Real-Time Volumetric Shadows in EON Studio

Jarl Lindrud and Henrik Löfgren

June 18, 2004

Abstract

The purpose of this thesis is to describe the implementation of realtime volumetric shadow algorithms in the virtual reality software EON Studio. The standard shadow volume algorithm is covered first, along with the modifications needed in order to handle occluders of arbitrary geometry. We also discuss the implementation of the recently presented penumbra wedge algorithm for volumetric soft shadows, though our conclusion is that this algorithm will need more hardware support before it can be used in a general setting such as EON Studio.

Sammanfattning

Syftet med detta examensarbete är att beskriva implementationen av volumetriska skuggor för realtidsrendering i virtual reality programmet EON Studio. Först beskrivs standardalgoritmen för skuggvolymer och därefter hur denna kan generaliseras för att hantera godtycklig geometri. Vi diskuterar även implementationen av den nya s.k. penumbra wedge algoritmen för att generera mjuka skuggvolymer. Vår slutsats är dock att denna sistnämnda algoritm behöver mer hårdvarustöd innan den verkligen blir användbar under generella omständigheter som i EON Studio.

Contents

1 Introduction

Any end user of computer graphics software will attest to the importance of shadows in adding a sense of depth, proportion and location to a rendered three dimensional scene. The question of how best to compute and render shadows, however, is far from straightforward, and despite vast improvements in graphics hardware over the last years, the addition of shadow-rendering capabilities to a real-time rendering engine is still very much an exercise in balancing quality against performance.

Today, the two commonly used algorithms for real-time volumetric shadow generation are the shadow volume and shadow map algorithms. These algorithms have in common the property that they operate on point light sources, and as a result the computed shadows are sharp, or hard, with instantaneous transitions from dark to light. The real-time computation of soft shadows, i.e. shadows with properly computed penumbra regions, is at present largely an unsolved problem, but considerable effort is being expended in this direction, for the simple reason that soft shadows are far more visually pleasing than hard shadows.

This thesis concerns the implementation of the shadow volume algorithm in the virtual reality software package EON Studio. Our choice of the shadow volume algorithm was determined to some extent by the desire to implement a recently developed soft shadow algorithm, the penumbra wedge algorithm, [2], which is built on top of the shadow volume algorithm.

The shadow volume algorithm is often claimed to be limited to occluders with closed geometry. In many applications, games for instance, this is not a big issue, because the models are usually under the control of the developers, but in our case it would not be practical to impose this requirement, since models are typically provided by the users of EON Studio, and therefore beyond our control. There are however several ways to extend the shadow volume algorithm to non-closed models, and part of our work will be concerned with this issue.

The implementation of the penumbra wedge algorithm is also a focus of this thesis, and the relative merits of our implementation in EON Studio will be discussed.

2 Background

The two real-time volumetric shadow algorithms in common use today are the shadow map and shadow volume algorithms. We will present only a brief overview here, since these algorithms are already very well documented elsewhere, e.g. [1].

2.1 Shadow map algorithm

The shadow map algorithm requires a scene to be rendered first from the point of view of the light, while storing the resulting depth values in a special texture, commonly referred to as the shadow map. The shadow map is then used as the scene is rendered from the point of view of the camera. For each fragment the screen position and the distance to the light is calculated and compared with the corresponding distance value stored in the shadow map at that screen position, and the results of this comparison determine whether the fragment is occluded or not.

This is an inherently robust method, in the sense that anything that can be rendered can be used as an occluder. It is also well supported by most hardware, and for this reason is quite fast. However, the visual quality of these shadows often leaves something to be desired. Individual pixels on the outline of the occluder are usually clearly visible on the edges of the shadow, giving the shadow a blocky, jagged look. Also, care needs to be taken to ensure that the limited precision of the depth buffer doesn't cause sporadic self occlusion.

2.2 Shadow volume algorithm

The shadow volume algorithm was first presented by Crow in 1977, [9]. The stencil buffer implementation described below was first proposed by Heidmann [10], using the z-pass formulation, and the z-fail formulation was described in 2000 by Bilodeau and Songy [11], and Carmack [12].

Given a light source, occluder and viewpoint there is a simple procedure for determining for any other point whether it is occluded, as follows. First we generate the shadow volume of the occluder, by projecting the silhouette of the occluder away from the light by a potentially infinite distance, and cap it on both ends by the front and back of the occluder. We then draw the ray emanating from the viewpoint and passing through the point P which is to be tested. Figures 1 and 2 illustrate the procedure for three different points.

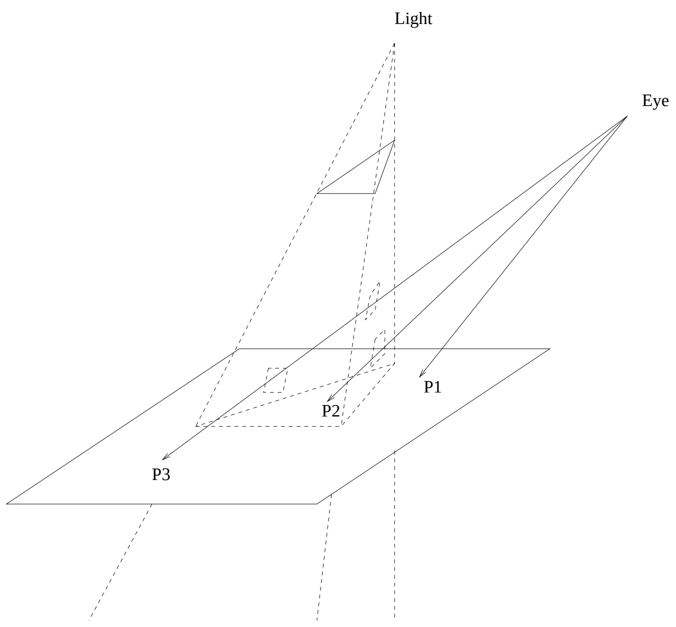


Figure 1: Shadow volumes.

If we move along this ray, from the viewpoint, counting up by 1 each time we enter the shadow volume and down by 1 each time we exit the shadow volume, we find that when we reach P the count will tell us whether P is occluded or not. If the viewpoint is outside the shadow volume, we start the count at 0, otherwise at 1, and we conclude that P is occluded if and only if the count is non-zero when we reach P.

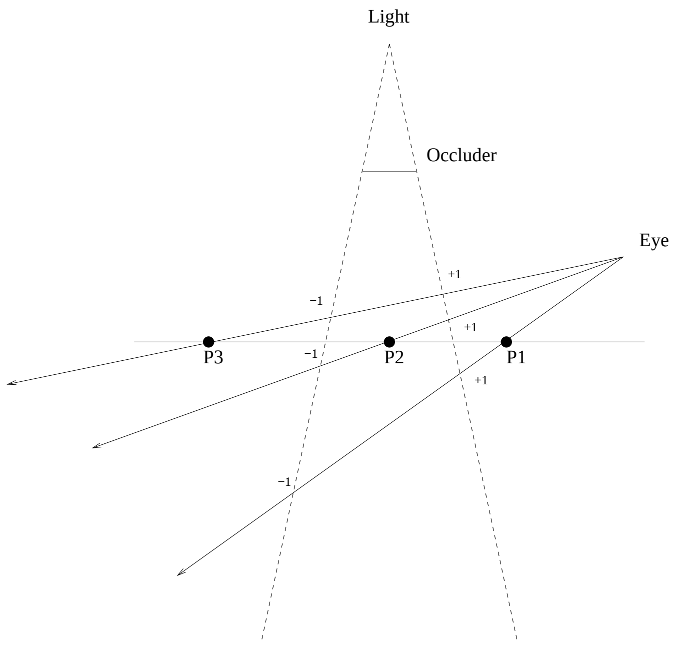


Figure 2: Shadow volumes - schematic.

If we instead start at P, with a count of zero, and proceed away from the viewpoint, we conclude that P is occluded if and only if the count, when we reach infinity, is non-zero.

This is the fundamental idea of the shadow volume algorithm. Counting entry/exits between the viewpoint and P is called the z-pass formulation, while counting entry/exits between P and infinity is called the z-fail formulation.

There are some important differences between z-pass and z-fail, that need to be kept in mind:

1. z-pass requires knowledge of whether the viewpoint is in shadow; z-fail does not.

2. z-pass entry/exits always occur on the sides of the shadow volume; z-fail entry/exits occur on the sides and also on the near and far caps of the shadow volume.

Because of (2), the z-pass formulation is faster, since the shadow volume near and far caps can safely be ignored. However, because of (1), z-pass is very difficult to implement in a robust fashion when the viewpoint is moving in and out of shadows, and for this reason z-pass is really only useful when the viewpoint is known to be outside of all shadows. For general situations, it is usually necessary to use the z-fail formulation instead.

2.2.1 Implementation

Calculating shadow volume intersection points and counting entry/exits may at first glance seem to be a computationally awkward task, but the stencil buffers available on modern graphics hardware can be utilized to do this in a straightforward fashion, as follows.

Assume that the scene has been rendered, with the corresponding depth values written to the depth buffer, and that the stencil buffer has been cleared and contains only zeros. If we now render the shadow volume geometry, we can simply increment/decrement in the stencil buffer for each frontfacing/backfacing primitive that passes the depth test, with the result that after the shadow volume has been rendered, the stencil buffer will contain the z-pass entry/exit count for each pixel.

To get the z-fail entry/exit counts, we proceed in exactly the same fashion, except that we only apply the increment/decrement operations when incoming fragments fail the depth test. Also, since stencil values cannot be negative, we need to ensure that more increments are applied than decrements, so it is necessary to apply the increment on backfacing fragments, since in the z-fail formulation there will always be at least as many shadow volume exits as entries.

Listing 1 demonstrates how one might proceed using OpenGL and C++.

```
#include <opengl/gl.h>

void disableShadowCastingLight();
void enableShadowCastingLight();
void renderScene();
void renderShadowVolume();

void renderShadowVolumeToStencil()
{
        glColorMask(0,0,0,0);    // Disable color buffer writing
        glDepthMask(0);          // Disable depth buffer writing
        glDepthFunc(GL_LESS);    // Set depth test to default

        // Render backfacing tris with +1 stencil operation on zfail
        glCullFace(GL_FRONT);
        glStencilFunc(GL_ALWAYS, 0, ~0);
        glStencilOp(GL_KEEP, GL_INCR, GL_KEEP);
        renderShadowVolume();

        // Render frontfacing tris with -1 stencil operation on zfail
        glCullFace(GL_BACK);
        glStencilFunc(GL_ALWAYS, 0, ~0);
        glStencilOp(GL_KEEP, GL_DECR, GL_KEEP);
        renderShadowVolume();

        glColorMask(1,1,1,1);    // Enable color buffer writing
        glDepthMask(1);          // Enable depth buffer writing
}

void renderSceneWithShadows()
{
        // Unlit pass
        disableShadowCastingLight();
        renderScene();
        enableShadowCastingLight();

        // Lit pass
        renderShadowVolumeToStencil();
        glDepthFunc(GL_LEQUAL);
        glStencilFunc(GL_EQUAL, 0, ~0);
        renderScene();
}
```

Listing 1: Shadow volumes in C++/OpenGL

2.2.2 Shadow volume generation

For the entry/exit counts to give a physically correct indication of the shadowed regions, it is necessary and sufficient that the shadow volume geometry itself is such that it partitions space into two disjoint parts; points that are inside it and points that are not. This is in turn equivalent to the condition that the shadow volume geometry be closed. In fact, we can take this to be the definition of a closed geometry: a geometry is closed if it partitions space into two disjoint parts. This definition can be shown to be equivalent to a more practical definition: a geometry is closed if and only if every edge in the geometry is shared by exactly two triangles.

So the question is: how do we generate closed shadow volume geometries, and what conditions does this impose on the occluder geometry?

For now we will just make the following observation. If the occluder geometry itself is closed, then its silhouette with respect to the light will consist of disjoint closed loops, and the shadow volume geometry can then be constructed by projecting these loops away from the light, using the lightfacing polygons of the occluder to cap the near end of the shadow volume, and the projected non-lightfacing polygons of the occluder to cap the far end. This results in a single closed shadow volume, so for occluders with closed geometry the issue is closed [sic].

A more practical issue that needs to be dealt with when rendering a shadow volume is the effect of the near and far clip planes of the camera projection, since either of these planes may well cull parts of the shadow volume geometry, in effect slicing it open, resulting in bands of incorrectly shadowed pixels that are usually glaringly obvious.

Near plane clipping is irrelevant when using the z-fail formulation, since it doesn't affect the parts of the shadow volume where the z-fail entry/exit counts are taking place. As for the z-pass formulation, since we only use it in situations where the eye is known to be some distance from the shadow, near-plane clipping doesn't matter in that case either.

Far plane clipping is a more serious matter, though, at least with the z-fail formulation. We have to ensure that the shadow volume is wholly inside of the far clip-plane, or else essential parts of the shadow volume will be clipped. The easiest solution to this problem is to use a camera projection that in effect places the far clip-plane at an infinite distance from the camera [6]. The alternative is to extrude the silhouettes a finite, occluder-specific distance such that the shadow volume geometry ends up inside the far clip-plane. For z-pass, far-plane clipping is irrelevant.

In our case, we chose the approach of modifying the projection matrix, since it was acceptable from the viewpoint of the rest of EON Studio's internal rendering process.

2.3 Relative merits

The shadow volume algorithm, as opposed to the shadow map algorithm, provides pixel accuracy, while still providing proper volumetric and self-shadowing effects. It also handles automatically scenes where a light source has occluders spread in a circle around it, a situation where it is not immediately apparent how to apply the shadow map algorithm.

These advantages come at a cost though, the primary ones being the requirements on the occluder geometries, requirement of a stencil buffer, and considerable fill rate consumption.

Continual advances in graphics hardware reduce the importance of the last two issues, and as for the first issue, it turns out that the algorithm can be modified to handle general occluders, albeit at a drop in performance. Given the relative robustness and superior visual quality of shadow volume-generated shadows, and the ever-improving capabilities of graphics cards, our conclusion was that shadow volumes represented a better choice for our purposes.

3 Shadow volumes with general occluders

We have already seen that the shadow volume algorithm works well with closed occluders. To deal with non-closed occluders, from now on referred to as open occluders, we have two basic alternatives. The first is to manually close the model, by adding geometry to the model in such a way that the result is closed, while preferably maintaining a resemblance to the original. The second is to leave the model as it is and instead find a way to apply the shadow volume algorithm to give correct results anyway, preferably without any performance penalties.

3.1 Manual closure

In NVIDIA's md2shader demo, available from [5], we found an example of using the shadow volume algorithm on models that are originally not closed. The models are preprocessed, in order to find loops of open edges, i.e. edges that belong to only one face, and then these loops are capped. The result is, in many cases, a closed model, and the shadow volume algorithm can then be applied. Note that this closure does not affect the usual rendering of the object, since the capped model is only used to generate the shadow volume geometry.

This changes the topology of the model in a nontrivial way, however. If we consider a cube with one face removed, a smaller (closed) model sitting inside it, and a light shining into the open side of the cube, the shadow volume of the cube would extend from the cap and away from the light, and the object inside would appear to be in shadow, as would the inside of the cube.

In this situation the correct closure is apparent - augmenting the 5-face cube geometry with the same 5 faces, but with the new faces facing the opposite direction from the originals. In other words, for each face in the model, add the same face, with the same vertices, but with the opposite normal. This method works better, in general, but at the (high) price of doubling the polygon count, albeit only for the shadow volume generation stage.

Trying to find a middle road between these two extremes proved to be fruitless for us. What the correct, or even the most useful, closure of a model is depends on the intentions of the models author and the models user, so some kind of interactivity would be necessary. One could use capping closures by default, and then let the user decide when this is insufficient, and for these cases use backface closure instead. In the example with the open cube, the user might choose to use capping if the cube is very small and contains no other geometry, while backface closure might be necessary if the cube plays a more substantial role in the scene. Requiring user input is an added complexity though, and the performance penalty of adding backfaces is intimidatingly high.

Our conclusion is that the hit-and-miss nature of this approach disqualifies it as the primary solution to our problem, and that it is better suited for optimizations in special cases.

3.2 Modifying the shadow volume algorithm

The second approach, that of modifying the shadow volume algorithm itself, has been outlined by Bergeron, in [3]. In order to provide some motivation for the otherwise somewhat mysterious modifications, let us first consider how one would apply the shadow volume algorithm in the following simple cases.

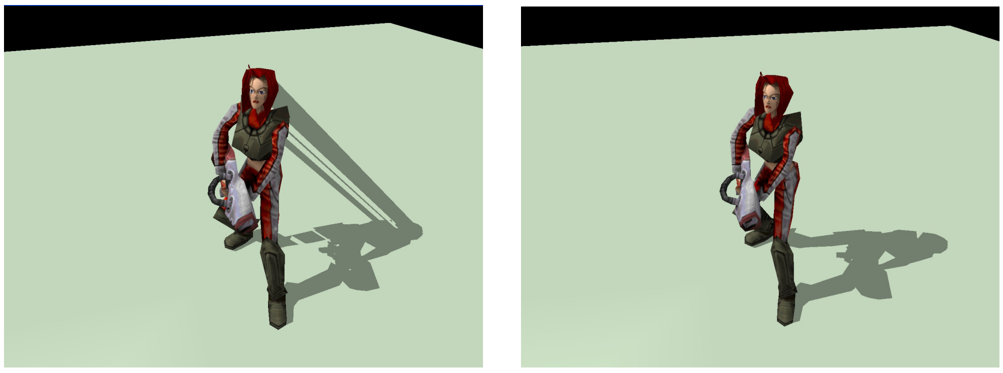
1. If an occluder consists of one triangle, we can generate a closed shadow volume by projecting the three sides and capping both ends, and the shadow volume algorithm can then be applied.

2. If an occluder consists of two disjoint triangles, we construct two disjoint shadow volumes, and apply the algorithm as usual.

3. If an occluder consists of two triangles sharing an edge, with common orientation, then we can once again consider them as disjoint and generate two disjoint shadow volumes. Here we make a key observation though: if the shared edge is not a silhouette edge, then the two shadow volume quads generated from it can be removed from the shadow volume geometry without modifying the entry/exit counts, since an entry into one automatically entails an exit from the other, and vice versa. If on the other hand the shared edge is a silhouette edge, then the two shadow volume quads generated from it can be identified, as long as we increment and decrement by two each time we pass through it.

Consider now the case of a general occluder. We can get the correct result by generating a closed shadow volume for each triangle in the occluder geometry, and then applying the shadow volume algorithm to the union of these shadow volumes. Using the observation above, though, we can eliminate many of the redundancies in this process. Let us by closed edge denote an edge that is shared by exactly two consistently oriented triangles, and by open edge any edge that is not closed. In effect, quads from closed non-silhouette edges should contribute nothing to the entry/exit counts, quads from closed silhouette edges should contribute +2/-2, and quads from open edges should contribute +1/-1. The near cap will consist of the complete occluder geometry, and the far cap will consist of the complete projected occluder geometry.

With these modifications then, the shadow volume algorithm will properly handle any kind of occluder geometry, regardless of whether the geometry is closed or not. A simple example is detailed in Figure 3.



(without modifications) (with modifications)

Figure 3: Shadow volumes with general models.

3.3 Performance

On a theoretical level, the performance hit of the generalized shadow volume algorithm comes mainly in the form of higher polygon counts in the near and far caps. For closed models, it was enough to use the lightfacing half of the occluder in the near cap, and the non-lightfacing half on the far cap, but for general occluders we have to use the whole geometry for both caps.

On a practical level, there is a more serious handicap: the stencil buffers on graphics cards in use today do not offer +2/-2 operations in hardware, only the usual +1/-1 operations. This means that shadow volume quads generated from closed silhouette edges must be rendered twice with +1/-1 to get the desired values in the stencil buffer.

Another practical issue is that of stencil buffer wraparound/overflow. With an 8 bit stencil buffer, there are 256 possible values for each pixel, so correct results can only be guaranteed if rays emanating from the viewpoint encounter less than 256 shadow volumes. With open occluder geometries, the entry/exit counts have been doubled, so artefacts may appear already at a shadow volume count of 128.

Note also that the generalized shadow volume algorithm only needs to be applied to non-closed occluders in the scene; closed occluders can be handled by the usual algorithm, since the entry/exit counts from the two algorithms do not invalidate each other.

3.4 Conclusion

We decided to proceed with Bergerons generalized shadow volume method. Its principal advantage, from our point of view, is that it does not require any modifications to the geometry of individual models. It is a better blanket solution than blindly adding backfaces to a whole model, and it is easily configured to automatically yield zero overhead for closed models.

The two alternatives that we have described are not mutually exclusive though, and capping closures, in particular, are worthy of consideration. Even if not a general solution, in many real-world situations they give a wholly acceptable result, with little or no runtime penalty. For this reason we decided to implement capping closures as an optional optimization and leave it to the users discretion whether to use it or not.

4 Soft shadows with general occluders

Real-time generation of volumetric soft shadows is the subject of active research in the field of computer graphics, but as far as we know there are as of yet no consumer-level graphics software packages that implement the proposed algorithms, simply because it has not been possible to combine real-time framerates with acceptable visual quality.

The most promising approach seems to be the penumbra wedge algorithm, proposed in [2]. Under benign conditions, this algorithm is indeed capable of real-time performance, as has been demonstrated [4].

Our intent was to implement this algorithm in EON Studio, and thus it was necessary to ensure that it was possible to use it with open models. It turned out that the penumbra wedge algorithm can be modified along the same lines as the shadow volume algorithm to handle non-closed occluders. The penumbra wedge algorithm itself is described in detail elsewhere [2], so we will only give a cursory review of it here.

The penumbra wedge algorithm starts by using the shadow volume algorithm to render a hard shadow into a light map texture. In a second step, this light map is then modulated by rendering a penumbra wedge for each silhouette edge. This second step adds and subtracts fractional luminance quantities from the existing values in the light map, and results in a light map, which when clamped to the [0,1] range, gives a soft shadow of generally high quality.

The modifications required to support open models are quite straightforward; we use the generalized shadow volume algorithm, and when rendering the penumbra wedges we double the increments/decrements for wedges generated from closed edges, and leave them as is for wedges generated from open edges.

The not inconsiderable technical details of implementing the penumbra wedge algorithm will be covered in the sequel.

5 Shadows in EON Studio

5.1 About EON Studio

EON Studio is a program for the Windows platform for constructing interactive 3D-scenes. It uses a graphical user interface and is designed to allow both professionals and beginners to build simple to advanced scenes with no programming skills required.

A scene in EON Studio is a scene graph built up from predefined primitives where each such primitive, or node, has a specific functionality. A node may hold data, like the geometry or texture of a model, or may specify some behaviour, like camera movement or how an object in the scene can be manipulated. All these types of nodes are available from the start and it is up to the user to choose which nodes to insert into the scene graph and to modify their default values. The overall structure of the scene graph then defines the visual appearance and functionality of the entire scene.

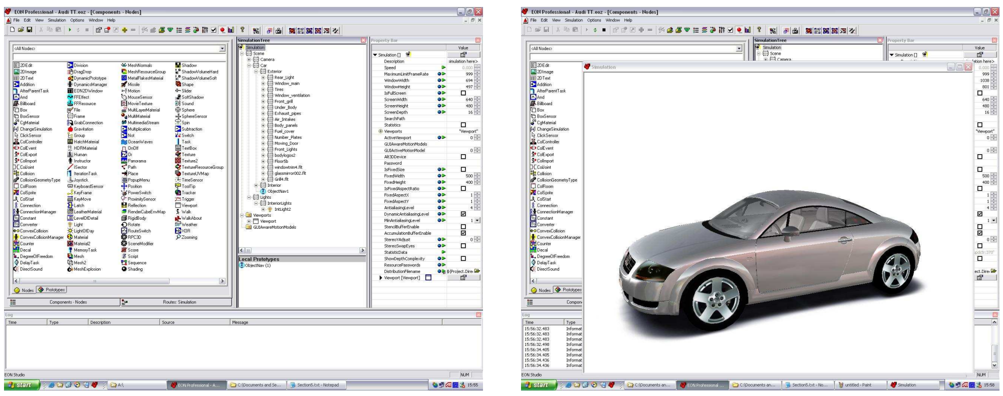


Figure 4: An EON Studio session (left) and the resulting simulation (right).

5.2 Planar soft shadows

Prior to the new shadow volume nodes one were limited to have planar shadows in EON Studio. These planar shadows are added to a scene by using a node called SoftShadow. Given a light source, an occluder and a receiver the SoftShadow node will project the occluder's geometry away from the light source and onto a plane defined by the receiver. To simulate soft shadows one may specify a number of point light sources which will result in an artificial penumbra region when the shadow samples are blended together.

Planar shadows have the advantage of being very fast and can easily be pre-rendered to a texture if there is a static relationship between the light, occluder and receiver.

There are however many limitations to generating shadows this way, of which the two most apparent are that shadow receivers must be planar and that occluders cannot self-shadow. Other limitations concern visual quality. If the shadow is rendered to a texture, as is the case in EON Studio, the texture must be scaled to fit the receiving plane in screen space. This will result in the shadows edges becoming increasingly jagged as the shadowed object is magnified. Also, because a plane is used to apply the texture instead of applying it directly onto the receiver itself, the shadow may extend over the edges of the receiver if it is not large enough to hold the entire shadow (Figure 5). Another limitation concerning visual quality resides in the fact that soft shadows are simulated using several point light sources. Theoretically, the transition between light and shadow can be made entirely smooth by using sufficiently many samples, but in practice there is a rather low upper bound before the frame rate will drop below interactive rates.

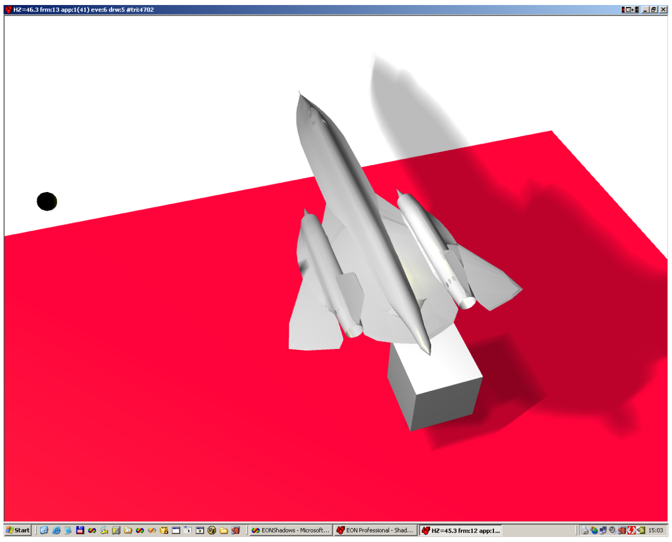


Figure 5: Planar soft shadows in EON Studio.

5.3 Hard shadow volumes

The new ShadowVolumeHard node in EON Studio implements the standard shadow volume algorithm for producing hard shadows but is generalized to handle open geometry. It is used much like the SoftShadow node described above but since the shadow volume algorithm automatically shadows all affected objects in the scene there is no need for specifying any shadow receivers.

Many of the limitations of planar shadows are eliminated when using shadow volumes. Arbitrary objects may receive shadows and occluders will display correct self-shadowing. Also, since the shadow calculation is done in screen space there is pixel-accuracy along the shadows edges regardless of the viewers position.

The main drawback of choosing shadow volumes over planar shadows is that of frame rate. The cost of using hard shadow volumes is significantly higher than that of hard planar shadows. By optimizing the available options in the ShadowVolumeHard node, some of the extra cost may be eliminated. If the viewer is known never to be inside a shadowed region the z-pass option should be used. Using the z-fail option instead allows the viewer to move in and out of shadow but also requires some additional calculations. If an occluders geometry is known to be closed the corresponding option should be set accordingly to achieve higher frame rate. The biggest drop in frame rate occurs when multi pass rendering is turned on. Multi pass will ensure that shadows cast from different light sources intersect correctly with a darker shadowed region, but is much more expensive than using single pass rendering.

In all, the ShadowVolumeHard node should be used more as a complement to than a replacement of the SoftShadow node.

5.4 Soft shadow volumes

The ShadowVolumeSoft node implements the recently developed penumbra wedge shadow volume algorithm for achieving a penumbra region around the shadows edges. Although the algorithm is very similar to the standard shadow volume algorithm, the node is entirely separate from the ShadowVolumeHard node. In addition to the original penumbra wedge algorithm there is support for handling models with open geometry.

The properties previously listed for hard shadow volumes also apply to soft shadow volumes. The limitation of the SoftShadow node where the transition from light to shadow isn't entirily smooth is resolved in the ShadowVolumeSoft node, since the penumbra is calculated in screen space.

An option not found in the ShadowVolumeHard node allows for automatically closing a possibly open geometry of an occluder so the original algorithm can be applied directly. The reason for doing so is not mainly to

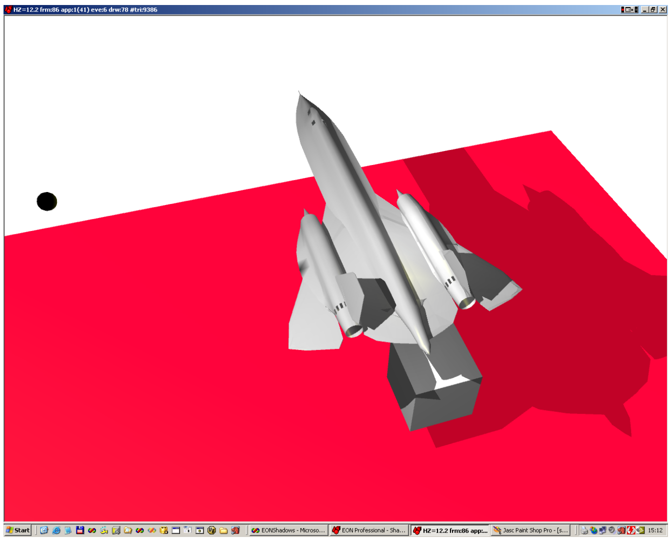


Figure 6: Hard shadow volumes in EON Studio.

gain frame rate but visual quality; when an occluder has open geometry the shadow will look slightly larger and the penumbra region somewhat more compact when compared to the shadow cast from a corresponding closed geometry. This is seldom a problem but can become obvious if a shadow cast from an open occluder is displayed next to a shadow cast from a closed occluder.

The computations for producing the penumbra rely on the light source having an area or volume. The larger the light source, the wider the penumbra region will become. There is an option for setting the radius of the light source since only spherical lights are supported. Increasing the radius of the light source can also be used as a quick hack to remedy the case mentioned previously with open occluders where the penumbra region looks too compact.

The last option not found in the ShadowVolumeHard node is that of shadow weight. The shadow weight is a floating point number between zero and one, indicating how dark the shadow will appear. It is an easy and intuitive way to get the desired darkness of the shadow but is not as physically correct as in the ShadowVolumeHard node where the darkness of a shadow is implicitly computed considering all the affecting light sources.

The calculations involved in generating soft shadow volumes require a modern graphics card supporting programmable vertex and pixel shaders. Even on a high-end machine the use of the ShadowVolumeSoft node may prove to be impractical in general scenes because of the extreme penalty in frame rate. The biggest bottleneck on today's graphics cards is the rendering of the penumbra region of the shadow. Therefore the best way to reduce this performance drop is to decrease the light source radius, thereby making the penumbra region smaller.

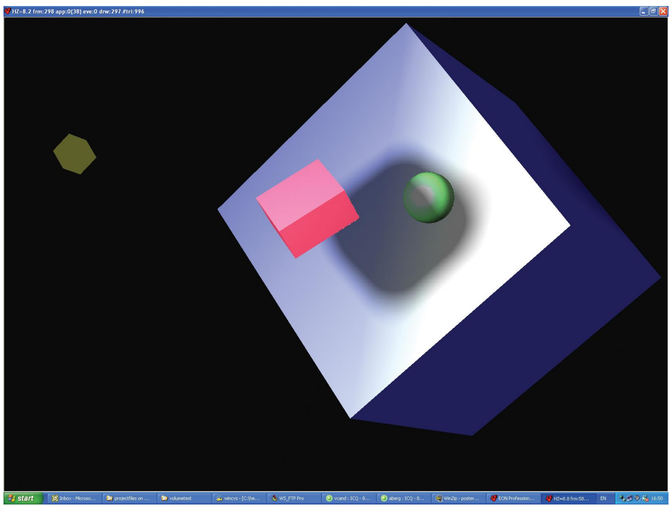


Figure 7: Soft shadow volumes in EON Studio.

6 Implementation in EON Studio

6.1 Initial plan

The main objective of this final thesis project was to extend the shadow volume algorithm and to implement these ideas in the environment of EON Studio. However, implementing the soft shadow volume algorithm from scratch is a great undertaking and would require much more focus on implementation details than on the theoretical aspects of the project. Therefore we decided to use the work of another final thesis group as a starting point and extend their implementation of the penumbra wedge algorithm to handle open geometry.

As always when it comes to software development, things have a tendency of being delayed and so when we were ready to move from theory to practice the code we needed just wasn't finished. To avoid any idle time in our schedule we decided to start our work in EON Studio by implementing the hard shadow volume algorithm, adding support for open geometry. Then, when the other final thesis group was finished, we should be able to move the new code into EON Studio with much of the supporting code already being in place.

6.2 The ShadowVolumeHard node

When we were ready to begin implementing hard shadow volumes in EON Studio we had already proven that the ideas for handling open geometry worked by modifying the Infinite Shadow Volumes demo by NVIDIA [7]. During that time we had also familiarized ourselves with the basic structure of an implementation of shadow volumes as well as some potential difficulties. The task at hand was then to transfer the core of the algorithm from our modified demo into the EON Studio environment and adapt it to work with the EON system and its internal geometry format.

The incorporation of hard shadow volumes in EON Studio was done by means of a new node. To create the first trembling lines of code we used an existing tool called the EON Node Wizard which generates skeleton code for a new node. Since we also had access to the source code for the SoftShadow node we had very little problems understanding how a node worked, what the different methods were used for and which ones we needed to implement.

Each node in the scene graph is accessed through a set of methods. If a node wants to render something to the screen, it must implement the rendering method which is then called once each rendering pass. If a node wants to be updated or maybe do some calculations before a rendering pass, the corresponding methods must be implemented and so on. This hierarchy of function calls means that though a specific node has full control over itself, it has very little possibilities of affecting other nodes. In our context of shadow volumes this means that our new node has too little control over each rendering pass. Since the shadow volume algorithm needs to render the scene multiple times we need to gain control at a higher level in the hierarchy.

The top level rendering function is located in something called GLRM, which is basically a collection of classes and functions for doing the actual rendering of a scene. To allow GLRM and the shadow volume nodes to communicate without disturbing the existing structure too much we designed a small interface (Figure 8). Through this new interface a shadow volume node can record itself in a registry and thereby notify GLRM of its presence and how it is configured. By collecting information from this registry GLRM can then calculate how the scene should be rendered.

Things are somewhat complicated by the fact that there may be several shadow volume nodes present in the same scene. This means that a light source may be used as a shadow casting light by several different nodes. For the multi pass nature of the shadow volume algorithm to work correctly in this case we must extract the light sources from all nodes prior to rendering. The reason for doing this is that each render pass must be performed on a per light source basis and not on a per node basis.

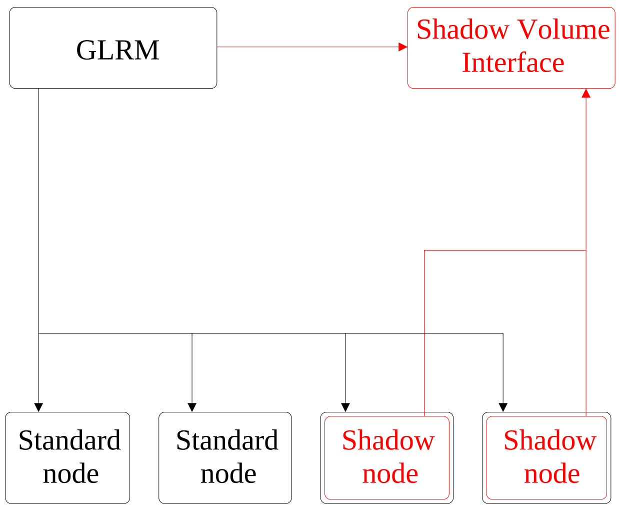


Figure 8: GLRM, the nodes and the new interface.

6.3 One node becomes two

By the time the soft implementation we had been waiting for was completed, we already had a working implementation for doing hard shadow volumes supporting open geometry in EON Studio. Not only did this implementation work quite well but it also had no special requirements on the graphics card other than the presence of a stencil buffer.

The newly obtained code for doing soft shadow volumes made extensive use of OpenGL extensions throughout the implementation and thereby had much greater requirements on the graphics card. Although the new code had built in support for hard shadow volumes as well, we were reluctant to replace the existing code in EON Studio. That would mean that users would be forced to buy more expensive hardware than was really needed for doing hard shadow volumes. Therefore we decided to split the implementation of shadow volumes in EON Studio into two nodes; a simpler one for doing hard shadows with modest requirements on the graphics card and a second one for doing soft shadows targeting high-end systems.

6.4 The ShadowVolumeSoft node

The penumbra wedge algorithm is technically more difficult to implement than the shadow volume algorithm. Among other things, it is no longer practical to use an 8-bit stencil buffer to define the shadow. The stencil buffer can only mask out hard-edged areas of the screen, so it is necessary to use some other mechanism to define and apply the shadow.

Instead of the stencil buffer, it becomes necessary to use textures, together with the render-to-texture capabilities available on modern graphics card. A typical texture will have 32 bits available for each pixel, and this data is available in the vertex transformation and lighting stage of the graphics pipeline. Instead of using `GL_INCREMENT/GL_DECREMENT` on the stencil buffer, we target a texture and render special color values, with the rasterizer configured for additive blending. Since it is not possible to do subtractive blending, it becomes necessary to use two textures; one for additive contributions and one for subtractive contributions. In a final stage the shadow value can then be computed by sampling both textures and taking the difference.

The actual application of the shadow to the screen must also be done differently from the shadow volume algorithm, since the shadow is stored in a pair of textures and not in the stencil buffer.

The most accurate way of applying the shadow is to do per-pixel lighting using dedicated pixel shaders, and in these shaders the per-pixel shadow value can be computed from the additive/subtractive shadow textures, and then used to modulate the result of the lighting computation.

This method may not be applicable in all situations though, in particular scenes that are being rendered with a fixed-function graphics pipeline or with per-vertex lighting. In these cases, one can resort to applying the shadow in a post-rendering pass, using alpha blending to render a screen-sized black rectangle to the screen, with the alpha values calculated from the shadow textures. This results in a darkening of the areas of the screen that are in shadow, but with some loss of accuracy. The most apparent artefact is that specular highlights from occluded lights will still be visible, albeit darkened, but the results are in most cases passable.

In EON Studio, it is possible to associate Cg shader programs with objects in a scene. When executed, the Cg programs are automatically supplied with various parameters from the rendering engine, such as the coordinate transformation matrices currently in use. By default, objects do not have shaders associated with them, and it is necessary for the user to supply and associate the shaders with the corresponding objects. One of the appeals of using Cg shaders is that they provide a substantial measure of extensibility; the shaders can be written by anyone, perhaps originally for other purposes, but still be seamlessly integrated into EON Studio's rendering process.

To use the first shadow application method, then, it would become necessary for all shadow receivers to use Cg shader programs. Furthermore, all such shaders, irrespective of origin and author, would have to implement code to retrieve and apply the shadow values from the shadow textures.

We deemed this to be impractical, and settled on the second method instead. This had the happy consequence that the architecture of the ShadowVolumeSoft node became considerably simpler than that of the ShadowVolumeHard node. There is a priori no longer any reason to interact with the GLRM DLL; in a post-render callback function we simply trigger the shadow computation, and then apply the shadow to the screen, on top of what has already been rendered.

Implementing the ShadowVolumeSoft node turned out to be quite time-consuming though, but for reasons that had mainly to do with the internal complexities involved with computing the shadow textures. As previously mentioned, we had at our disposal a functioning implementation [4], but the strong coupling of the shadow code to the rest of the application made it difficult to extract the requisite code without breaking anything, and our task was further complicated by the fact that there were in essence two separate rendering paths, depending on the available graphics hardware (ATI or NVIDIA), of which we only had access to one at a time.

The modifications needed to handle open occluders were implemented in a relatively straight-forward way. As mentioned previously, penumbra wedges are rendered with single or double contributions depending on whether they were generated from open or closed edges.

7 Conclusion

In general, our work with this thesis had considerably more practical than theoretical emphasis.

Our initial problem was to apply the shadow volume algorithm in a general setting, without restrictions on occluder geometry, and we found a solution to this problem at a relatively early stage. As noted previously, our approach is suitable for an application like EON Studio, where there is a priori no knowledge of the models which need to be handled. In situations where the models are known beforehand, as is often the case, the whole issue of occluder geometry becomes to some extent irrelevant, since the models can be doctored to yield correct results.

The remainder of our work was concerned with the implementation in EON Studio of the shadow volume and penumbra wedge algorithms.

Despite the relative simplicity of the shadow volume algorithm, implementing it in the context of a large and pre-existing codebase was not wholly straightforward. Implementing the shadow volume algorithm required modifications to the rendering engine itself, and these modifications needed to be accessed by the user interface; this meant that some additional infrastructure was needed, that would not interfere with any of the pre-existing code.

In retrospect, it would have been wise to commence some boilerplate coding in EON Studio at the outset, instead of waiting until we knew what we wanted to implement. Getting up to speed with the build environment and EON Studio itself took quite a bit of time, and could well have been done in parallel with our previous activities, which would easily have saved us two or three weeks.

By the time we turned to implementing soft shadows using the penumbra wedge algorithm, we were better versed in the ways of EON Studio, and almost all the time we spent on this portion was concerned with internal details in the algorithm and the implementation of it that we had at hand. The EON-specific details were relatively simple; the only complication that we came across was needing to learn and write some simple Cg shader programs.

We have not had the time to do proper profiling and optimization of our two implementations, and for this reason we have not presented any performance figures. The soft shadow frame rates vary widely with hardware, but with the hardware we have tested it is not likely that they would be practical in typical scenes. The hard shadows provide a well functioning alternative, especially in dynamic scenes where the differences between hard and soft shadows are not as noticeable.

References

[1] T. Akenine-Möller, E. Haines, Real-Time Rendering, 2nd edition, 2002.

[2] U. Assarsson, "A Real-Time Soft Shadow Volume Algorithm", Ph.D. thesis, Chalmers University of Technology, 2003.

[3] P. Bergeron, "A General Version of Crow's Shadow Volumes", IEEE Computer Graphics and Applications, vol 6, no. 9, pp. 17-28, September 1986.

[4] U. Borgenstam, J. Svensson, http://www.ce.chalmers.se/staff/tomasm/soft/soft_shadow_v3_1_src.zip

[5] NVIDIA Developer Relations, http://developer.nvidia.com

[6] C. Everitt, M. Kilgard, "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering", NVIDIA Corporation 2002.

[7] Infinite Shadow Volumes, http://developer.nvidia.com/view.asp?IO=robust_shadow_volumes

[8] "OpenGL Programming Guide", 2nd edition, Addison Wesley, Silicon Graphics, Inc. 1997

[9] F. Crow, "Shadow Algorithms for Computer Graphics", Proceedings of SIGGRAPH, 1977, pp. 242-248.

[10] T. Heidmann, "Real shadows, real time", Iris Universe, No. 18, p.23-31, Silicon Graphics Inc., November 1991

[11] B. Bilodeau, S. Mike, "Real Time Shadows", Creativity 1999, Creative Labs Inc. sponsored game developer conferences, Los Angeles, California, and Surrey, England, May 1999.

[12] J. Carmack, unpublished material, 2000.

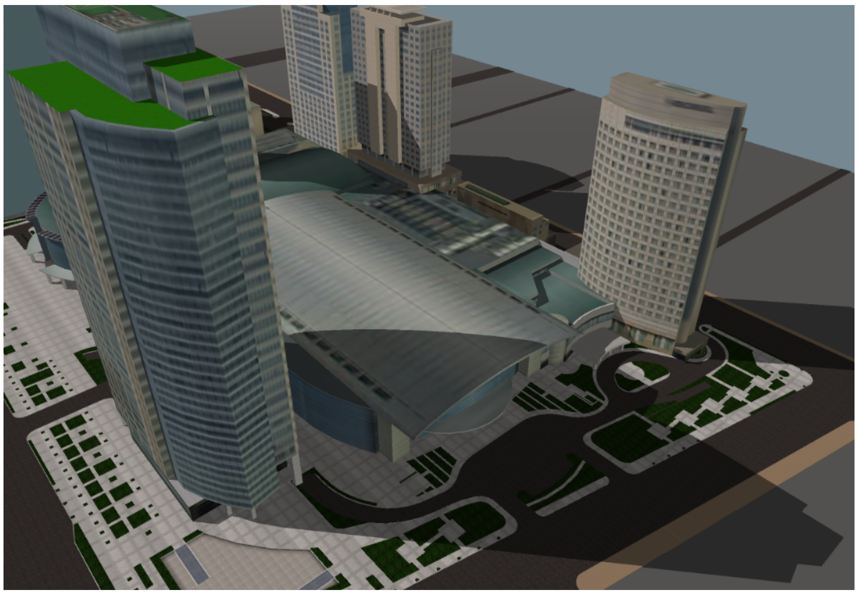


Figure 9: The ShadowVolumeHard node used in a more complex scene.

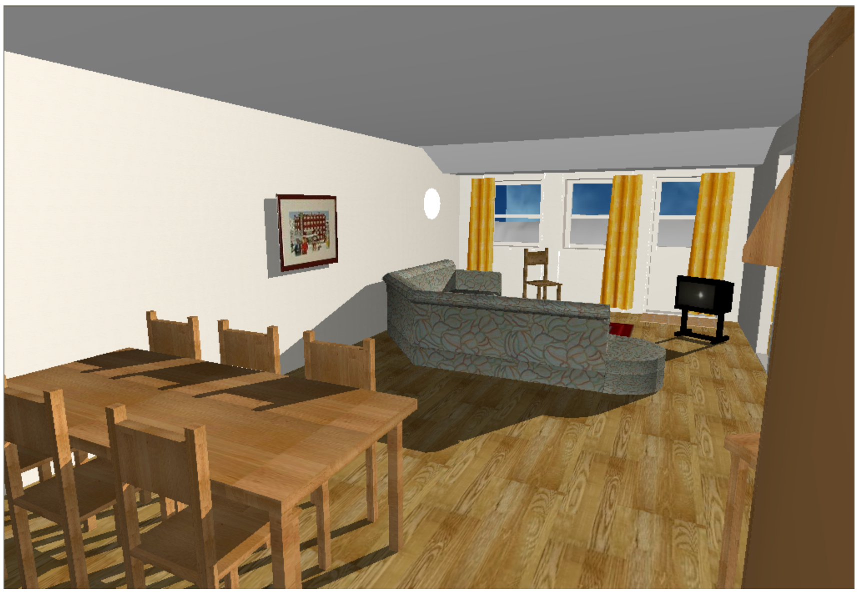


Figure 10: Hard shadow volumes generated from all objects in the room.

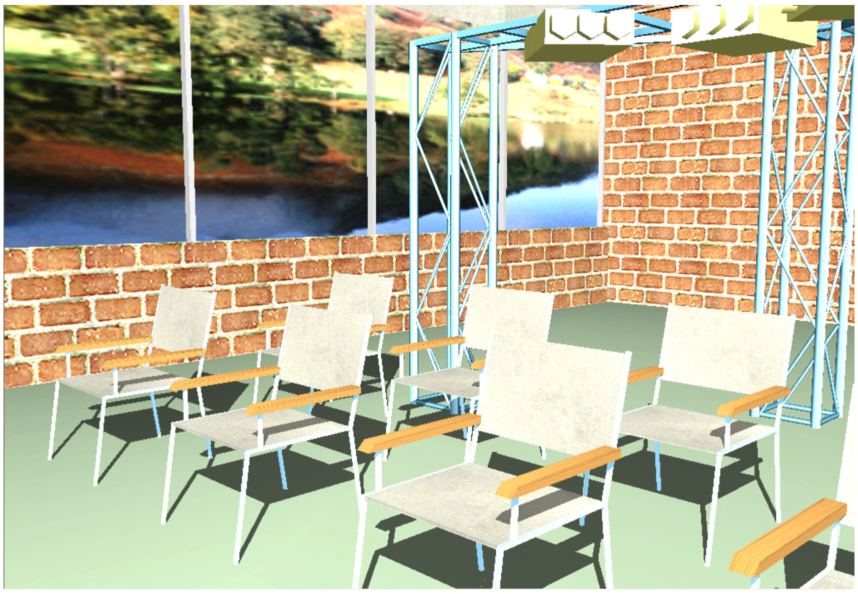


Figure 11: Hard shadows at 60 fps on an NVIDIA Quadro FX 3000.

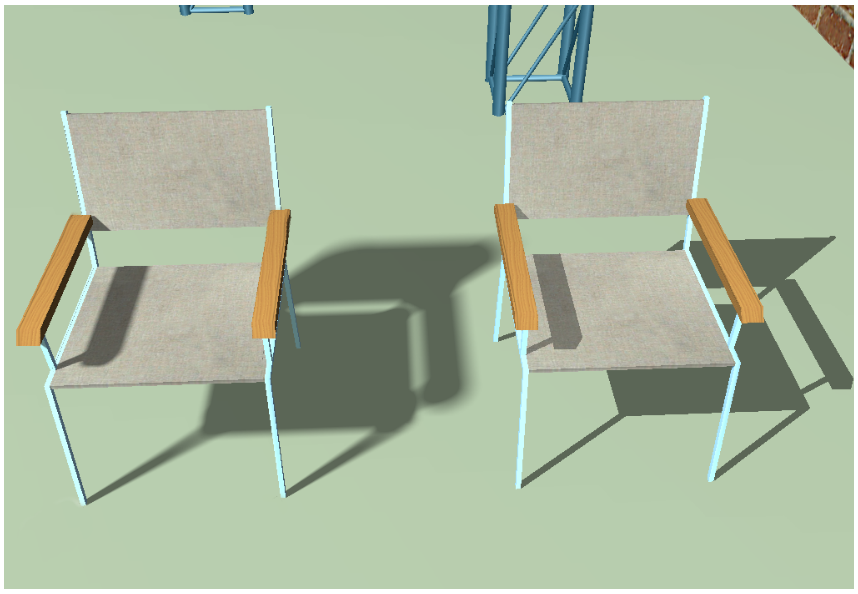


Figure 12: Soft versus hard at 4 fps on an NVIDIA Quadro FX 3000.