Emulating an Offline Renderer by 3D Graphics Hardware

Jörn Loviscach
Hochschule Bremen
Flughafenallee 10
28199 Bremen, Germany
jlovisca@informatik.hs-bremen.de

ABSTRACT

3D design software has since long employed graphics chips for low-quality real-time previewing. But their dramatically increased computing power now paves the way to accelerate the final high-quality rendering, too. While as yet only one major 3D software package offers a dedicated "hardware renderer" for final output, a small number of design applications can leverage the graphics card to support game creation: They display vertex and pixel shader effects in the same way as they appear on the game's screen. We present an approach unifying the use of graphics cards for game design and for final rendering. It is implemented as a plug-in for Maxon Cinema 4D, a standard commercial software package for modelling, animation, and rendering. We examine which factors determine the performance of this solution and discuss corresponding improvements.

Keywords

vertex shader, pixel shader, Cg high level shader language, .fx shader files, shadow maps

1 INTRODUCTION

Most 3D software solutions for constructional or graphics design contain two renderers: one hardware-accelerated renderer delivering a rough but interactive preview and another renderer that only works offline, but produces high-quality images with complex lighting, shading, and texturing effects, usually including shadows, reflections, and refractions. The main drawbacks of this approach are:

- Much guesswork is involved as only the slow offline renderer shows the actual result.
- The high-quality renderer does not make any use of the computing power of the graphics card.

In some of the major 3D packages, the first problem is addressed by an interactive preview renderer: In a first pass, the usual final renderer is started, but for each pixel the material, the texture coordinates, and the lighting situation are stored. After this first pass, changes in materials can be quickly rendered with help of the intermediate data. Such an interactive preview renderer fails, however, if the geometry of the scene is changed. Furthermore, typically it does neither leverage the computing power of the graphics card nor offer a speedup for the final rendering.

Modern graphics cards possess a much-improved functionality that may allow a unified approach to rendering, offering high speed and high quality at the same time. The main new building blocks introduced to graphic cards in the recent years are vertex shaders [Lin01] and pixel shaders. Through them, 3D graphics chips have evolved from configurable to programmable processing units. Vertex shaders are small programs run for each vertex on the graphics card, pixel shaders are run for each pixel (or pixel fragment). Besides being programmable, modern graphics cards apply many different textures in one pass; they offer more than hundred megabytes of on-board memory for very detailed textures and for off-screen buffers up to 2048×2048 pixels, such as needed for depth maps. Full-scene antialiasing adds to the cleanliness of the result.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee, provided that no copies are made or distributed for profit or commercial advantage and that all copies bear this notice and the full citation on the first page. To otherwise copy or republish, to post on servers or to redistribute to lists, a prior specific permission and/or a fee are required.

Journal of WSCG, Vol. 12, No. 1–3, ISSN 1213-6972
WSCG '2004, February 2–6, 2004, Plzen, Czech Republic.
Copyright UNION Agency — Science Press

Vertex shaders and especially pixel shaders allow to render complex-looking materials in real time. Even for simple materials they easily surpass the quality of the typical interactive presentation used in 3D design software: In the usual previews, the colors themselves are interpolated across the faces of a mesh (Gouraud interpolation), which results in faceted highlights. Pixel shaders, however, allow to interpolate the normal vectors (Phong interpolation), thus producing specular highlights of the same quality as most offline renderers.

The objective of this work is to implement a hardware-based renderer which works at near-interactive speed and emulates the offline renderer of a standard commercial 3D graphics design software. Furthermore, we want to study the limits to such an approach, as imposed by the graphics hardware and the structure of the host 3D software. The presented solution is implemented as a plug-in for the software package Maxon Cinema 4D [Max03]. This software features one of the fastest commercial raytracing renderers; it has much code in common with Cebas finalRender for discreet$^{(R)}$ 3ds max^{TM}. This is interesting for comparisons. In addition, no “official” real-time shader plug-in has been published for Cinema 4D up to now.

Our plug-in called C4Dfx combines real-time shader effects given as industry-standard .fx (“effects”) files with an emulation of standard lighting, shading, texturing, and shadowing features of the host 3-D software, all rendered with help of the graphics card either offline or concurrently at near-interactive speed in a separate window or on an additional display, see Fig. 1.

Section 2 introduces related methods and solutions developed for high-quality rendering accelerated by standard 3D graphics hardware. Our implementation is described in section 3. Section 4 investigates the factors determining speed and quality. Design issues following from these observations are discussed in section 5. In section 6 we summarize the results and outline future developments.

2 RELATED WORK

Up to now, among the major 3D design software packages only Alias$^{(R)}$ Maya$^{(R)}$ 5.0 [Ali03] offers a dedicated “hardware renderer” that makes use of the computing power of the graphics card. This hardware renderer requires a workstation-class graphics accelerator; but according to our experiments, even with some of such expensive graphics cards it does not support features such as shadows. Furthermore, this hardware renderer is not intended to be used interactively, but acts only as a speedy replacement of the usual software renderer.

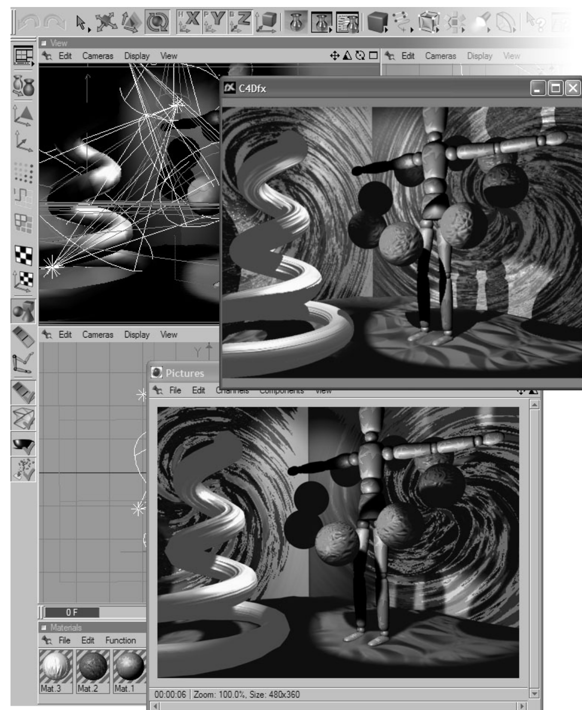

Figure 1. The hardware-accelerated renderer can work at near-interactive speed, continuously displaying its result in an own window (upper right). The usual Cinema 4D preview is visible in the top left view port; the lower window shows the result of its built-in offline raytracer.

The absence of a hardware renderer does not mean that typical 3D design software does not support vertex and pixel shaders. Rather, in several such software packages, real-time shaders can be used in the *interactive* rendering. This is intended as a preview to assist game designers, who thus are able to build models and materials with perfect visual feedback.

To describe a material including pixel shaders, vertex shaders, texture file references, and adjustable parameters, Microsoft$^{(R)}$ and NVIDIA$^{(R)}$ have established the .fx file format [Aro03]. 3D design software and game engines can share such files in order to offer identical materials. Microsoft$^{(R)}$’s programming interface DirectX$^{(R)}$ allows to load such files; NVIDIA$^{(R)}$ offers the CgFX toolkit to use .fx materials with both DirectX$^{(R)}$ and OpenGL$^{(R)}$.

To extend Alias$^{(R)}$ Maya$^{(R)}$ 4.5/5.0 [Ali03] and discreet$^{(R)}$ 3ds max^{TM} 5.1 [dis03] by a real-time shader display based on .fx files, NVIDIA$^{(R)}$ has published corresponding plug-ins. Softimage$^{(R)}$ offers an own .fx support for its package XSI$^{(R)}$ 3.5.

While .fx files may contain shaders written as assembler code for the 3D chip, the trend is to use a high level shading language such as Microsoft$^{(R)}$'s HLSL (High Level Shading Language, a part of DirectX$^{(R)}$ 9.0), or NVIDIA$^{(R)}$'s Cg ("C for Graphics") [Mar02]. Both languages are virtually identical. An OpenGL$^{(R)}$ Shading Language has been introduced recently as an extension of OpenGL$^{(R)}$ 1.5. Currently, it is not certain if this shading language may in future be used in .fx files, too.

NVIDIA$^{(R)}$ offers a set of scripts along with the plug-in for Maya$^{(R)}$. Among them is "Conv", which attempts to generate .fx and texture files to emulate the standard materials of a 3D scene. This approach allows, for instance, to reuse the generated files in a game engine. However, neither does it work fully transparently as a substitute for the final renderer, nor does it, for instance, produce shadows.

Today's commercial applications mainly use graphics hardware to accelerate classic lighting and shading models such as Phong's. Current research, however, is concerned with using consumer-class 3D graphics cards to render complex shading (such as characterized by bidirectional reflection distribution functions, BRDFs) and global illumination effects in real time. For an overview, see [Kau03].

Another approach to harness the computing power of modern graphics chips for high-quality rendering is to use the 3D chip for raytracing. Though it has not been designed for this task, the chip can compute ray-object intersections by clever use of textures and shaders [Pur02].

3 IMPLEMENTATION

The current .fx shader plug-ins for 3D design software (see section 2) typically use the interactive view ports of the software for display. For our plug-in for Maxon Cinema 4D under Microsoft$^{(R)}$ Windows$^{(R)}$, we elected to open a new, dedicated window to display the results. This is due to the following reasons:

- The plug-in has full control over the window. This is indispensable for instance to let the user switch on the fly between different levels of full-scene antialiasing, an option not available in other solutions.
- The window is operated by a dedicated processing thread. The hardware-accelerated renderer works concurrently to Cinema 4D itself, so that the user needs not wait for the renderer to finish. This means however, that the renderer must receive a cloned copy of the 3D scene, because the user might edit the scene inside Cinema 4D before the rendering is completed.
- The separate window may be dragged to a secondary display unit, maybe enlarged to full-screen.

In addition to watching this near-interactive rendering, the user may start a hardware-accelerated offline renderer. It uses the same routines, but works in an off-screen buffer, does not process a single frame but an animation sequence and writes its result into an .avi file. This offline renderer works concurrently to both Cinema 4D itself and the hardware-accelerated near-interactive rendering window. For feedback to the user, the offline renderer opens a window in which it displays a thumbnail version of every frame it writes to disk.

The plug-in uses OpenGL$^{(R)}$ as programming interface for the graphics card. To parse and compile .fx files and set the graphics card accordingly, it employs NVIDIA$^{(R)}$'s CgFX toolkit.

The plug-in processes two different kinds of material definitions: standard Cinema 4D surface materials and shaders loaded from .fx files. The user-accessible parameters defined in an external .fx file are made accessible via usual GUI elements of Cinema 4D, see Fig. 2. Standard surface materials, on the other hand, are transparently converted to internal .fx shaders emulating Cinema 4D's offline raytracer. The .fx shaders, internal or file-based, are handed to the CgFX toolkit before rendering.

The design objective for shadows was: All objects can cast shadows; the objects carrying the Cinema 4D standard material can also receive shadows. For objects with an external .fx material, however, it would not be feasible to receive shadows, because the lighting model used in the .fx shader is not known: A matte surface should be influenced by a shadow, a mirror-like surface should not.

We elected not to add the contribution of each light source as another rendering pass (as done by the Conv script for the Maya$^{(R)}$ plug-in) for the following reasons: First, to start an additional pass causes a fixed overhead and demands to repeat basic computations such as the normalization of the normal vectors. Second, the passes are composed in 24 bit RGB frame memory, thus accumulating roundoff errors. (The current graphics cards optionally offer floating point precision buffers, but then the functionality is strongly limited.)

However, the pixel shader of each rendering pass can access only a certain number of textures and parame-

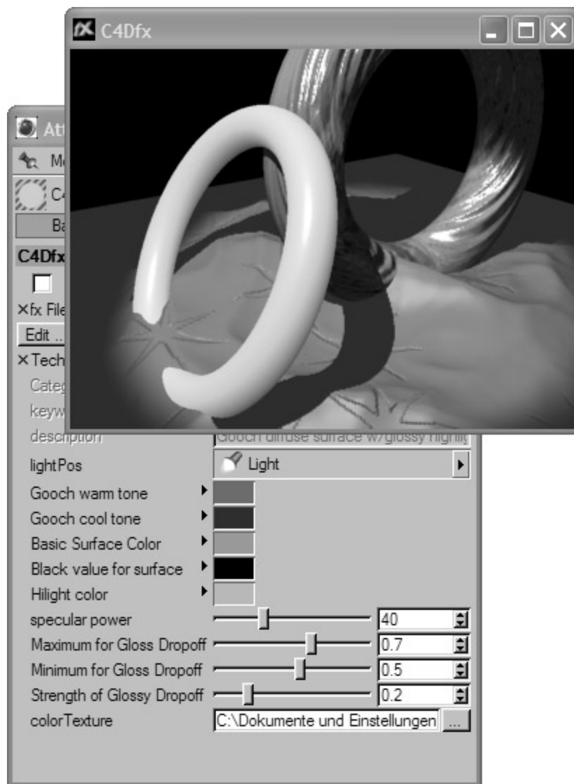

Figure 2. Shaders loaded from .fx files (on the left torus) and the emulated standard materials (other objects) are displayed in the same scene. For the parameters defined in .fx files, a graphical user interface is built on the fly. (Example .fx file from NVIDIA(R))

ters. Current graphics cards are limited to 16 simultaneous textures; each shadow-generating light source occupies one of them as a depth map. An even more stringent restriction is due to the current version 0.0.0.4 of the CgFX toolkit: It does not allow to use more than eight textures inside a pixel shader. Together with speed considerations, this lead to the decision to let the user optionally add a tag to each light source specifying whether or not it generates illumination and shadows in the hardware-accelerated renderer.

The plug-in collects the information about the light sources and builds for each standard Cinema 4D material a corresponding .fx shader in memory. Each of these internal shaders and each external .fx file is loaded into an own FXEffect object of the CgFX toolkit. When rendering an .avi file, this step happens only at the start, to save time. But because nearly every parameter in Cinema 4D can be animated, all of the steps described in the following are executed once per frame. In particular, each parameter of every FXEffect object is set according to its current value in the current frame of the Cinema 4D animation.

Next, depth maps are rendered into off-screen buffers for all light sources that cast shadows. In the plug-in, this is currently limited to round spot lights. In addition, displacement maps in the standard Cinema 4D material as well as vertex shader deformation are ignored for the shadow maps. These may be available in a later version of the plug-in. With shadow maps [Eve02b] this can be achieved quite easily. To implement this dynamic displacement with shadow volumes [Eve02a], however, would require to determine shadow silhouettes on the fly. This is rather difficult for objects already carrying vertex shaders from .fx files. The most promising way to compute shadow silhouettes in such a situation would be to render the vertex positions encoded as color values and read those back from the graphics card [Bra03].

Image files or 2D procedural shaders applied as diffuse color map, bump map, and environment map are read into memory arrays. To reduce artifacts due to insufficient resolution, Cinema 4D's spherical environment map is converted to a cube map. The bump map (containing height values) is converted to a normal map (containing normal vectors), which is the usual form to generate bump mapping on graphics hardware. Finally, the texture maps are handed on to OpenGL(R).

To render the final image frame, the plug-in invokes the corresponding FXEffect for each 3D object of the scene. The positions and texture coordinates of each object are sent to OpenGL(R) as an indexed vertex array. Already by itself, Cinema 4D tessellates all scene geometry such as parametric objects and spline surfaces ready to let them be used in vertex arrays.

Typical .fx shaders also require the values of the components of an orthogonal coordinate frame at each vertex: normal, tangent, and binormal vector. Per vertex, the bump map computation for internal .fx shaders needs the normal vector and vectors pointing in the u and v directions of the texture. To compute these data, the plug-in employs the adjacency data provided by Cinema 4D to first determine normal vectors per face and then, from those, normal vectors per vertex. From a linear approximation of the surrounding vertices, the u and v directions of the texture are determined. The u direction is used as tangent vector, too. The binormal vector is formed as the vector product of normal and tangent.

In total, the plug-in requires a graphics card capable of vertex shaders and pixel shaders of version "2.0 Extended" and the following OpenGL(R) extensions: WGL_ARB_pixel_format, WGL_ARB_pbuffer, GL_ARB_multisample, GL_ARB_depth_texture, WGL_ARB_render_texture,

and WGL_NV_render_depth_texture.

4 PERFORMANCE

To evaluate the speed and the quality of the solution, we measured the scaling behavior with different output resolutions, different amounts of geometry, etc. As a reference for an industry-standard offline renderer, we invoked the Cinema 4D raytracer for the same test scenes with the same settings. For brevity, we report here only the timing figures for the one test scene shown in Fig. 1. It comprises approximately 17.000 polygons, mostly quadrangles, and four spot light sources with shadows. The scene contains four materials with specular highlights and mixed use of diffuse color textures, bump maps, and environment maps. The benchmark system was an PC with a 2.5 GHz Intel(R) Pentium(R) 4 processor and an NVIDIA(R) GeForce™ FX 5900 graphics card running under Microsoft(R) Windows(R) XP.

Because Maxon released a code snippet illustrating the computation of specular highlights in Cinema 4D, we were able to emulate the lighting computations perfectly. In addition, 2D procedural shaders used for instance as diffuse textures or as bump map are reproduced nearly exactly: These are converted to bitmap textures, the precision of the emulation only being limited by the bitmap's resolution.

The plug-in uses shadow maps as offered by current graphics cards. This simple interpolation cannot fully reproduce the virtually aliasing-free shadows of Cinema 4D's own depth map algorithm. Other slightly visible differences occur with bump mapping: The exact computation that Cinema 4D uses here is not publicly accessible up to now.

Concerning rendering speed, the hardware-based solution may outperform Cinema 4D's fast software raytracer only when rendering a sequence of frames. As the leftmost data point of Fig. 3 shows, the initial expense to build shaders and off-screen buffers is quite high. While the raytracer needs 5.5 seconds per frame without a noticeable startup time, the plug-in completes each frame in 2.4 seconds—but only after a 2.8 seconds startup phase. A major part of this is used to generate and compile the program code of the .fx shaders. According to our measurements, this takes approximately 0.4 seconds per shader.

To test the behavior with varying scene complexity, we rendered a scene with the same setup but containing only one single quadrangle, one scene where the geometry was duplicated and slightly translated, and one scene with quadruplicated geometry, see Fig. 4. This benchmark revealed that Cinema 4D uses strong

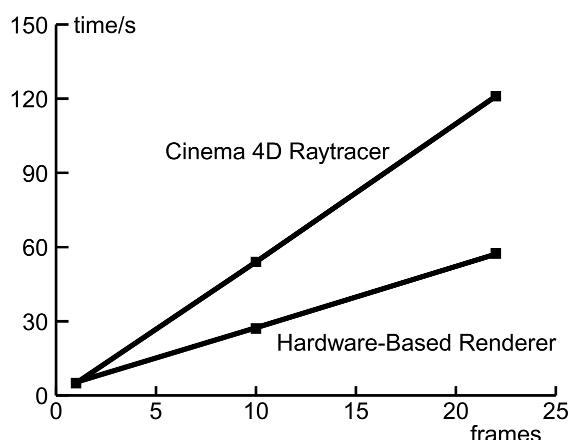

Figure 3. In contrast to the hardware-based renderer the standard Cinema 4D raytracer does not need a fixed startup time for a frame sequence. (No antialiasing, output: 640×480 **pixels, texture and depth map resolution:** 256×256 **texels)**

optimization techniques to better cope with complex scenes. (The curve reminds one of the $O(\log N)$ behavior of typical optimized raytracers.) Thus, for the settings used here, a scene of approximately 70.000 polygons yields a tie between the raytracer and the hardware-based solution.

The relationship between the dimensions of the resulting images and the rendering time is again linear for both solutions, but contains an initial constant cost for the hardware shaders, see Fig. 5. While the Cinema 4D raytracer delivers approximately 54.000 pixels per second, the graphics card achieves about 560.000 pixels per second—once the first pixel has been drawn. This also leads to a low impact of multisampling for the hardware-based solution.

Note that the timing figures in Fig. 5 are for 22 frames, so that the constant cost of 41 seconds for the hardware solution does not so much represent the setup of the shaders at the beginning, but (in decreasing order of importance) the filling of textures maps, the tessellation of the scene and construction of vertex arrays, the setting of parameters in the FXEffect objects and the validation of the shaders through CgFX.

Another contribution to the constant cost stems from the computation of the four shadow maps used in the scene. Both Cinema 4D's own renderer and the hardware-based solution take approximately 0.1 seconds to compute one shadow map for one frame. So in our context, the cost of preparing a shadow map is nearly negligible for both systems, at least for a map resolution of 256×256 pixels.

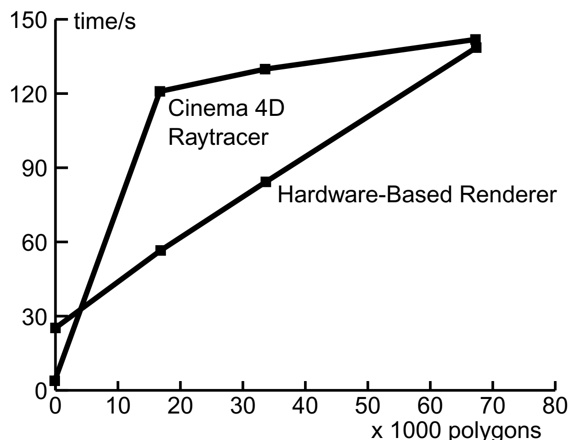

Figure 4. The standard Cinema 4D raytracer achieves a sub-linear dependency of the rendering time on the complexity of the scene. (22 frames, no antialiasing, output: 640×480 **pixels, texture and depth map resolution:** 256×256 **texels)**

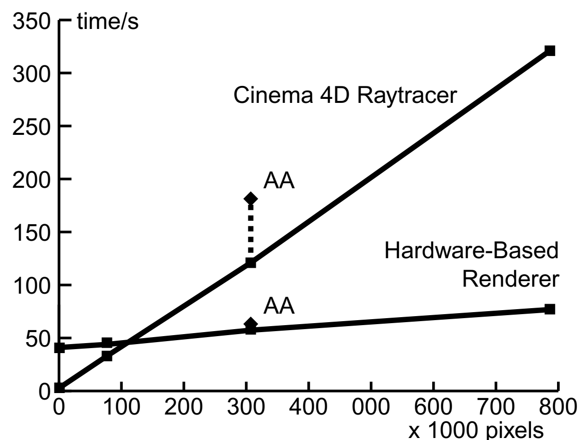

Figure 5. Once the textures and objects are set up, the hardware solution delivers high speed. (22 frames, texture and depth map resolution: 256×256 **texels, AA: four-sample multisampling for hardware, setting "antialiasing: geometry" for the raytracer)**

A design decision was not to use a single rendering pass for each light source. To estimate the cost of a multi-pass solution, we measured the rendering speed with those parts of vertex shader and pixel shader code left out that would have to repeated in each pass. This lead to a speedup of 0.43 seconds per frame. Thus, one can safely estimate that each additional pass would require more than 0.4 seconds render time, because the setup needed for each pass would even add to that number.

Cinema 4D's computation of specular highlights includes several transcendental functions:

$$a\left(1 - \text{clamp}(b\,\text{acos}(\mathbf{v}\cdot\mathbf{r}) - c)^d\right)^e,$$

where a to e are parameters determined from user input, and the unit vectors $\mathbf{v}$ and $\mathbf{r}$ point to the viewer and along the reflected light ray, respectively. To check which impact the large quantity of transcendental functions involved here has on rendering speed, we left out the arc cosine. This resulted in a speedup of 0.07 seconds per frame and light source. In fact, the arc cosine is the most expensive function here: To cancel the whole computation yielded 0.18 seconds speedup per frame and light source.

5 DISCUSSION

The plug-in can use Cinema 4D scene "as is", without any specific changes. External .fx files can seamlessly be used as materials, too. In rendering quality, the hardware-accelerated solution can largely keep up with the software raytracer, at least when no real reflections (but only environment maps) and no transparency or even refraction effects are asked for. These are missing up to now in the plug-in, because they would not only require extreme programming effort but also lead to poor rendering times.

As it turned out, hardware acceleration does not guarantee shorter render times. Our solution excels when producing movies in large, antialiased frame formats, When used as an interactive display, it has to handle, however, single frames of relatively small dimensions.

There is hope for faster pixel shaders in the next generation of graphics cards. Also, a later version of the CgFX toolkit may not only support 32 bit floating point computations but also NVIDIA(R)'s faster 16 bit floating point types as well as compiler profiles to fully employ the current generation of graphics cards. In the meantime, a feasible way to speed up computations involving transcendental functions (such as Cinema 4D's routine for specular highlights) would be to cast them into textures used as look-up tables.

However, what is needed more than raw computational power is clever optimization. This becomes obvious in the benchmarks of the hardware-accelerated solution against Cinema 4D's built-in raytracer. Clearly, the $O(N)$ behavior of a brute-force z-buffer renderer cannot beat an optimized raytracer. A natural idea is to use the occlusion culling offered by current graphics cards. A further improvement would be not to build .fx shaders (including textures etc.) for materials only used on completely occluded objects—or even never used.

Much could be achieved with a little more information delivered by the host application. In particular, textures that did not change since the last frame do not need to be rebuilt. The application could quite easily track which materials and which bitmaps and/or procedural textures used in materials change and which do not. Such a change can happen both through animation and through user input; it could be signaled to the plug-in via some kind of "dirty" flag. Another helpful addition to the host software would be to grant access not only to the vertex coordinates but also to normal vector, tangent vector, etc, as is the case in Maya(R).

Up to now, all .fx shaders are built with full—and costly—lighting and shading functionality (bump mapping, environment mapping, etc.). This happens to cope with the problem that most of the parameters can be animated: A material that starts pitch black in frame 0 may turn into a dented mirror at frame 10. Here, only an elaborate method to track animations and user input could help. It would have to balance whether it takes more time to build a new, specialized .fx shader in the middle of an animation or to use a more general and therefore slow shader throughout. One could even prepare a pool of shaders (with and without bump mapping etc.) on startup.

To build textures maps by evaluating Cinema 4D's bitmap textures or procedural textures allows to treat a broad range of materials with minimum programming effort. For the bitmap textures, this results in a resampling. This is nearly unavoidable because the usual graphics chips need bitmaps with dimensions that are an integer power of two; otherwise the chips will refuse to employ MIP-mapping.

If many polygons of the scene share the same material, the sampling into a bitmap can reduce rendering time because it saves expensive recomputations of procedural textures. Usually, however, materials are not reused that often. Hence it would be useful to deliver pixel shaders emulating at least a standard set of procedural textures. This would also allow to treat procedural 3D textures like wood or marble. These could in principle be turned into 3D bitmaps on the graphics card, even at reasonable resolution thanks to texture compression. But the computational expense to fill these 3D bitmaps would be prohibitively large.

In the current version of the plug-in, the user can—and has to—control at which resolution the textures of Cinema 4D are (re-)sampled for the hardware-accelerated rendering. For this task an automatic solution may be devised. One could even think of spatial adaptivity so that some parts of a texture map contain more detail than others.

The decision to use a single-pass shader is justified

Feature	Implementation
complex highlights	pixel shader
environment mapping	cube map, pixel shader
bump mapping	normal map, pixel shader
shadows	depth maps
anti-aliasing	multisampling
reflections	—
transparency	—
refractions	—
soft shadows	—

Table 1. Implemented features and possible extensions.

from the benchmarks. Although the number of parameters and simultaneous textures is limited, the plug-in can still render five light sources with shadows. With a future version of CgFX this may increase to the limit imposed by the number of 16 simultaneously usable texture maps.

Like virtually all solutions employing current graphics chips, also this one suffers from several drawbacks as a matter of principle: First, the choice of operating systems is narrowed down to two or even one. Second, incompatibilities between graphics cards as well as software bugs in drivers and toolkits make it difficult to produce a reliable solution. Third, the steady introduction of new hardware features lets any existing solution quickly become outdated. A toolkit like CgFX which accepts high-level code and compiles it for the installed graphics card seems to be an ideal way to cope with most of these problems. However, its final version still has to be released.

6 RESULTS AND OUTLOOK

We have demonstrated a plug-in for a major 3D design software that emulates the offline raytracing engine of this package with help of graphics hardware. The solution allows a reliable preview and/or accelerated movie rendering at high quality including shadows but not reflections or refractions, see Table 1. Especially for movie rendering of not-too-complex scenes at large frame sizes, possibly with full-scene antialiasing, the hardware-accelerated renderer can outperform the built-in raytracer of Cinema 4D.

Several worthwhile approaches to speed improvement result from the benchmarks of section 5:

- The initial setup time for a frame sequence and for each frame may be shortened by checking which materials with which features are needed at which time, and to act intelligently on this information.

- To offer better performance than the built-in raytracer also for complex scenes, the hardware solution needs strong optimization methods, maybe occlusion culling and similar techniques.
- Cinema 4D's standard set of procedural 3D textures may be implemented as pixel shaders.

After that, one may think of implementing a better shadow algorithm [Wei93, Has03], rendering environment maps on the fly to simulate reflections, allowing transparency [Eve2001], and even emulating global illumination [Kau03] as computed by Cinema 4D's built-in engine.

7 REFERENCES

[Ali03] Alias(R) Maya(R) 5.0. www.alias.com, 2003.

[Aro03] Aronson, D., Gray, K. Using the Effects Framework. Published online at www.microsoft.com/msdn, 2003

[Bra03] Brabec, S., Seidel, H.-P. Shadow Volumes on Programmable Graphics Hardware. Computer Graphics Forum 22(3), pp. 433–440, 2003

[dis03] discreet(R) 3ds maxTM. www.discreet.com.

[Eve01] Everitt, C. Interactive Order-Independent Transparency, published online at developer.nvidia.com, 2001.

[Eve02a] Everitt, C., Kilgard, M. J. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. Published online at developer.nvidia.com, 2002.

[Eve02b] Everitt, C., Rege, A., Cebenoyan, C. Hardware Shadow Mapping. Published online at developer.nvidia.com, 2002.

[Has03] Hasenfratz, J.-M., Lapierre, M., Holzschuch, N., Sillion, F. X. A Survey of Real-time Soft Shadow Algorithms. EUROGRAPHICS 2003 State of the Art Reports, pp. 1–20, 2003.

[Kau03] Kautz, J., Hardware Lighting and Shading. EUROGRAPHICS 2003 State of the Art Reports, pp. 33–57, 2003

[Lin01] Lindholm, E., Kilgard, M. J., Moreton, H. A User-Programmable Vertex Engine. Proc. of SIGGRAPH 2001, pp. 149–158, 2001.

[Mar03] Mark, B., Glanville, S., Akeley, K., Kilgard, M. J. Cg: A System for Programming Graphics Hardware in a C-like Language. ACM Transactions on Graphics 22(3), Proc. SIGGRAPH 2003, pp. 896–907, 2003.

[Max03] Maxon Cinema 4D. www.maxon.net.

[Pur02] Purcell, T. J., Buck, I., Mark, W. R., Hanrahan, P. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics 21(3), Proc. SIGGRAPH 2002, pp. 703–712, 2002.

[Sof03] Softimage(R) XSI(R). www.softimage.com.

[Wei93] Weiskopf, D., Ertl, T. Shadow Mapping Based on Dual Depth Layers. EUROGRAPHICS 2003 Short Presentations, pp. 53–60, 2003.