Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

Distributed rendering of interactive soft shadows

M. Isard†, M. Shand and A. Heirich

Compaq Computer Corporation

Abstract

Recently several distributed rendering systems have been developed which exploit a cluster of commodity computers by connecting host graphics cards over a fast network to form a compositing pipeline. This paper introduces a new algorithm which takes advantage of the programmable compositing operators in these systems to improve the performance of rendering multiple shadow-maps, for example to produce approximate soft shadows. With an nVidia GeForce4 Ti graphics card the new algorithm reduces the number of required render nodes by nearly a factor of four compared with a naive approach. We show results that yield interactive-speed rendering of 32 shadows on a 9-node Sepia2a distributed rendering cluster.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: 3-Dimensional Graphics and Realism

1. Introduction

There is an extensive literature on rendering shadows at interactive rates. The two most popular methods are shadow volumes[1] and shadow maps[12] both of which can be implemented partially or completely in the current generation of programmable commodity graphics cards, such as the nVidia GeForce4 and the ATI Radeon 8500. Shadow volumes can be used to cast accurate hard shadows without aliasing artifacts, but there is some extra cost in preprocessing the geometry[3] and if the scene is made up of many small objects, for example the leaves of a plant, performance can become limited by stencil-buffer fill rate. It is also difficult to modify shadow volume techniques to deal with effects such as hair and fog[6].

Shadow maps remove some of the limitations of shadow volumes at the cost of introducing aliasing artifacts due to mismatched projection resolutions in the shadow map and the eye view[12, 6]. Shadow maps become inefficient when rendering multiple shadows however, since $2L$ rendering passes are needed to render a scene lit by L point sources (L passes to render the shadow maps, and L passes feeding into an accumulation buffer to composite the illumination information from each light in turn). Hardware-accelerated shadow mapping also consumes at least one texture unit which is a scarce resource in current graphics cards.

Several distributed rendering systems have recently been developed which exploit a cluster of commodity computers, each with a host graphics card and linked by a fast network[5, 11, 8]. Typically these systems can be programmed with a range of simple compositing operators which combine each locally rendered pixel with a remote pixel from one or more nodes, and output the transformed result to subsequent nodes in the rendering pipeline. The available compositing operators usually include for example depth compare, alpha blending and antialiasing. A given application programs the nodes under its control with an appropriate set of compositing operators to create a rendering pipeline that meets that application's overall rendering needs.

This paper presents an algorithm which uses a custom compositing operator to render a shadow mapped scene on a distributed rendering cluster. Illumination by L point sources can be rendered by $(L/K)+1$ nodes where K is the number of texture units on each graphics card. For walkthrough applications each node requires a single rendering pass, while for scenes with moving lights or geometry $K+1$ passes are needed per node. In addition all K texture units are avail-

† Now at Microsoft Research

able for rendering material properties allowing a full range of multi-texture material effects.

2. Shadow mapping

Shadow maps were proposed by Lance Williams in 1978[12] and have become increasingly popular for interactive shadows as hardware support for rendering them has become prevalent[10, 4]. We will briefly describe how shadow maps are implemented when depth textures are available, for example using the `ARB_shadow` OpenGL extension. Shadow maps can also be implemented somewhat less efficiently without depth textures, for example using the ATI Radeon 8500, and there is a brief discussion of the tradeoffs involved in section 4.2.

Following standard notation[10] we introduce three homogeneous coordinate systems; *clip coordinates*, *world coordinates* and *light coordinates*. The scene's geometry is expressed in world coordinates $\mathbf{x} = (x,y,z,w)^T$. The geometry can be projected into the eye's viewpoint using the projective transformation matrix F^cM^c to give clip coordinates $\mathbf{x}^c = (x^c,y^c,z^c,w^c)^T = F^cM^c\mathbf{x}$. Similarly the projective transformation F^lM^l converts world coordinates to light coordinates $\mathbf{x}^l = (x^l,y^l,z^l,w^l)^T = F^lM^l\mathbf{x}$. Following OpenGL coordinate-transformation conventions, M^c and M^l are typically rigid-body transformations effecting scale and rotation and F^c and F^l are projections to a clip frustum.

A scene illuminated by a single point light source is rendered in two passes. First the scene is rendered from the viewpoint of the light source and the resulting image (the shadow map) is stored in a depth texture where $T(u,v)$ is the depth value stored at coordinate (u,v). The shadow map is used to store the scene depth projected into light-view coordinates, so

$$T(\frac{x^l}{w^l},\frac{y^l}{w^l}) = \frac{z^l}{w^l}.$$

Next the scene is rendered again from the eye's viewpoint with the shadow map bound to one of the graphics card's texture units. During polygon rasterisation the texture coordinates $\mathbf{x}^t = (x^t,y^t,z^t,w^t)$ at pixel $\mathbf{x}^c$ are generated using the texture matrix $F^lM^l(F^cM^c)^{-1}$ and thus transformed to light coordinate space. At each pixel the texture hardware is used to read a value

$$z^m = T(\frac{x^t}{w^t},\frac{y^t}{w^t})$$

from the shadow map and a depth comparison is done; if $z^t/w^t < z^m + \varepsilon$ then the pixel is considered to be illuminated, otherwise the pixel is in shadow. The depth bias ε is included to reduce self-shadowing artifacts and should be chosen based on the geometry of the model being used[9].

Percentage closer filtering[9] can be achieved by enabling linear filtering on the texture map using e.g. the `ARB_depth_texture` OpenGL extension, resulting in a per-pixel illumination value $s \in [0,1]$ which varies from $s = 0$ for points completely in shadow to $s = 1$ for points which are completely illuminated. Programmable fragment shaders available on high-end commodity graphics cards can then be used to set the final pixel value $\mathbf{p}^f = s.\mathbf{p}$ where $\mathbf{p}$ is the fully-illuminated pixel colour.

When $K > 1$ texture units are available on a graphics card it seems attractive to render multiple shadows in one pass by generating K shadow maps from the viewpoints of K different lights and binding each map to a different texture unit. It is then straightforward to compute s_k, the illumination coefficient for each light, however computing the final pixel value $\mathbf{p}^f$ is problematic since the rendering pipeline has already summed the lighting contributions from the K lights into a single pixel colour $\mathbf{p}$. This difficulty is overcome by decomposing the final pixel colour into illumination and material properties as described in the following section.

3. A lighting compositing operator

In recent years several programmable distributed rendering frameworks have been developed, including Sepia2[5], Lightning-2[11] and PixelFlow[8]. Rendering nodes are arranged in a linear pipeline and each node generates an image of pixel values which are computed as a function of a locally rendered image and the output of the preceding render node in the pipeline. This function is known as a *compositing operator* and can be programmed in an application-specific manner. We have taken advantage of this programmability to design a new compositing operator suitable for the distributed rendering of global-illumination properties such as shadows.

Typically the locally rendered image is captured by the card using a DVI interface and each pixel consists of a 24-bit (r,g,b) triplet. The data layout of the compositing function has more flexibility and in general a network pixel can contain more than 24 bits of data to be interpreted in an application-specific manner. We use this flexibility to decompose the pixel colour information into illumination-dependent and material-dependent channels which can then be recombined in a final compositing step as described in the following paragraphs; implementations using OpenGL on specific commodity graphics cards are given in section 4.

A typical pixel lighting equation, taken from the OpenGL specification, gives the final colour $\mathbf{p}^f$ of a pixel illuminated by L lights as

$$\mathbf{p}^f = \mathbf{e} + \mathbf{m}^a \times \mathbf{c}^a + \sum_{\lambda=1}^{L} \left((\mathbf{m}^a \times \mathbf{c}_\lambda^a) i_\lambda^a + (\mathbf{m}^d \times \mathbf{c}_\lambda^d) i_\lambda^d + (\mathbf{m}^s \times \mathbf{c}_\lambda^s) i_\lambda^s \right). \quad (1)$$

Here $\mathbf{e}$ is the light emitted by the material, $\mathbf{m}^a$, $\mathbf{m}^d$ and $\mathbf{m}^s$ are the ambient, diffuse and specular material colours respectively and $\mathbf{c}^a$ is the global ambient illumination.

Each light λ has ambient, diffuse and specular illumination colours $\mathbf{c}^a_\lambda$, $\mathbf{c}^d_\lambda$ and $\mathbf{c}^s_\lambda$ respectively, and i^a_λ, i^d_λ and i^s_λ are ambient, diffuse and specular attenuation coefficients which depend on per-pixel lighting parameters such as the location and orientation of the illuminated object, spotlighting, fog, etc. In this notation a bold-face variable $\mathbf{u}$ refers to a colour vector (u_r, u_g, u_b) and $\mathbf{u} \times \mathbf{v}$ denotes component multiplication $(u_r v_r, u_g v_g, u_b v_b)$.

The lighting equation can be modified to include shadowing effects by including shadowing coefficients s_λ as follows:

$$\mathbf{p}^f = \mathbf{e} + \mathbf{m}^a \times \mathbf{c}^a + \sum_{\lambda=1}^{L} \left((\mathbf{m}^a \times \mathbf{c}^a_\lambda) i^a_\lambda s_\lambda + (\mathbf{m}^d \times \mathbf{c}^d_\lambda) i^d_\lambda s_\lambda + (\mathbf{m}^s \times \mathbf{c}^s_\lambda) i^s_\lambda s_\lambda \right)$$

which can be rewritten as

$$\mathbf{p}^f = \mathbf{e} + \mathbf{m}^a \times (\mathbf{c}^a + \mathcal{I}^a) + \mathbf{m}^d \times \mathcal{I}^d + \mathbf{m}^s \times \mathcal{I}^s \quad (2)$$

where

$$\mathcal{I}^a = \sum_{\lambda=1}^{L} \mathbf{c}^a_\lambda i^a_\lambda s_\lambda, \;\; \mathcal{I}^d = \sum_{\lambda=1}^{L} \mathbf{c}^d_\lambda i^d_\lambda s_\lambda, \;\; \mathcal{I}^s = \sum_{\lambda=1}^{L} \mathbf{c}^s_\lambda i^s_\lambda s_\lambda.$$

Since $\mathcal{I}^a$, $\mathcal{I}^d$ and $\mathcal{I}^s$ do not depend on $\mathbf{m}^a$, $\mathbf{m}^d$ or $\mathbf{m}^s$ this suggests a strategy for partitioning the compositing pipeline into *illumination nodes* which take account of lighting parameters and shadowing and *material nodes* which are programmed with the material properties of the scene objects. Given a pipeline in which each render node has K active textures, we will assign N nodes to be illumination nodes allowing NK distinct light sources. We make the somewhat limiting assumption that the lights can be partitioned into N subsets $(\mathcal{L}_n)_{n=1}^N$ each of size K such that all the lights in a given subset are the same colour, i.e.

$$(\mathbf{c}^a_\lambda = \mathbf{c}^a_n, \; \mathbf{c}^d_\lambda = \mathbf{c}^d_n, \; \mathbf{c}^s_\lambda = \mathbf{c}^s_n) \; \forall \, \lambda \in \mathcal{L}_n.$$

This assumption is reasonable for many scenes, in particular when soft shadows are being simulated by placing multiple point light sources at sample positions on an area light source.

The illumination node is then programmed so that the colour at each pixel is given by the triplet $(\mathcal{I}^a_n, \mathcal{I}^d_n, \mathcal{I}^s_n)$ where

$$\mathcal{I}^a_n = \sum_{\lambda \in \mathcal{L}_n} i^a_\lambda s_\lambda, \;\; \mathcal{I}^d_n = \sum_{\lambda \in \mathcal{L}_n} i^d_\lambda s_\lambda, \;\; \mathcal{I}^s_n = \sum_{\lambda \in \mathcal{L}_n} i^s_\lambda s_\lambda. \quad (3)$$

The compositing function at illumination node n computes three colours, $\mathbf{p}^a_n$, $\mathbf{p}^d_n$ and $\mathbf{p}^s_n$ where

$$\mathbf{p}^a_n = \mathbf{p}^a_{n-1} + \mathbf{c}^a_n \mathcal{I}^a_n, \;\; \mathbf{p}^d_n = \mathbf{p}^d_{n-1} + \mathbf{c}^d_n \mathcal{I}^d_n, \;\; \mathbf{p}^s_n = \mathbf{p}^s_{n-1} + \mathbf{c}^s_n \mathcal{I}^s_n$$

and $\mathbf{c}^a_n$, $\mathbf{c}^d_n$ and $\mathbf{c}^s_n$ are constants programmed into the compositing hardware on a per-frame or per-scene basis.

The output of node N, the final illumination node in the pipeline, is an image of 9-component pixels $(\mathbf{p}^a_N, \mathbf{p}^d_N, \mathbf{p}^s_N)$ which can be composited with the material colours of the scene in up to four material nodes at the end of the pipeline. The material nodes compute the following pixel colour triplets and compositing operations:

Specular material node $N+1$: $\mathbf{S} = (m^s_r, m^s_g, m^s_b)$

$$\mathbf{p}^a_{N+1} = \mathbf{p}^a_N, \;\; \mathbf{p}^d_{N+1} = \mathbf{p}^d_N, \;\; \mathbf{p}^s_{N+1} = \mathbf{p}^s_N \times \mathbf{S}$$

Diffuse material node $N+2$: $\mathbf{D} = (m^d_r, m^d_g, m^d_b)$

$$\mathbf{p}^a_{N+2} = \mathbf{p}^a_{N+1}, \;\; \mathbf{p}^{ds}_{N+2} = \mathbf{p}^d_{N+1} \times \mathbf{D} + \mathbf{p}^s_{N+1}$$

Ambient material node $N+3$: $\mathbf{A} = (m^a_r, m^a_g, m^a_b)$

$$\mathbf{p}^{ads}_{N+3} = \mathbf{p}^a_{N+2} \times (\mathbf{c}^a + \mathbf{A}) + \mathbf{p}^{ds}_{N+1}$$

Emissive material node $N+4$: $\mathbf{E} = (m^e_r, m^e_g, m^e_b)$

$$\mathbf{p}^f = \mathbf{E} + \mathbf{p}^{ads}_{N+3}$$

where $\mathbf{c}^a$ is a constant ambient lighting colour programmed into node $N+3$ on a per-frame or per-scene basis. Note that no shadow or lighting computations are done on any of the material nodes, so all texture units are available for rendering the material properties of the scene.

Obvious simplifications to the lighting model can be made which reduce the number of material nodes. For example, photorealistic rendering usually assumes no ambient lighting, which removes the need for node $N+3$. Scenes which do not include light-emitting materials can be rendered without node $N+4$. A final simplification can be made if the specular material colour $\mathbf{m}^s$ is the same for all objects. In this case (assuming no ambient or emissive lighting), the compositing function in the illumination nodes is modified to compute

$$\mathbf{p}^d_n = \mathbf{p}^d_{n-1} + \mathbf{c}^d_n \mathcal{I}^d_n, \;\; \mathbf{p}^s_n = \mathbf{p}^s_{n-1} + (\mathbf{m}^s \times \mathbf{c}^s_n) \mathcal{I}^s_n \quad (4)$$

and only a single material node is needed which computes

$$\mathbf{D} = (m^d_r, m^d_g, m^d_b), \;\; \mathbf{p}^f = \mathbf{p}^d_N \times \mathbf{D} + \mathbf{p}^s_N. \quad (5)$$

Our implementation in fact also provides limited support for global ambient illumination when $\mathbf{c}^a = \mathbf{c}^d_n$ for some n and either $\mathbf{m}^a = 0$ or $\mathbf{m}^a = \mathbf{m}^d$ for all materials in the scene. Illumination node n is then programmed to compute

$$\mathbf{p}^d_n = \mathbf{p}^d_{n-1} + \mathbf{c}^d_n (\mathcal{I}^d_n + \mathcal{I}^a), \;\; \mathbf{p}^s_n = \mathbf{p}^s_{n-1} + (\mathbf{m}^s \times \mathbf{c}^s_n) \mathcal{I}^s_n \quad (6)$$

where

$$\mathcal{I}^a = \begin{cases} 1 & \text{if } \mathbf{m}^a = \mathbf{m}^d \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

and we use this for example to simulate the light-emitting material in the bottom image of figure 3 (see color plates).

4. Implementation

We have implemented the lighting compositing operator (6) in the Sepia2a parallel rendering framework using nVidia

GeForce4 Ti 4600 cards. (At the time of writing a Sepia cluster populated with nVidia GeForce4 Ti 4600 cards is not yet available so results shown are based on a cycle accurate simulated execution of the compositing hardware designs.) We have also investigated the feasibility of using ATI Radeon 8500 cards and this is briefly discussed in section 4.2. Sepia2a is based on the Sepia2 distributed rendering architecture[5] but supports transmission of the local image from a graphics card directly to the Sepia PCI card using a DVI interface without passing through the host PCI bus. The natural size for a network pixel is 64 bits and illumination node n must compute $\mathbf{p}_n^d$ and $\mathbf{p}_n^s$, for which we allocate 11 and 10 bits per channel respectively leaving 1 bit unused.

Illumination node n renders K lights in $K+1$ passes. The first K passes are used to generate shadow-maps from the viewpoint of each light in turn, and details for the two graphics cards are given in sections 4.1 and 4.2 respectively. The final pass renders the image which will be sent to the Sepia card for compositing.

The illumination-node compositor computes (6) so the host graphics card must supply $(\mathcal{I}_n^d + \mathcal{I}^a)$ and $\mathcal{I}_n^s$ defined in (3) and (7). The K texture units can be programmed to generate the s_k, so it remains to generate i_k^d, i_k^s for each light along with $\mathcal{I}^a$ and combine the coefficients. Both target graphics cards contain programmable *fragment shader* stages which can be used for per-pixel lighting computations. In both cases the number of interpolator inputs is severely limited to a four-channel (r,g,b,a) primary colour, a three-channel (r,g,b) secondary colour, and K four-channel (r,g,b,a) texture values. Since we wish to use all of the texture units for shadowing we are constrained to place i_k^s, i_k^d and $\mathcal{I}^a$ in the primary and secondary colours which permits seven channels in all.

We limit ourselves to generating diffuse and specular illumination components for at most three unique lights, and place $(i_1^d, i_2^d, i_3^d, \mathcal{I}^a)$ in the primary colour and (i_1^s, i_2^s, i_3^s) in the secondary colour. If $K > 3$ we therefore enforce the restriction that the K lights must be partitioned into three subsets $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ such that the light positions $\mathbf{l}_k$ are clustered around centres $\mathbf{c}_1$, $\mathbf{c}_2$ and $\mathbf{c}_3$ and

$$\mathbf{l}_k \approx \mathbf{c}_i \; \forall \; k \in \mathcal{G}_i.$$

On our target cards K is at most 4 so in practice this restriction amounts to placing two of the four lights close together, which is reasonable for our target application of soft-shadowing which clusters many identical lights close together in any case. Since shadow boundaries have much higher spatial frequency than either diffuse or specular lighting variations it is still worthwhile to generate 4 shadows given only 3 lighting locations. It would be possible at the expense of abandoning support for simple ambient lighting to place i_4^s for a fourth light location in the alpha channel of the primary colour to account for the higher spatial variation of specular lighting compared with diffuse lighting.

It is straightforward to persuade OpenGL to place the desired information in the primary and secondary colour channels. All material diffuse and specular RGB values are set to $(1,1,1)$, while the other specular parameters such as shininess are set according to the desired material properties for each object. Materials for which $\mathbf{m}^a = \mathbf{m}^d$ have their alpha diffuse material colour set to 1, otherwise it is set to 0. Three lights are enabled at locations $\mathbf{c}_1$, $\mathbf{c}_2$ and $\mathbf{c}_3$ with diffuse and specular colours both set to $(1,0,0,0)$, $(0,1,0,0)$ and $(0,0,1,0)$ respectively and programmed with the desired parameters for attenuation, spotlighting, etc. Details of programming the fragment shaders are given in sections 4.1 and 4.2.

The material node images are all straightforward to generate. No special graphics card programming is required; the scene is rendered from the eye's viewpoint with lighting disabled and object colours set to the appropriate material colour $\mathbf{m^d}$ or $\mathbf{m^s}$.

4.1. nVidia GeForce3/4 Ti

We have implemented the illumination-node code on an nVidia GeForce4 Ti 4600 graphics card. The programming model is identical for other cards in the GeForce3 and GeForce4 Ti series. These cards support depth textures so generating shadow map k is straightforward. A texture map of the desired size is created with internal format `DEPTH_COMPONENT24_ARB` and the scene is rendered from viewpoint $\mathbf{l}_k$ with all lighting, texturing and colour buffers disabled. If available the `WGL_ARB_render_texture` extension can be used to render directly to the texture otherwise the image is rendered to the framebuffer and copied internally to the graphics card using `glCopyTexSubImage2D`.

Before rendering the scene from the eye's viewpoint, texture k is bound to texture unit k and all texture units are programmed to clamp to a border depth of 1.0, with linear filtering enabled. `GL_TEXTURE_COMPARE_MODE_ARB` is set to `GL_COMPARE_R_TO_TEXTURE` with `GL_TEXTURE_COMPARE_FUNC_ARB` set to `GL_LEQUAL`. Coordinate generation is enabled for all four texture coordinates in `GL_EYE_LINEAR` mode, and the (s,t,r,q) `GL_EYE_PLANE` values are respectively set to the four rows of the matrix SF^lM^l where

$$S = \begin{pmatrix} 0.5 & 0.0 & 0.0 & 0.5 \\ 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & 0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}.$$

Three general combiner stages are used, and the program is given in figure 1.

4.2. ATI Radeon 8500

We have investigated implementing illumination nodes using an ATI Radeon 8500 card which does not support depth

Stage			
0	$\text{spare0}'_{\text{rgb}}$	=	$\text{texture0}_{\text{rgb}} \times \text{const0}_{\text{rgb}} + \text{texture1}_{\text{rgb}} \times (1 - \text{const0}_{\text{rgb}})$
0	$\text{spare0}'_{\text{a}}$	=	$\text{texture2}_{\text{a}}.1 + \text{texture3}_{\text{a}}.1$
1	$\text{spare0}'_{\text{rgb}}$	=	$\text{spare0}_{\text{rgb}} \times (1 - \text{const1}_{\text{rgb}}) + \text{spare0}_{\text{a}} \times \text{const1}_{\text{rgb}}$
2	$\text{spare0}'_{\text{rgb}}$	=	$\text{spare0}_{\text{rgb}} \bullet \text{primary}_{\text{rgb}}$
2	$\text{spare1}'_{\text{rgb}}$	=	$\text{spare0}_{\text{rgb}} \bullet \text{secondary}_{\text{rgb}}$
Final	$\text{final}_{\text{rgb}}$	=	$\text{spare0}_{\text{rgb}} \times \text{const0}_{\text{rgb}} + \text{spare1}_{\text{rgb}} \times (1 - \text{const0}_{\text{rgb}}) + \text{primary}_{\text{alpha}} \times \text{const0}_{\text{rgb}}$

On entry

$$\text{const0} = (1,0,0,0) \quad \text{const1} = (0,0,1,0)$$
$$\text{primary} = (i_0^d, i_1^d, i_2^d, \mathcal{I}^a) \quad \text{secondary} = (i_0^s, i_1^s, i_2^s, 0) \quad \text{texture}k = (s_k, s_k, s_k, s_k)$$

On exit

$$\begin{aligned}\text{final} = (&i_0^d s_0 + i_1^d s_1 + i_2^d (s_2 + s_3) + \mathcal{I}^a,\\ &i_0^s s_0 + i_1^s s_1 + i_2^s (s_2 + s_3),\\ &i_0^s s_0 + i_1^s s_1 + i_2^s (s_2 + s_3))\end{aligned}$$

Figure 1: *The register combiner program for rendering four shadows on an nVidia GeForce4 Ti graphics card.*

textures. Although the card has $K = 6$ active textures, the depth comparison must be performed as part of the fragment shader program and so two texture coordinates must be assigned to each shadow map so at most three shadows can be rendered in a single pass. A 16-bit precision depth comparison can be implemented in a two-pass shader program as opposed to the 24-bit comparison performed by the depth texture functionality on the nVidia card. Unfortunately percentage-closer filtering is not possible with this implementation and so aliasing artifacts are much more visible. In addition the performance is significantly worse than the GeForce4 Ti 4600 implementation so all results are shown using the nVidia card.

5. Results

We tested the algorithm on a set of simple models using an nVidia GeForce4 Ti 4600 graphics card in a Compaq Evo D500 1.7 GHz P4 workstation running Redhat Linux 7.2. Figure 3 (see color plates) shows images as they would be rendered at 800×600 pixels on a 9-node Sepia2a cluster using 512×512-pixel textures for the shadow maps. As a Sepia2a cluster populated with GeForce4 Ti 4600 graphics cards was not available, the compositing is done in simulation. The local rendering code is run exactly as it will be in a full system, then the image is grabbed using `glReadPixels` and a software simulation of the compositing operators is run to generate the final image. Table 2 shows timings measured on our single node test setup. The Sepia architecture introduces a latency of approximately two frame refresh periods, while the number of frames rendered per second is approximately that of the slowest node in the pipeline. The nVidia driver we used for Linux does not support direct rendering to textures, though this is supported by the Windows drivers. We measure the time for the `glCopyTexSubImage2D` call to be 1.31 ms per computed shadow map.

6. Discussion

We demonstrate an algorithm which is able to render approximate soft shadows at interactive rates on a cluster of commodity computers linked by a Sepia2a compositing network. The number of lights scales linearly with the number of available nodes and increasing the number of rendering nodes results in a negligible reduction in performance. For walkthrough applications the new algorithm reduces the number of required rendering nodes by a ratio of $1 + \varepsilon : 4$ compared with a naive approach, where $\varepsilon \to 0$ as the number of lights increases. For scenes with changing geometry a naive approach renders one shadow per node using 2 rendering passes. The new algorithm must perform $K + 1$ rendering passes to render K shadows, so as long as the timing budget

Model\Lights per node	1	2	3	4
Balls (no shadows)	0.98	1.13	1.43	1.44
Balls (walkthrough)	1.11	1.26	1.55	1.57
Balls (moving lights)	3.31	5.65	8.25	10.71
Horse (no shadows)	1.17	1.30	1.45	1.60
Horse (walkthrough)	1.26	1.40	1.54	1.69
Horse (moving lights)	2.39	3.82	5.66	7.49
Plant (no shadows)	1.13	1.08	1.18	1.35
Plant (walkthrough)	1.24	1.21	1.29	1.47
Plant (moving lights)	3.47	5.64	7.98	10.70
Box (no shadows)	0.54	0.55	0.58	0.62
Box (walkthrough)	0.77	0.79	0.81	0.83
Box (moving lights)	2.04	3.53	5.31	6.98

Figure 2: *Rendering times in ms for the models in figure 3 (see color plates) which are shown rendered with 4 lights per node. Times marked "no shadows" correspond to rendering the scene in full colour with OpenGL lighting enabled. Times marked "walkthrough" correspond to rendering the scene with precomputed shadow maps with the fragment shader programmed as described in figure 1. Times marked "moving lights" are as for "walkthrough" but the shadow maps are recomputed at each frame. Timings are given for a single node and the Sepia architecture renders composited frames at approximately the speed of the slowest node in the pipeline. Note that for "walkthrough" scenes these rendering times are a small fraction of typical screen refresh rates and even the slowest "moving lights" rendering times correspond to refresh rates above 93 Hz.*

permits at least two shadow maps to be rendered per node the algorithm still decreases the number of required render nodes by a ratio of $1+\varepsilon : 2$, while the ratio of $1+\varepsilon : 4$ is achieved if the timing budget permits 5 shadow maps per illumination node.

The main limitation of the method is that it scales badly with increasing scene complexity as each node must render the full scene geometry. For walkthrough applications it would be straightforward to perform a sort-first decomposition of the scene so that each illumination node renders only a subset of the visible geometry. It is well known[7] however that sort-last scene decompositions allow better load balancing of distributed rendering than sort-first methods. A sort-last Z-compositing approach is feasible using the algorithm presented here at the expense of transferring more data in each network pixel. Illumination nodes would have to transmit not only the 63-bit diffuse and specular components $(\mathbf{p}_n^d, \mathbf{p}_n^s)$ but also (r,g,b,z) channels describing a partially rendered image which would typically be assigned 48 bits in total: 8 bits per channel for colour and 24 bits for the Z-buffer. Alternatively it would be possible to use a rendering architecture which supports a "join" operator taking network pixels from more than one preceding render node in the pipeline without increasing the maximum number of bits transmitted in a network pixel.

Minor updates to the algorithm would allow rendering shadows cast by pseudo-transparent objects using depth-peeling[2]. Unfortunately the number of rendering nodes used by such an approach is $O(D^2)$ where D is the number of depth-peeling layers, so this is probably not feasible for D much greater than 4.

References

1. F. Crow. Shadow algorithms for computer graphics. In *Proceedings of SIGGRAPH*, pages 242–248, 1977.
2. C. Everitt. Order independent transparency. Technical report, nVidia, `developer.nvidia.com`.
3. C. Everitt and M.J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Technical report, nVidia, `developer.nvidia.com`.
4. C. Everitt, A. Rege, and C. Cebenoyan. Hardware shadow mapping. Technical report, nVidia, `developer.nvidia.com`.
5. A. Heirich and L. Moll. Scalable distributed visualization using off-the-shelf components. In *IEEE Parallel Visualization and Graphics Symposium*, pages 55–60, 1999.
6. T. Lokovic and E. Veach. Deep shadows. In *Proceedings of SIGGRAPH*, pages 385–392, 2000.
7. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.
8. S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using pixel composition. *Computer Graphics*, 26(2):231–240, 1992.
9. W.T. Reeves, D.H. Salesin, and R.L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of SIGGRAPH*, pages 283–291, 1987.
10. M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics*, 26(2):249–252, 1992.
11. G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lighting-2: A high-performance display subsystem for PC clusters. In *Proceedings of SIGGRAPH*, pages 141–148, 2001.
12. L. Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH*, pages 270–274, 1978.

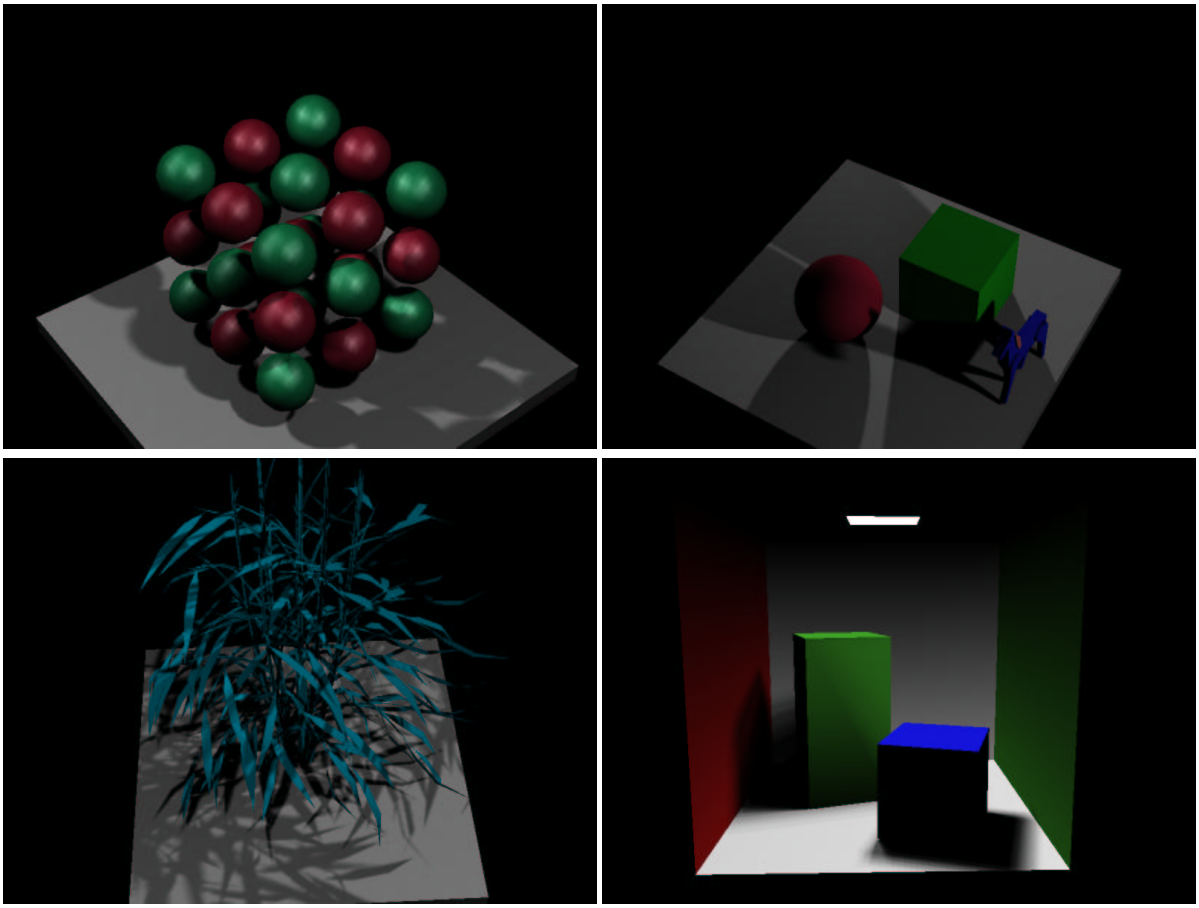

Figure 3: *Various simple models rendered with 32 point light sources. The first three images approximate two area lights with 16 samples each and the bottom right image approximates a single area light with 32 samples.*