Enhancing X3D for advanced MR appliances

Yvonne Jung* Tobias Franke† Patrick Dähne‡ Johannes Behr§

Fraunhofer Institut für Graphische Datenverarbeitung

Abstract

In this paper, we explore and discuss X3D as an application description language for advanced mixed reality environments. X3D has been established as an important platform for today's web-based visualization and VR applications. Yet, there are very few examples for augmented reality systems utilizing X3D beyond a simple geometric description format. In order to fulfill the image compositing and synthesis requests of today's augmented reality applications, we propose extensions to X3D, especially with a focus on lighting and realistic rendering.

CR Categories: I.3.7 [Three-Dimensional Graphics and Realism]: Virtual reality—Color, shading, shadowing, and texture I.3.6 [Methodology and Techniques]: Standards—Languages

Keywords: X3D, Differential Rendering, Image Based Lighting, Shader, Real time, Mixed Reality, Camera model

1 Introduction

X3D is a well established standard for creating VR applications, but there are few examples for using X3D in augmented (AR) and mixed reality (MR) applications. Many MR systems allow to use geometric models to be loaded from X3D files, but none use X3D for the application logic. The reason for this is that X3D has been developed with web-based 3D applications in mind and therefore lacks important features needed for MR applications. In contrast to VR, in MR we do not have completely synthetic scenes. Instead we try to integrate virtual objects in existing real scenes. To do this, we have to determine the exact pose of the user, which is usually done by using video tracking systems. Furthermore, we have to put a video image of the real scene behind the virtual objects. Image analysis in general and tracking in particular are out of scope research topics not covered by this paper, but we have to provide interfaces to integrate various sensor data streams. Unfortunately, X3D currently does not have language features that allow integrating data streams from external devices like tracking systems and video cameras into the scene graph. To overcome this limitation we present a set of data stream sensor nodes.

For MR applications, the X3D Viewpoint node is not sufficient. The Viewpoint node implements the classic pinhole camera model. Real video cameras do not follow this simple model. Because we integrate virtual objects into video images, we need to use a camera model that simulates the real camera as closely as possible. Therefore we present an extension to the Viewpoint node as well as a completely new node that allows directly specifying the projection and modelview matrices.

The previously mentioned extensions are essential for all AR applications, but they only allow standard fixed function shading methods with an unrealistic looking local lighting model. These shading methods suffice for typical AR scenarios like assembly simulations, where the augmentations often only consist of additional textual information or simple virtual objects like gouraud shaded arrows. This kind of augmentation requires at least geometric camera calibration for recovering camera pose parameters, and possibly the 3D geometry of the scene. But for other applications, ranging from the film industry, to virtual prototyping in the automotive industry, to virtual furniture in real environments for the consumer market, it is becoming more and more important to fit virtual objects seamlessly into real world scenes.

To seamlessly integrate virtual objects into the real scene at interactive frame rates, it is also important to employ advanced rendering techniques. The virtual objects must be lit in a realistic way, and they have to cast shadows onto real objects. To achieve this, the synthetic objects not only need to be registered geometrically but also photometrically for consistent lighting. This is usually done by means of image based lighting (IBL) techniques for computing the lighting by previously acquiring a high dynamic range (HDR) light probe image or by using a 180 degree fish-eye lens for capturing real time environment maps. Lastly, the effects of changing the light transport paths of the real scene by inserting synthetic objects (which, for example, can be occluded by real objects or throw shadows onto them) also have to be taken into account. This can be solved with differential rendering, a rendering technique that adds the difference of two synthetic images to the real input image.

Consistent illumination in AR applications is still an open field for research because besides the lighting simulation, i.e. photo-realistic rendering, three other problems have to be solved in advance: geometry reconstruction for handling e.g. occlusions, and second, lighting reconstruction for recovering number and type of primary light sources (which eventually are the cause for shadows as well). The third problem is the material reconstruction for determining the reflectance properties - like the bidirectional reflectance distribution function (BRDF) and the bidirectional texturing function (BTF) - of real materials, which is quite essential for computing correct interreflections and shadow color, but far beyond the scope of this paper.

Our paper has 7 sections: Following the introduction is section 2, in which we discuss related work in the field of MR and X3D. In section 3.1, a short introduction to irradiance mapping with spherical harmonics and shadow mapping is presented. Furthermore, in section 4, differential rendering, as the standard method for computing consistently illuminated MR images, and its implementation details are examined. Consequently, in section 5 the integration of those mechanisms including our node extension proposal is addressed. Results of our work are presented in section 6, followed by the conclusion in section 7.

2 Related Work

Differential rendering is a two pass composition technique that is feasible for augmenting images or video streams with consistent illumination. It requires two lighting simulations, one with the real

*e-mail:yvonne.jung@igd.fraunhofer.de

†email:tobias.franke@igd.fraunhofer.de

‡email:patrick.daehne@gris.informatik.tu-darmstadt.de

§e-mail:johannes.behr@igd.fraunhofer.de

Web3D 2007, Perugia, Italy, April 15–18, 2007.
© 2007 ACM 978-1-59593-652-3/07/0004 $5.00

scene only and a second one with additional virtual objects taken into account. The term lighting simulation usually implies global illumination techniques, therefore the first implementations were radiosity based (i.e. hierarchical radiosity, with a line-space hierarchy of links and light shafts for rapidly identifying modified links when interacting with the virtual object [Drettakis et al. 1997]), because of its view independence and good quality. This approach of Drettakis et al. was improved by Loscos et al. [Loscos et al. 1999] by introducing a special pixel data structure with a final gathering pass for handling the direct lighting component. Although they did not deal with moving viewpoints and light sources, animations or even complex BRDFs, the use of global illumination methods has the advantage of automatically computing shadows and indirect illumination, which for example is observable in reflections and color bleeding effects.

Although global illumination is much too slow for realtime applications, it is still prevailing in the field of consistent illumination. An interactive ray tracing approach was proposed by Pomi and Slusallek in [Pomi and Slusallek 2004], but it needs special ray tracing hardware. Real-world lighting information for shading virtual objects was first introduced by Debevec [Debevec 1998]. He distinguishes between the local scene, for which a reflectance model is needed, and a distant, light-based scene, which only serves as the source for natural illumination. This incident lighting information, the real scene radiance, is captured by using an omni directional HDR image [Debevec and Malik 1997] of a mirrored ball, the so called light probe. He also states, that the realism of the final composited image depends on the error in the rendered local scene compared to the real image and hence on having a good model of the reflectance properties of the real materials.

A GPU based real time method was proposed by Gibson et al. [Gibson and Chalmers 2003; Gibson et al. 2004], in which the virtual object is illuminated by using an irradiance volume in combination with cubic reflection maps. Another important aspect of this context is the final tone mapping, in which the radiance is transformed into displayable pixel colors in the range [0;1] by means of the camera response curve. Shadows are computed by utilizing a shaft-based data structure between source and receiver patches in the sense of a radiosity system with pre-computed transfer of the radiance along these light-, and therefore potentially shadow-, paths. Because correct shadows are essential for a proper perception, in [Korn et al. 2006] some methods are shown for retrieving the number and 3D positions of the light sources. An interesting approach combining geometric and photometric calibration was recently outlined by Pilet et al. [Pilet et al. 2006]. It uses a calibration object with known shape and normals and view independent albedo from which a light map is calculated based on Lambert's law and the observed mean intensities.

Another framework using IBL and soft shadows (but not HDR), can be found in [Supan and Stuppacher 2006]. Image based lighting and irradiance maps are used in mixed reality simulations as a means to transfer real world lighting onto surfaces of virtual objects. Image data is captured via a 180 degree fish eye lens or photos of light probes. Because these images contain direct and indirect lighting, and the positions of the light sources cannot assumed to be readily available from the footage, they must be filtered to extract different frequencies of lighting for different types of material, for instance low frequency lighting for diffuse surfaces. The authors use an image space blur-filter to generate the different irradiance maps. A downscaled mipmap is used for very low frequencies. While this approach is fast and stable, the deviation from a correctly filtered image becomes greater with decreasing frequencies. Another important aspect is, that for specular irradiance maps, there is no clear connection between the radial blur and lighting parameters like the shininess as given in the here used phong lighting model.

To extract direct light sources for shadow rendering, more algorithms are needed to identify direct lights inside the irradiance maps, some of which are proposed in [Korn et al. 2006], but they still all have to deal with problems like processing time and jitter concerning the derived light position and area. In order to cast realistic, dynamic shadows onto real as well as virtual objects from reconstructed or even virtual light sources (e.g. additional lamps), appropriate shadowing algorithms must be integrated, too. Suitable techniques are discussed in more detail in section 3.2.

3 A new lighting model

A major problem with X3D's applicability in the MR domain is the missing support for global illumination techniques. Since virtual objects have to resemble reality as close as possible, simple lighting models won't suffice in this context. In this section we shortly describe the irradiance mapping technique for enhancing visual quality to wards more realism.

3.1 Irradiance Mapping

Simulating complex lighting behavior with indirect illumination has been solved for mirror-like surfaces through standard environment mapping techniques. A fish eye image or a reflective surface respectively, most often a simple sphere, is captured and directly applied to another more complex surface, for instance via sphere mapping. While this usage introduces some errors, at least for different shapes, the results are optically pleasing and usually feel right. But it should be noted, that this only captures the far field.

The idea of irradiance mapping is to further extend this method. Instead of just calculating textures for reflective materials, others - for instance diffuse or specular surfaces - can have their own textures as well. Collectively, these textures are called irradiance (environment) maps. A great deal of research has been invested in finding out how to generate irradiance maps effectively and without much deviation from real light-integration through spherical harmonic analysis. Likewise more recent papers have shed light on base functions that enable quick analysis of different ranges of frequencies[Ng et al. 2004]. In [Ramamoorthi and Hanrahan 2001] the authors showed that low-frequency representations of spheremaps can be efficiently calculated with spherical harmonic analysis. The base function for these representations is the Legendre polynomial, which can be mapped onto a sphere and redefined for real numbers as shown in equation 1.

$$y_l^m(\theta,\phi) = \begin{cases} \sqrt{2}K_l^m \cos(m\phi)P_l^m(\cos\theta), & m > 0 \\ \sqrt{2}K_l^m \sin(-m\phi)P_l^{-m}(\cos\theta), & m < 0 \\ K_l^0 P_l^0(\cos\theta), & m = 0 \end{cases} \quad (1)$$

$$K_l^m = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} \quad (2)$$

In our simulation, the reconstruction process has been implemented in a shader, which simply dots the constant spherical harmonic values with the corresponding coefficients. The function values of $y_l^m(\theta,\phi)$ are precomputed and stored inside textures, which are used within the shader program to determine the values of the spherical harmonic functions. Another implementation can be found in [King 2005]. In order to make these irradiance textures (which have to be regenerated whenever the input image changes), readily available, we propose the *SphericalHarmonicsGenerator* node which will be described in detail later.

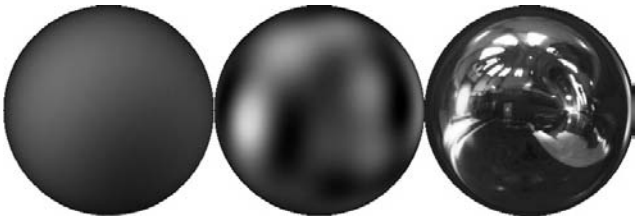

Figure 1: Debevec's grace probe reconstructed with 4 and 11 coefficients (original image on the right).

Figure 2: Comparison: Real lightprobe (left) and rendered sphere with irradiance map (right).

By representing a spheremap as a parameterized function, it can be analyzed and filtered to create other irradiance maps. In figure 1, the famous "Grace Cathedral" probe from Paul Debevec has been filtered to lower frequencies. The original image on the right is used to simulate highly reflective material, whereas the spheremap in the middle and left represent glossy and diffuse surface appearances. In figure 2, a direct comparison can be seen between the real light probe that was used to capture incoming light from a real scene and a slightly filtered irradiance map applied to a virtual sphere.

The MR simulation setup is shown in figure 3. We use a lightprobe or a fish eye camera to capture the real lighting configuration and a second (preferably HDR) camera for the scene. Together with the reconstructed geometry and material information about the real scene, the object is fed to the simulation that merges the real world with the virtual object via differential rendering (see section 4). The composition is then displayed on a screen.

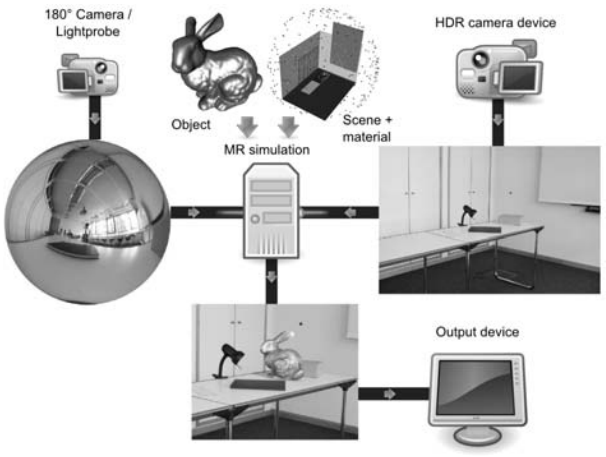

Figure 3: The system setup of our MR simulation.

3.2 Shadows

For lighting configurations that can be represented as images or spherical functions, the subsequent harmonic analysis can produce sets of spheremaps to simulate the reflection of light on the surface of an object. This method allows for high quality rendering and lighting simulation - a task which is crucial for MR applications. Since the frequency representation of an irradiance map does not include transfer functions on the model's surface other than basic reflection behavior (diffuse, glossy, reflective) nor any drop shadows the model might cast, those have to be included by other means, e.g. Precomputed Radiance Transfer[Sloan et al. 2002]. PRT however is limited to rigid objects. LDPRT[Sloan et al. 2005] on the other hand addresses this problem by concentrating on local effects, leaving distant effects like shadows aside.

To solve this problem, we have used high quality shadow algorithms like PCF or PCSS, which are described in the following paragraphs, in conjunction with a modification of real time ambient occlusion methods [Sattler et al. 2004] as a substitute for more complex transfer function properties. While this approach does not address special surface or material properties like subsurface scattering, it yields better visual quality than the simple lighting models of today's graphics hardware while retaining high rendering speed. High dynamic range irradiance maps further help to add realism, thus making a clear distinction between real and simulated objects harder than with simple rendering techniques.

Realtime shadows can be computed by using e.g. plane projections (a simple technique which only works for planar surfaces), texture mapping, and shadow volumes. Although the latter method yields very accurate shadow edges, besides being heavily multi pass based, it additionally requires the creation of shadow volumes for every object, and therefore scales bad with size and complexity of a scene. For arbitrary scenes current shadow mapping algorithms are much better suited, because they are fast and nearly independent of the scene complexity.

On the one hand here we have filtering approaches like the so-called percentage-closer filtering shadows. PCF [Reeves et al. 1987] simulates shadows from area light sources, by calculating the mean value of n shadow tests for every pixel, which, for small n, already is supported by modern GPU's. Perspective PCF soft shadows (PCSS) also include the distance between occluder and occludee by using a variable kernel size based on an estimate of the penumbra size [Fernando 2005]. In contrast to uniform PCF shadows the latter technique leads to more convincing results, but because of the more complex filtering, in our tests frame rates dropped to at most 30 to 40 percent of those achieved with PCF.

On the other hand we have hard shadows like standard shadow mapping (which even runs on old graphics hardware, but suffers from aliasing artifacts caused by the finite shadow map resolution and the well known polygon offset problem caused by the finite depth buffer precision), and high quality hard shadows like the light space perspective shadow maps (LISP) [Wimmer et al. 2004]. Just like with perspective shadow mapping the shadow maps are calculated in post-perspective camera space for reducing aliasing artifacts by scaling up those regions, which are next to the near plane by transforming the camera frustum to a unit cube. But here the calculation of the virtual camera is done in lightspace to alleviate the perspective distortion which otherwise leads to a remarkable loss in quality for objects outside the close-up range. In our test scenes the frame rates of LISP shadows were only around 7 percent lower than with standard shadow mapping, which is a minor loss compared to the big gain in visual quality.

The X3D lighting model[Web3DConsortium 2006] does not in-

Figure 4: Different materials and indirect lighting simulated through irradiance mapping.

clude any form of real time shadows since it follows more or less the GPU-fixed-function pipeline[Chronos 2006]. However, there have already been some proposals to include real time shadows in X3D and we believe that this feature is essential for future applications, even beyond the field of AR. Sudarsky[Sudarsky 2001] generates shadows for dynamic scenes but only on user defined surfaces and for point lights which must be outside of the current scene. The *Shadow*-node extension from Octaga[Octaga 2006] supports all X3D-light types - *DirectionalLight*, *PointLight* and *SpotLight* - and provides a "detail" field-parameter which is used to specify the resolution of the shadows. However, soft-shadows are not supported, and the implementation as *Group*-node which links all occluders and lights does not look natural. The Contact-browser[bitmanagement 2002] provides vertex/fragment-shaders which can be used to generate dynamic shadows, but the application-developer has to handle the shadow-map generation and processing per object directly in the shader code. This may be a very flexible solution but not one which all scene authors would intuitively think of.

Usability improves much, if shadows can be displayed without the need for modifying already existent shader programs. In our implementation we solved this problem by also introducing a "shadow factor map", an intermediate texture which contains the results of the shadow tests. A quality improvement without noticeable decrease in performance can be gained by filtering this texture [Shastry 2005]. Because simple filtering leads to halo effects around unshadowed objects which are occluding shadowed regions, another texture containing an edge image from camera view is generated with the help of an edge detection shader. Now, blurring is only done, if the filter region does not contain a silhouette edge. The final shadow composition is done by first rendering the whole scene without additional shadows into a texture called "color map", followed by the "shadow factor map" creation, and finally the multiplication of both.

4 Differential Rendering

As already stated, a reconstruction of the real geometry, for which we used an external tool, is necessary at least for handling occlusions and receiving shadows. Because two lighting simulations or respectively rendering passes are needed for calculating the difference image (see Figure 5 for visualization), we first need to introduce the concept of multi pass rendering in the context of X3D.

Multi pass can basically be understood in two ways. On the one hand it means the ability to dynamically render a partial scene graph, which does not necessarily need to be part of the original scene, to an offscreen texture, that can then be used for creating effects like reflection or refraction. In the Xj3D extension documentation [Xj3D 2004] a simplified possibility for creating such offscreen images was first proposed with the *RenderedTexture* node.

On the other hand the term multi pass denotes the ability to render geometry in an ordered sequence, usually with different drawing operations like blending, depth or stencil enabled. A first but rather non-intuitive proposal for providing access to low-level rendering modes can be found in [bitmanagement 2002]. Because occlusions must be handled correctly beforehand, for doing differential rendering the virtual objects need to be rendered after the real scene, with the stencil test function set to "always". This way a mask is created at only those pixel positions ultimately containing parts of the virtual object despite the original order of incoming fragments. Therefore a fine grained control over rendering order and rendering states is needed, which is discussed in section 5.4.

After that we can go into details: Let L_{orig} be the original scene radiance provided by the background image, L_{with} the appearance with the virtual objects inserted and $L_{without}$ the appearance without the virtual objects. Then the error in the rendered scene without synthetic objects compared to the real image is $\Delta L_{err} = L_{without} - L_{orig}$. An alternative approach employs the relative error for avoiding possibly negative results. As can be seen, the better geometry and material reconstruction are, the smaller the resulting error is. By subtracting the error from L_{with}, the changes in illumination caused by inserting the virtual object (which are e.g. forming shadows if L_{with} is less than $L_{without}$) can be represented as follows:

$$\begin{aligned} L_{final} &= L_{with} - \Delta L_{err} \\ L_{final} &= L_{orig} + (L_{with} - L_{without}) \end{aligned} \tag{3}$$

By using our modified *RenderedTexture* node for rendering the different scenes, and shadow generation turned on, the values for L_{with} and $L_{without}$ are obtained for all corresponding pixels of the original image. In combination with our slightly modified *LayoutLayer* node for simplifying the rendering of a window sized view aligned quad with arbitrary appearance - a requirement well known from computer games for creating special visual effects by means of a post processing step in image space - the combination of all renderings to form the final output image is now very easy. In an after effects layer these three textures are combined in the following GLSL shader program for maximum performance.

```
void main()
{
 vec4 vir=texture2D(with,gl_TexCoord[0].st);
 vec4 rea=texture2D(sans,gl_TexCoord[0].st);
 vec4 img=texture2D(orig,gl_TexCoord[1].st);
 gl_FragColor = img + vir - rea;
}
```

Because reconstruction is always inaccurate and hence $\Delta L_{err} \neq 0$, in the final imaging layer, the texture containing the real scene with the augmentation is rendered in such a way that it simply replaces

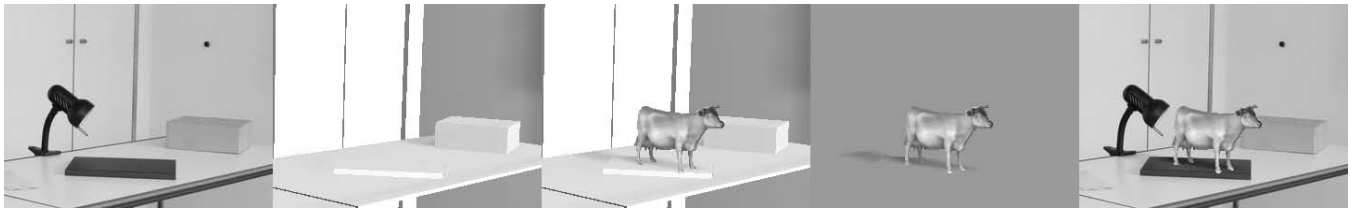

Figure 5: Steps in differential rendering (from left to right): Original image, reconstructed scene without virtual object, scene with virtual object, difference image with mask (for illustration purposes an offset is added, so that zero is grey), final augmented image.

the virtual objects by means of a stencil buffer mask by setting the stencil function int the *StencilMode* node to "keep".

5 X3D Integration

In this section we explore how suitable X3D and W3C technologies can be utilized as an application model for AR environments. In addition we are presenting and discussing X3D-extensions necessary to facilitate AR applications in general and particularly in support of advanced lighting methods. Those proposed nodes allow application developers to integrate data-streams, advanced cameras and shadows. Further, the process of rendering different layers of information (e.g. video-images and 3D-annotations) with and without the proposed extensions is discussed.

5.1 Sensors for data-streams

Sensors are a key concept in X3D. They are the scene graph elements that make the scene dynamic - every change in the scene graph is triggered by sensors. Unfortunately, the sensors currently specified in X3D are quite limited - there are *TimeSensors* that allow one to control animations, as well as sensors that allow user interaction with the scene (see key and pointing device components). The specification was written with simple 2D mouse interaction in mind and for this reason lacks support for typical AR devices like cameras, tracking systems and so on.

Currently, there are some proposals about how to integrate devices into X3D scenes. They all have their specific advantages and disadvantages, and finding a common denominator seems to be difficult. [Polys and Ray 2006] gives an overview over the discussion. In [Behr et al. 2004] we proposed an extension to X3D that allows the integration of arbitrary data streams into X3D scenes. In our system, we have special sensor nodes for each of the X3D data types (SFBool, SFFloat, etc.). The interface of these nodes looks like this (replace "x" by the concrete data type):

```
xSensor : X3DDirectSensorNode {
  x        [in,out] value
  SFBool   []       out   FALSE
  SFString []       label ""
}
```

For example, to receive images from a video camera, we use an *SFImageSensor* node that has the interface shown below. The exposed field "value" allows the reception of data values from devices or the sending of data values to devices. The "out" field specifies the direction of the sensor - whether it is used to receive or send data values. The "label" field specifies what the data stream is used for in the scene. For example, when we have a *SFVec3fSensor* node that receives the head position from the tracking system, the label could be "Head position". It is important to realize that we do not specify the device we want to use in the label - the developer of a X3D application is simply not able to foresee which devices are

```
DEF cam Viewpoint { ... }
DEF headPos SFVec3fSensor
  {label "Head Position"}
DEF headRot SFRotationSensor
  {label "Head Rotation"}
ROUTE headPos.value_changed
   TO cam.set_position
ROUTE headRot.value_changed
   TO cam.set_orientation
Shape {
  appearance Appearance {
    DEF surfaceTex PixelTexture {}
  }
  geometry IndexedFaceSet { ... }
}
DEF frame SFImageSensor
  {label "Video Frames"}
ROUTE frame.value_changed
   TO surfaceTex.set_image
```

Figure 6: Template of an AR application in X3D.

available when the application is run on the target system. Instead, we specify what the data is used for. The actual mapping between concrete devices and sensors is done in the user interface of the rendering system.

```
SFImageSensor : X3DDirectSensorNode {
  SFImage  [in,out] value
  SFBool   []       out   FALSE
  SFString []       label ""
}
```

This set of low level sensors for data streams allows one to create AR applications quickly using only X3D features. In many other AR systems, the central component is the video tracking system that captures the video frames, calculates the head position and orientation and puts the video image into the framebuffer. The scene graph is only used to render the virtual objects over the video image. In contrast to this approach, in our system the application (written in X3D) is the central component. It receives the current position, orientation and video image via sensors from the device management system and transfers them via routes to other nodes in the scene graph. Figure 6 for example shows the template of an AR application written in X3D.

The fact that we receive the video frames as *SFImages* allows us to put the live video as a texture on arbitrary geometrical objects. This enables us to efficiently eliminate almost all kinds of distortions generated by the optical system of the camera by mapping the video images on meshes that are distorted in a way that counteracts the distortion of the camera. Figure 7 e.g. shows how to compensate the barrel distortion of the camera by using a correlative mesh. An alternative would be to use a special shader parameterized with

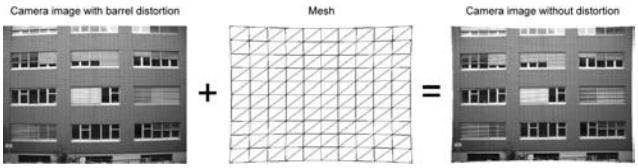

Figure 7: Efficient compensation of camera distortion by mapping the video images on correlative meshes.

the camera parameters in the ”appearance” field of the previously mentioned modified *LayoutLayer*-node.

5.2 Camera

The Viewpoint node defines a perspective view of the scene. A perspective view is one in which all projectors are coalesced on a single position in space. The ”position” and ”orientation” fields define the camera transformation and the ”fieldOfView” field specifies a preferred minimum viewing angle from this viewpoint.

This abstraction fulfills the needs of common desktop-VR based applications. Real-world cameras are far more complex and therefore AR applications require comprehensive viewing models which include the inner and outer orientation of the device[Hartley and Zisserman 2000]. Hence, we provide an extension to the *Viewpoint* node and additionally a new *Viewfrustum* node, which give the application developer more freedom.

```
Viewpoint : X3DViewpointNode {
  ...
  SFVec2f [in,out] principalPoint 0 0
  SFFloat [in,out] fovAspectRatio 1.0
}
```

The two new fields provide a more general camera model than the standard *Viewpoint*. The ”principalPoint” field defines the relative position of the principal point. If the principal point is not equal to zero, the viewing frustum parameters (left, right, top, bottom) are simply shifted in the camera’s image plane. A value of x=2 means the left value is equal to the default right value. A value of x=-2 means the right value is equal to default. If the principal point is not equal to zero, the ”fieldOfView” value is not equal to the real field of view of the camera, otherwise it complies with the default settings.

The field ”fovAspectRatio” defines the aspect ratio for the viewing angle defined by the ”fieldOfView” range. This setting is independent of the current aspect ratio of the window, but reflects the aspect ratio of the actual capturing device. This extension allows us to model cameras with a non-quadratic pixel format.

In addition to the *Viewpoint*-extension we include a new camera node named *Viewfrustum*. With the *Viewfrustum* node we are able to define a camera position and projection utilizing a standard projection/modelview matrix pair. This encoding is quite common in today’s online visualization and tracking systems[Chronos 2006] and eases the integration of existing systems.

```
Viewfrustum : X3DViewpointNode {
  ...
  SFMatrix4f [in,out] modelview  (identity)
  SFMatrix4f [in,out] projection (identity)
}
```

5.3 Layering

AR-applications in general require methods that render different layers of information - at least some sort of video-stream as background and 2D- and 3D-annotations. Even more complex layering techniques with compositing methods are needed to implement the above proposed differential rendering algorithms.

The current X3D ISO standard does not really support layers but only very simple and rigid background and foreground settings. The *BackgroundBindable*-nodes are always sky-boxes and therefore not really useful for our purpose. The image which is synthesized while rendering the scene defines the foreground. No additional layers exist. There are ways to simulate 2D-Overlays, e.g. HUDs, with the *ProximitySensor* node, but they are very limited and are not image-based methods at all.

The proposed Spec Revision 1 (CD, Jul 2006) [Web3DConsortium 2006] includes two new components, Layering and Layout, which provide nodes and functionality to render and layout different scene-parts in different layers. The *Layer* and *LayerSet* nodes define the sub-trees and rendering order but do not define what kind of composition method is used. The spec only defines that the layers will be rendered ”on top” of each other. For our proposed environment, both following extensions are essential. We need window-sized and view-aligned quads and additionally some way to control the composition method used. The first requirement can be fulfilled by using a slightly extended *LayoutLayer* node from the Rev1 specification. For the second requirement we introduce novel *X3DAppearanceChildNode* types to control the color-/ stencil-/ depth-buffer writing and merging.

These extensions can be used to perform a wide number of different multi pass methods. Multi pass techniques in general are not only necessary for differential rendering but they are also useful for all image space rendering operations (for example rendering to texture space, and writing normals or depth into a backbuffer or texture for accessing and evaluating e.g. neighboring information in an appropriate shader). In combination with our modified ”LayoutLayer” node as described before, it is also a very powerful method for doing post processing effects like blur or gloom well known from games (see Figure 8 for some examples).

As already mentioned we are using an extended *RenderedTexture* node (see below) in order to provide the ability for offscreen rendering including associated buffers like the depth buffer. Our modified *RenderedTexture* is derived from the *X3DEnvironmentTextureNode* and has a SFBool field called ”depthMap”, which allows the automatic generation of depth maps for e.g. additional user created shadows. Because this is only useful in combination with appropriate transformation matrices, the projection (modelview projection matrix of camera space) and viewing fields (model matrix of parent node) are added. Using offscreen buffers has the additional advantage that the creation of floating point textures can be forced. This not only allows doing shading calculations with higher precision but also allows HDR rendering, especially in combination with support for special HDR formats like OpenEXR and Radiance.

```
RenderedTexture : X3DEnvironmentTextureNode{
  SFNode     []       textureProperties NULL
  SFString   [in,out] update "NONE"
  MFNode     []       excludeNodes []
  SFNode     [in,out] viewpoint NULL
  SFNode     [in,out] background NULL
  SFNode     [in,out] fog NULL
  SFNode     [in,out] scene NULL
  MFInt32    [in,out] dimensions [128 128 4]
  SFBool     [in,out] depthMap FALSE
```

```
  SFMatrix4f [out]    projection identity
  SFMatrix4f [out]    viewing identity
}
```

However it is not possible to use sensors or even to navigate directly within an offscreen buffer. Furthermore, for special effects one often just needs a texture containing the framebuffer content. For that purpose we provide the *TextureGrabLayer* node with an *SFNode* field "texture", which - depending on its position in the layers field - simply contains the grabbed framebuffer. Here any *X3DTexture2DNode*, which automatically is resized when the viewport size changes, can be used for later re-USE. An *SFImage* outslot is unappropriate because in this case the texture first has to be transferred back to the CPU.

```
TextureGrabLayer : X3DLayerNode {
  SFBool [in,out] isPickable FALSE
  SFNode [in,out] viewport   NULL
  SFNode [in,out] texture    NULL
}
```

Figure 8: Post processing: engine hood with blur (left) and dynamic hdr glow (right).

5.4 Render state control

As a requirement, for more complex VR/AR applications the user sometimes needs control over the rendering order of different geometries as well as over low level rendering modes. So, for representing the other type of multi pass rendering, and because the *Appearance* node finally reveals how a rendered *Shape* node looks like, we extended the shape component with some new nodes for setting different render states and therewith the *Appearance* node with the appropriate fields. Because such functionalities directly map to the API of the graphics board driver, they cannot be encapsulated in protos by the user.

First we introduce the "sortKey" field with $sortKey \in \mathbb{Z}$ for defining the rendering order, what is essential in combination with e.g. alpha blending and color masking. Alternatively, one can think about a special ordering group, but this way usage is much more intuitive and is automatically correct for the whole scene graph. For better readability the "metadata" fields are omitted in the following node descriptions.

```
Appearance : X3DAppearanceNode {
  SFInt32 []       sortKey  0
  SFNode  [in,out] fillProperties  NULL
  SFNode  [in,out] lineProperties  NULL
  SFNode  [in,out] material  NULL
  MFNode  [in,out] shaders  []
  SFNode  [in,out] texture  NULL
  SFNode  [in,out] textureTransform NULL
  SFNode  [in,out] blendMode  NULL
  SFNode  [in,out] stencilMode  NULL
  SFNode  [in,out] colorMaskMode  NULL
  SFNode  [in,out] depthMode  NULL
}
```

Additionally we propose an *AppearanceGroup* node which extends the *Group* node with an "appearance" field. This is quite useful if a whole group of *Shape* nodes should share the same material properties. In the following, a few nodes for allowing finer control over the rendering modes are shown. If the corresponding fields in the *Appearance* node are not set by the user, the browser uses standard settings which fit best for the current material, texture or shader. Otherwise the state modes, for example the *StencilMode* as mentioned earlier, override the default settings. A short outline of the proposed node interfaces shall conclude this proposal.

As the name implies, the *BlendMode* node allows access to blending and alpha test. The field values, for instance "src_alpha" and "one_minus_src_alpha" for standard alpha blending, map directly to the corresponding rendering state names. The fields "alphaFunc" and "alphaFuncValue" specify the conditions under which a fragment is drawn or discarded. With "alphaFunc" set to "lequal" and a given reference value c the fragment passes if the incoming alpha value is less than or equal to c. Concerning choice and naming conventions, a common subset of the OpenGL and DirectX graphics standards, which have already been very well documented (e.g. [Chronos 2006]), was chosen.

```
BlendMode : X3DAppearanceChildNode {
  SFString [in,out] srcFactor  "one"
  SFString [in,out] destFactor "zero"
  SFColor  [in,out] color 1 1 1
  SFFloat  [in,out] colorTransparency 0
  SFString [in,out] alphaFunc "none"
  SFFloat  [in,out] alphaFuncValue 0
}
```

The other three modes, *StencilMode*, *ColorMaskMode* and *DepthMode*, are likewise almost self-explanatory. The *ColorMaskMode* permits control over color masking (the color channel is written if the corresponding mask field is true), and with the *DepthMode* depth functions can be set (especially useful in combination with the previously introduced "sortKey" field and the "solid" field of the *X3DComposedGeometryNode*). The *StencilMode* was already mentioned in section 4 and permits fine grained control over stencil bit masks and functions. For all fields with default values equal to -1 or "none", implementation specific default values are used. This way, complex multi pass appearances as needed for advanced rendering can be created. Moreover, our extensions of the *X3DAppearanceChildNode* are also useful for other appliances (such as for instance image based CSG operations).

```
StencilMode : X3DAppearanceChildNode {
  SFString [in,out] stencilFunc "none"
  SFInt32  [in,out] stencilValue 0
  SFInt32  [in,out] stencilMask  0
  SFString [in,out] stencilOpFail  "keep"
  SFString [in,out] stencilOpZFail "keep"
  SFString [in,out] stencilOpZPass "keep"
  SFInt32  [in,out] bitMask  -1
}

ColorMaskMode : X3DAppearanceChildNode {
  SFBool [in,out] maskR TRUE
  SFBool [in,out] maskG TRUE
  SFBool [in,out] maskB TRUE
  SFBool [in,out] maskA TRUE
}

DepthMode : X3DAppearanceChildNode {
```

```
  SFBool   [in,out] enableDepthTest TRUE
  SFString [in,out] depthFunc "none"
  SFBool   [in,out] readOnly FALSE
  SFFloat  [in,out] zNearRange -1
  SFFloat  [in,out] zFarRange  -1
}
```

5.5 SphericalHarmonicsGenerator

The *SphericalHarmonicsGenerator* node is a special texture node that generates an irradiance map from another spheremap that is given as input texture. Furthermore, with this node we introduce the concept of *DependentTextureGenerator* nodes, which are textures, whose pixel-values depend and are controlled by other textures. This new irradiance map can then be used, via sphere mapping, to simulate reflected light from an object with a certain material type. The original spheremap is a transformed fish eye image or a quadratic image of a mirror-like sphere and contains the reflection of the surrounding scene. The corresponding parameter is called ”irradianceMap”.

Two other parameters are needed for the spherical harmonic implementation to work: The number of bands that will be integrated, and that determine the type of irradiance map that will be generated, which is identified by the parameter ”numBands”; and the number of samples that are used to do the monte-carlo integration, which is identified by the parameter ”numSamples”.

As soon as the *SphericalHarmonicsGenerator* node has finished calculating the coefficients, they are available through the ”coefficients_changed” slot. Another slot, which has been of special interest to our mixed reality implementation, is called ”ambient_changed”: The very first coefficient is computed with the first spherical harmonic function, which is constant. Therefore, the coefficient can be used to determine the ambient brightness from a spheremap. This has one major application: To adjust the intensity of shadows, the overall brightness inside a scene has to be known upfront. If the surrounding scene is very bright, the statistical intensity of the shadow is low and vice versa. The slot returns a value scaled to $[0; 1]$ if the ambient brightness changes, with 0 being totally dark and 1 being totally bright. This value can then be used to adjust the shadow’s intensity by routing it to the new ”shadowIntensity” field of a *Light* node.

```
SphericalHarmonicsGenerator :X3DTextureNode{
  SFNode  []       textureProperties NULL
  SFImage [in,out] irradianceMap NULL
  SFInt32 [in,out] numBands    3
  SFInt32 [in,out] numSamples  1000
  MFFloat [out]    coefficients_changed
  SFFloat [out]    ambient_changed
}
```

5.6 Shadow Extensions

The current X3D spec does not include any kind of shadow for dynamic scenes. The previous proposals for shadow extensions are limited in the types of lights or methods supported. Our extension has two major advantages. First of all it works with every type of scene and is very intuitive. We do not introduce new nodes but extend the existing light nodes with additional fields. Therefore the light-node regulates illumination and shadows simultaneously, just like in the real world. Second, our parameter and abstraction level allows the support and implementation of different methods, e.g. shadow-map or shadow-volumes, to fulfill different requirements.

```
X3DLightNode : X3DChildNode {
  ...
```

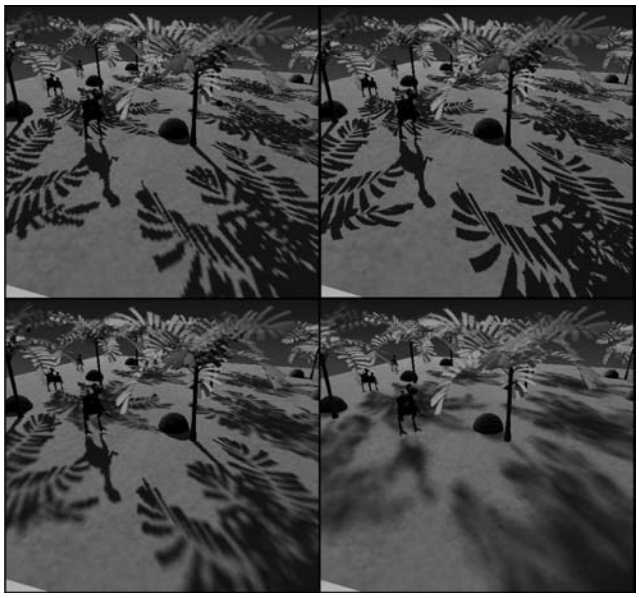

Figure 9: Different shadow modes: fastHardShadow, niceHardShadow, fastSoftShadow and niceSoftShadow.

```
  SFFloat [in,out] shadowIntensity 0 [0,1]
  SFFloat [in,out] resolution 0.5    [0,1]
  SFFloat [in,out] mode "auto"
}
```

The ”shadowIntensity” defines the intensity of the shadows. If the field value is equal to 0, the shadow is off, otherwise it defines the shadow opacity. The ”resolution” is a mode-dependent scale factor, which determines the tradeoff between speed and quality: 1 equals best and 0 the fastest results for the given mode.

The ”mode”-field defines the shadow method. All run-time environments should at least support the following values: ”auto”, ”fastHardShadow”, ”niceHardShadow”, ”fastSoftShadow, and ”niceSoftShadow”. Whereas ”auto” activates the default browser settings, i.e. no shadows, all other modes are implementation dependent. We only assume that the given order reflects the increasing complexity of the method. ”fastHardShadow” should be the fastest and ”niceSoftShadow” should offer the best quality shadows. In our implementation, for example, we map the previously mentioned PCF-method for ”fastSoftShadow”-modes and the modified PCSS for ”niceSoftShadow”-modes. Standard shadow maps correspond to the ”fastHardShadow”-mode and LISP shadows are mapped to the ”niceHardShadow”-mode.

6 Results

Our test implementation of the proposed extensions was conducted using the Avalon system [Behr et al. 2004]. The Avalon framework is a VR/AR middleware which utilizes OpenSG [OpenSG 2006] for rendering and a super-set of X3D nodes as application description language. Tests and images were generated on a standard Intel P4/2.4 GHZ PC equipped with a NVidia 6600 GT graphics board.

Image 10 shows an application using the *Viewfrustum* node in combination with a *DirectSensor* node for augmenting the real model with a virtual flow field running at interactive frame rates. For making the creation of such applications easier, we have developed a plug-in based framework on top of our Avalon system called Composer for editing scenes. The left window shows the tracking

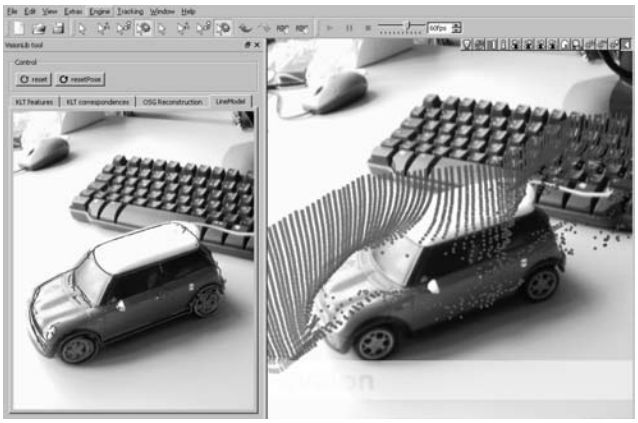

Figure 10: A car augmented with a dynamic flow field (the left image shows the tracking view).

Figure 11: An MR application merging the dragon into the scene in real time.

plug-in used for data integration. The tracking system which we connected via the here proposed sensor nodes was developed by members of our group and uses an approach based on a combination of line tracking for initialization (which internally also needs a geometric model of the real scene), and KLT point features for frame to frame tracking as described in [Bleser et al. 2006].

The image displayed in figure 11 combines all techniques presented in this paper in one X3D application to merge virtual and real environments. The well known dragon model is rendered in a 1500×1000 pixel context with $8 \times$ FSAA, a spherical harmonic analysis of the incoming lighting data with 9 coefficients, a mixture of 25% diffuse and 75% specular lighting, static ambient occlusion and PCF shadows. The PCF shadow intensity is set via the "ambient_changed" outslot of the *SphericalHarmonicsGenerator*. The image is afterwards combined with the background image through differential rendering as described in section 4 and drawn at 10 frames per second. For less complex models dynamic ambient occlusion can be enabled without major performance hits, even though depending heavily on the sampling rate.

Figure 12 shows the Archeoguide system, an example of an outdoor AR system. The X3D scene consists of three different layers: The video in the background, the 3D reconstruction of a temple that does not exist anymore, and the user interface. The structure of the scene graph is similar to the template shown in 6. The application logic is written in Java/ Javascript using standard X3D Script nodes.

Figure 12: Archeoguide - example of an outdoor AR system.

7 Conclusion

In this paper we explored and discussed how far the current X3D standard could be utilized for advanced augmented and mixed reality applications. Currently X3D has some major limitations concerning the integration of external devices like cameras and tracking systems. Furthermore the virtual camera handling follows a simple pinhole camera model which is not sufficient for AR applications. Finally the local lighting model in X3D does not support advanced rendering methods which include shadows, general multi pass techniques and render state control.

To overcome these limitations we presented various enhancements to the present X3D ISO standard. These comprise a set of data stream sensor nodes and extensions to the viewpoint node on the one hand and extensions for advanced rendering techniques like irradiance mapping, shadows, and multi pass techniques on the other hand. Designing the new nodes and interfaces we focused on three main goals: The first is to be consistent with the ideas of the current specification, the second one is extensibility for future enhancements. And last but not least, our proposed node extensions allow application developers to create complex photo-realistic mixed reality environments easily.

Although the proposed shadow extensions are fast and stable and work well in practice, the major drawback of the presented shadowing algorithms is the fact, that transparencies are not handled correctly and therefore omitted during the shadow pass. This might be okay for a window pane but not for glass bricks or other semi transparent objects like sunglasses. Although there exist approaches for special applications based on scene knowledge there exists no general solution for arbitrary scenes yet. In combination with differential rendering this is an even bigger issue because parts of the real scene which are behind transparent objects are composed wrong. Another problem arises from the approach of using special shaders for the shadow calculations: user defined shader programs are ignored in the shadow pass, even those which transform vertices in world space or discard fragments, and therefore the shadows are calculated wrong.

Although the idea to split the irradiance map calculations into a CPU and a GPU part, in which the final lighting reconstructions is done, is very fast and leads to convincing results, it has the disadvantage that it only works in combination with this specially designed shader and therefore the original material properties of the virtual objects, especially if they already contain shaders, are mostly ignored. Another challenge, which is left as future work, arises in using the layer component for doing differential rendering and the like in the context of VR, e.g. CAVE and PowerWall, where stereoscopic projection and multiple viewports demand an extended concept of layout and layering.

References

BEHR, J., DÄHNE, P., AND ROTH, M. 2004. Utilizing x3d for immersive environments. In *Web3D '04: Proc. of the ninth int. conf. on 3D Web technology*, ACM Press, NY, USA, 71–78.

BITMANAGEMENT, 2002. Drawgroup and drawop, dec. http://www.bitmanagement.de/developer/contact/examples/multitexture/drawgroup.html.

BLESER, G., WUEST, H., AND STRICKER, D. 2006. Online camera pose estimation in partially known and dynamic scenes. In *ISMAR 2006 : Proc. of the Fourth IEEE and ACM Int. Symposium on Mixed a. Augmented Reality*, IEEE Computer Society, Los Alamitos, Calif., 56–65.

CHRONOS. 2006. *OpenGL Man Pages*. http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/.

DEBEVEC, P. E., AND MALIK, J. 1997. Recovering high dynamic range radiance maps from photographs. In *Proceedings of SIGGRAPH 97*, CG Proc., Annual Conference Series, 369–378.

DEBEVEC, P. 1998. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proc. of SIGGRAPH 98*, CG Proc., Annual Conf. Series, 189–198.

DRETTAKIS, G., ROBERT, L., AND BOUGNOUX, S. 1997. Interactive common illumination for computer augmented reality. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, Springer-Verlag, London, UK, 45–56.

FERNANDO, R. 2005. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, ACM Press, New York, NY, USA, 35.

GIBSON, S., AND CHALMERS, A., 2003. Photorealistic augmented reality. Eurographics 2003 Tutorial, September.

GIBSON, S., COOK, J., HOWARD, T., AND HUBBOLD, R. 2004. Aris: Augmented reality image synthesis. Tech. Rep. IST-2000-28707, University Manchester, July.

GREEN, R. 2003. Spherical harmonic lighting: The gritty details. *Archives of the Game Developers Conference* (Mar.).

HARTLEY, R., AND ZISSERMAN, A. 2000. *Multiple View Geometry in Computer Vision*. Cambridge Univ. Press.

KING, G. 2005. Real-time computation of dynamic irradiance environment maps. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar., ch. 10, 167–176.

KORN, M., STANGE, M., VON ARB, A., BLUM, L., KREIL, M., KUNZE, K.-J., ANHENN, J., WALLRATH, T., AND GROSCH, T. 2006. Interactive augmentation of live images using a hdr stereo camera.

LOSCOS, C., FRASSON, M.-C., DRETTAKIS, G., WALTER, B., GRANIER, X., AND POULIN, P. 1999. Interactive virtual relighting and remodeling of real scenes. In *Rendering techniques (Proc. of the 10th EG Workshop on Rend.)*, 235–246.

NG, R., RAMAMOORTHI, R., AND HANRAHAN, P. 2004. Triple product wavelet integrals for all-frequency relighting. *ACM Trans. Graph. 23*, 3, 477–487.

OCTAGA, 2006. Octaga - bringing enterprise data to life. http://www.octaga.com/.

OPENSG, 2006. Opensg. http://www.opensg.org.

PILET, J., GEIGER, A., LAGGER, P., LEPETIT, V., AND FUA, P. 2006. An all-in-one solution to geometric and photometric calibration. In *Int. Symposium on Mixed and Augmented Reality*.

POLYS, N. F., AND RAY, A. 2006. Supporting mixed reality interfaces through x3d specification. In *Workshop at the IEEE Virtual Reality*.

POMI, A., AND SLUSALLEK, P. 2004. Interactive Mixed Reality Rendering in a Distributed Ray Tracing Framework. In *IEEE and ACM Int. Symp. on Mixed a. Augmented Reality ISMAR 2004*.

RAMAMOORTHI, R., AND HANRAHAN, P. 2001. An efficient representation for irradiance environment maps. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 497–500.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proc. of the 14th annual conference on CG and interactive techniques*, ACM Press, NY, USA, 283–291.

SATTLER, M., SARLETTE, R., ZACHMANN, G., AND KLEIN, R. 2004. Hardware-accelerated ambient occlusion computation. In *Vision, Modeling, and Visualization 2004*, B. Girod, M. Magnor, and H.-P. Seidel, Eds., 331–338.

SHASTRY, A. S., 2005. Soft-edged shadows. http://www.gamedev.net/reference/articles/article2193.asp.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*, 3 (July), 527–536.

SLOAN, P.-P., LUNA, B., AND SNYDER, J. 2005. Local, deformable precomputed radiance transfer. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM Press, New York, NY, USA, 1216–1224.

SUDARSKY, S. 2001. Generating dynamic shadows for virtual reality applications. In *IV '01: Proceedings of the Fifth International Conference on Information Visualisation*, IEEE Computer Society, Washington, DC, USA, 595.

SUPAN, P., AND STUPPACHER, I. 2006. Interactive image based lighting in augmented reality.

WEB3DCONSORTIUM. 2006. *Extensible 3D (X3D) ISO/IEC CD 19775-1r1:200x*. http://www.web3d.org/x3d/specifications/ISO-IEC-19775-X3DAbstractSpecification_Revision1_to_Part1/.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004)*, A. Keller and H. W. Jensen, Eds., 143–151.

XJ3D, 2004. Xj3d dynamic texture rendering ext. http://www.xj3d.org/extensions/render_texture.html.

Enhancing X3D for advanced MR appliances

Figure 1: Different materials and indirect lighting simulated through irradiance mapping.

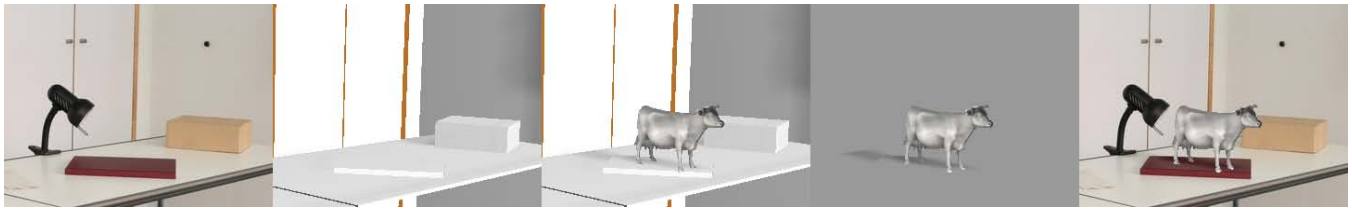

Figure 2: Steps in differential rendering.

Figure 3: System setup of our MR simulation.

Figure 4: Dragon merged into real scene.

Figure 5: Debevec's grace probe (right) reconstructed with 4 and 11 coefficients.

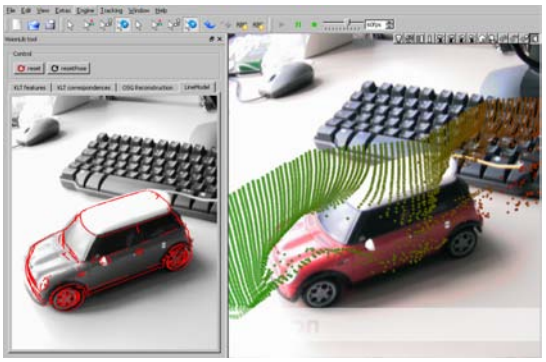

Figure 6: A car augmented with a dynamic flow field.