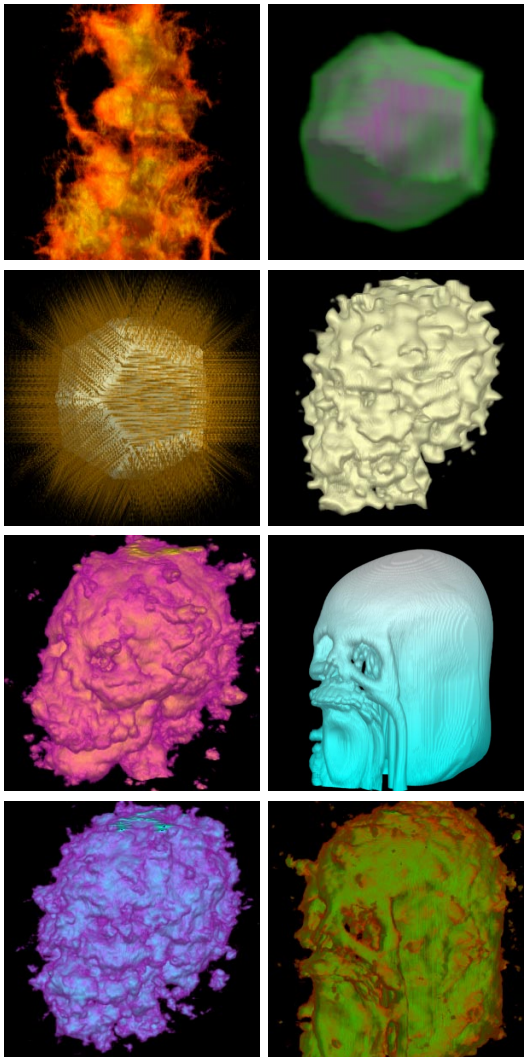


Figure 8: Hypertexture applied to the distance fields for a chess piece, a dodecahedron and the UNC CThead dataset.

4 CONTROLLING HYPERTEXTURE

There is little material available in the literature on ways to control the application of hypertexture

Distance matrix	Execution time (s)	Error range min ⇒ max	Incorrect voxels (%)	Average error per voxel
VCVDT	3.420	-0.334 ⇒ 0.446	1.300	0.000644
8VCVDT	5.470	-0.334 ⇒ 0.334	0.343	0.000223

Table 2: Results of the comparing the VCVDT distance fields to the true Euclidean distance field.

to an object. This section will demonstrate three application control methods which are applicable to hypertexture –

- *Clipping* – hypertexture is only applied to selected parts of the object.
- *Blending* – two or more textures can be blended together.
- *Animation* – animations can be created by altering control parameters.

4.1 Clipped Hypertexture

More often than not it is only necessary to texture a certain part of an object. This selective texture addition can be achieved if it is possible to identify such sections.

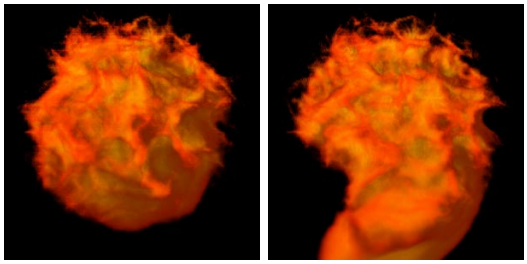
One approach is to define a clipping surface to separate the object into the desired sections. With the clipping surface defined, a simple calculation on the current ray location indicates whether or not the texture should be applied. In Fig. 9(a) a clipping surface has been constructed allowing the barrel of the tank to be melted. Fig. 9(a) also shows that an object does not have to be restricted to one type of texture, the tank's camouflage was produced using Perlin's *bozo* solid texture.

4.2 Blending Hypertexture

A problem with clipped hypertexture application is that artifacts can occur at the clipping surface if the textures used on either side are disjoint. These artifacts can be removed if the textures are blended together in the region of the clipping surface.

Examples of blended hypertexture are given in Fig. 9(b) and 9(c). Fig. 9(b) shows a fire hypertexture *fading* away as it moves down the sphere and Fig. 9(c) demonstrates a blend between a fire and melting hypertexture.

(a) Clipped hypertexture application



(b) Fading (c) Merged

Figure 9: Examples of controlled hypertexture.

4.3 Animation

As hypertexture is created by manipulating the surface densities of an object with a three dimensional density modulation function, the object may be rendered from different view points without any inconsistencies in the texture. Thus introducing frame coherency for animation. Furthermore, making a slight alteration in the control parameters of the hypertexture can produce some interesting animations. Fig. 10 shows four frames from a melting animation, here the frequency of the distorting noise is increased in each frame.

5 CONCLUSION

This paper has introduced a new method for the easy application of hypertexture to complex, open or closed, volume objects. In addition to showing that datasets obtained by evaluating an implicit function can be immediately hypertextured, it has also been shown that, with the use of a distance transform, complex volume objects may be

Figure 10: An animated hypertexture.

made to mimic such datasets, and therefore have a hypertexture applied.

Details have been given about the transform process, in particular two new VDTs – the four and eight pass VCVDT, have been introduced. The performance of the new VCVDTs has been compared that of the best distance transform represented by the literature – Mullikin's EVDT, and has shown the new methods to be both faster and more accurate.

A brief explanation of the main aspects of hypertexture – soft objects, density modulation functions and the ray marching process, has been given. Finally examples of hypertextured distance field encoded objects have been given, and also methods for controlling the application of hypertexture have been demonstrated.

ACKNOWLEDGEMENTS

We would like to thank Šrámek and Kaufman [Srame98, Srame99] for allowing us to use their voxelised tank. This work has been undertaken with funding from EPSRC, UK, under grant GR/L88238.

REFERENCES

[Borge86] G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34(3):344–371, 1986.

[Borge96] G. Borgefors. On digital distance transforms in three dimensions. *Computer Vision and Image Understanding*, 64(3):368–376, 1996.

[Danie80] P-E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[Disch95] J-M. Dischler and D. Ghazanfarpour. A geometrical based method for highly complex structured textures generation. *Computer Graphics Forum*, 14(4):203–215, 1995.

[Jones96] M. W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, December 1996.

[Levoy88] M. Levoy. Display of surface from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Lewis89] J. P. Lewis. Algorithms for solid noise synthesis. In *Computer Graphics*, volume 23(3), pages 263–270, 1989.

[Leyma92] F. Leymarie and M. D. Levine. Fast rater scan distance propagation on the discrete rectangular lattice. *CVGIP: Image Understanding*, 55(1):84–94, January 1992.

[Mulli92] J. C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing*, 54(6):526–535, 1992.

[Peach85] D. R. Peachey. Solid texturing of complex surfaces. In *Computer Graphics*, volume 19(3), pages 279–286, 1985.

[Perli85] K. Perlin. An image synthesizer. In *Computer Gaphics*, volume 19(3), pages 287–296, 1985.

[Perli89] K. Perlin and E. Hoffert. Hypertexture. In *Computer Graphics*, volume 23(3), pages 253–262, 1989.

[Rosen66] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13(4):471–494, 1966.

[Srame98] M. Sramek and A. Kaufman. Object voxelization by filtering. In *IEEE Symposium on Volume Visualization*, pages 111–118, 1998.

[Srame99] M. Sramek and A. Kaufman. `vxt`: a c++ class library for object voxelization. In *International Workshop on Volume Graphics*, pages 295–306, 1999.