Enhancing Realism of Mixed Reality Applications through Real-Time Depth-Imaging Devices in X3D

Tobias Franke* Svenja Kahn† Manuel Olbrich‡ Yvonne Jung§

Fraunhofer Institut für Graphische Datenverarbeitung
Darmstadt, Germany

Abstract

Until recently, depth sensing cameras have been used almost exclusively in research due to the high costs of such specialized equipment. With the introduction of the Microsoft Kinect device, real-time depth imaging is now available for the ordinary developer at low expenses and so far it has been received with great interest from both the research and hobby developer community. The underlying OpenNI framework not only allows to extract the depth image from the camera, but also provides tracking information of gestures or user skeletons. In this paper, we present a framework to include depth sensing devices into X3D in order to enhance visual fidelity of X3D Mixed Reality applications by introducing some extensions for advanced rendering techniques. We furthermore outline how to calibrate depth and image data in a meaningful way through calibration for devices that do not already come with precalibrated sensors, as well as a discussion of some of the OpenNI functionality that X3D can benefit from in the future.

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

Keywords: X3D, Augmented Reality, Mixed Reality, Rendering

1 Introduction

With the introduction of Microsoft's Kinect depth imaging device and the subsequent release of the OpenNI framework together with the PrimeSense NITE SDK [OpenNI 2011] to enable user tracking and gesture recognition, the Mixed Reality (MR) and electronic art performance scene saw a sudden increase in installations and demos using these devices. Blog pages such as "kinecthacks.net" display a wide range of hobbyist applications ranging from gesture-enabled controllers for already existing games to playful art experimentation or even robots. Most of these applications make use of toolkits such as openFrameworks or Cider to create graphical output and require considerable programming skills to create even the simplest applications. We believe this kind of application development to be inside the domain of X3D [Web3DConsortium 2008b]. Cheap depth sensing cameras are now available from different vendors and with the underlying support of the OpenNI standard they are able to provide tracking data for Natural Interaction (NI). However, even without NI sensing data, the depth image alone is the basis for several effects in any MR application.

In this paper we discuss several enhancements for X3D to enable the use of all OpenNI compatible devices and general depth cameras in section 3. We provide an overview of a generic subsystem that supports MR interaction with X3D and how non-calibrated depth cameras can be configured to match up their data with a given background image in section 4. In the following section 5, we analyze several effects for MR applications, which can be achieved with a depth image of the real scene. Ultimately, MR application writers will benefit from the X3D standard and faster development.

2 Related Work

Previous approaches used Time-of-Flight (ToF) cameras for real-time depth imaging [Oggier et al. 2006]. ToF cameras emit near-infrared modulated light which is reflected by the scene. The phase shift of the reflected light is used to calculate the distance to the measured objects. An overview on ToF sensors and applications which use real-time depth imaging is given in [Kolb et al. 2009].

[Schiller et al. 2010] provide a starting framework for incorporating a ToF camera into a Mixed Reality system. The captured depth map of the scene is used to create a background model of the scene with the background image being used as a texture. The reconstructed scene is used for shadow mapping, and light sources are inferred from the background image or placed manually. Viewed from the camera's original angle, shadows are inserted into the image with the correct topological distortion. Occluding real geometry is handled by comparing depth values from the camera's image with the rendered objects.

[Kahn et al. 2010] propose an Augmented Reality system which uses real-time depth imaging for 3D discrepancy checks. They couple a ToF camera and a color camera on a rigid camera rig and use camera tracking algorithms to estimate the positions of both cameras while the user moves the rig around an object. The depth images of the ToF camera are used to compare the surface of the real object with a 3D model of the object. The Augmented Reality (AR) visualization of the differences assists the user in detecting differences between the real object and the 3D model, which for instance is helpful for maintenance tasks.

Visualization techniques for advanced AR/MR rendering especially in the context of X3D were presented in [Jung et al. 2007] and [Lee 2009]. Both proposals presented nodes for integrating live camera streams into X3D. Both papers also discussed MR visualization problems like occlusions (which in [Lee 2009] were only tackled on a rather primitive level by extending the *Shape* node with an `isGhost` flag) and MR relighting. In [Jung et al. 2007] the latter is solved with a combination of image-based rendering and differential rendering by also introducing some shadowing extensions, whereas in [Lee 2009] this topic is left open for future work.

Shadow mapping, due to its requirement of obtaining the depth values from the camera and light perspective, integrates well with depth images for real scenes. In the remainder of this section we

*email:tobias.franke@igd.fraunhofer.de
†email:svenja.kahn@igd.fraunhofer.de
‡email:manuel.olbrich@igd.fraunhofer.de
§email:yvonne.jung@igd.fraunhofer.de

will review previous approaches of integrating shadowing techniques into X3D. For an introduction and comprehensive overview on real-time shadows in general we refer to [Akenine-Möller et al. 2008, p. 331 ff.]. The X3D lighting model [Web3DConsortium 2008b] still does not include any form of real-time shadows. However, there have already been some proposals to include real-time shadows into X3D. For instance in [Sudarsky 2001] shadows for dynamic scenes are generated, but only on user defined surfaces and for point lights that must be outside of the current scene. The *Shadow* node extension from Octaga [Octaga 2009] supports all X3D light types and provides a `detail` field parameter that 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 very natural.

BS Contact [Bitmanagement 2002] provides shaders, which can be used to generate shadows, but the 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 can handle and would intuitively think of. Kamburelis presented another set of node extensions for integrating shadow maps and projective texture mapping into X3D [Kamburelis 2010]. However, the proposed extensions only deal with simple standard shadow mapping and, on a rather primitive level, PCF shadows [Reeves et al. 1987]. Furthermore, the proposed shader integration is very low-level and some of the proposed extensions even seem to be mutual contradictory. Nevertheless, despite their drawbacks some of the proposed extensions might ease improving rendering quality a lot and should be simple to use, since e.g. the new *GeneratedShadowMap* node in combination with the *TextureCoordinateGenerator* extension for calculating projective texture coordinates is rather close to the X3D design principles.

3 DepthSensor and accompanying Nodes

In [Behr et al. 2004], an interface specification for external devices interacting with X3D was proposed. These node types are descendants of the `X3DDirectSensorNode` [Behr et al. 2005] and specify the data to be received, not the device itself. In line with this previous discussion, we present a `DepthSensor` to receive depth images from any camera type the underlying system is able to provide, and an `NISensor` for OpenNI-supported devices.

The `DepthSensor`, whose interface is shown next, provides a basic SFImage of a ToF camera or similar device. On many devices, this depth data is represented with either 16 bit unsigned or float values, implying equivalent data type support in X3D images to directly use the same information on the GPU.

```
DepthSensor : X3DDirectSensorNode {
  SFImage   [out]     depthImage
  SFInt32   [in,out] width           640
  SFInt32   [in,out] height          480
  SFInt32   [in,out] fps             30
  SFBool    [in,out] normalizeDepth TRUE
  SFString  []       label           ""
}
```

On top of it, the `NISensor` provides basic fields that can be accessed through OpenNI production nodes.

```
NISensor : DepthSensor {
  ...
  SFImage   [out]     colorImage
  SFBool    [in,out] alignViewports TRUE
  MFInt32   [out]     users
}
```

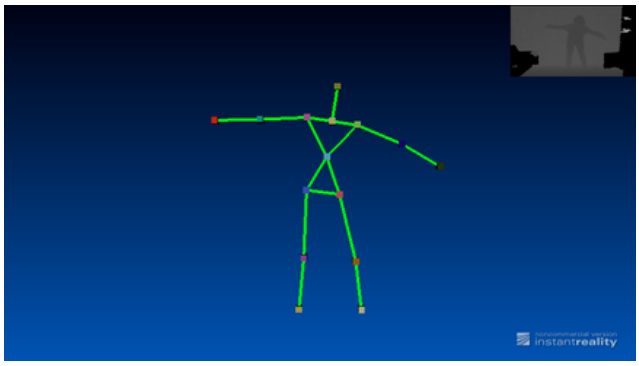


Figure 1: *A skeleton is tracked from the depth image of a Kinect device inside InstantReality [IR 2010].*

Several fields are used to configure the node upon instantiation: `width`, `height` and `fps` are parameters of the SFImage outslots returned from the `NISensor` node (i.e. `colorImage` and `depthImage`). The field `alignViewports` is a boolean value to automatically transform either the colorImage or the depthImage into the viewing space of each other to compensate for the parallax distortion if both cameras are set apart. OpenNI provides such functionality, but manual calibration may be beneficial for other devices. `normalizeDepth` is used to transform depth values into scaled floating point values from $[0.0, 1.0]$ instead of the raw range and data type provided by the device. For user tracking, a field `users` can provide a list of ID's assigned to each person tracked by the device. To track several users inside a scene, a special `UserSensor` node provides the data for each individual ID.

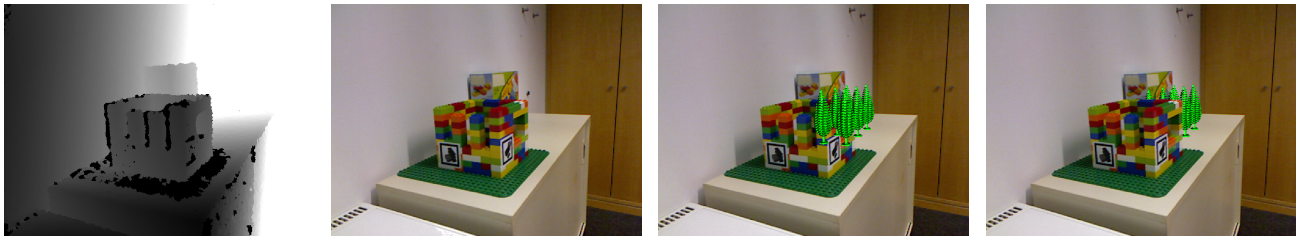
```
UserSensor : X3DDirectSensorNode {
  ...
  SFInt32     [in,out] userID            0
  SFImage     [out]    userMask
  MFVec3f     [out]    jointPositions
  MFRotation  [out]    jointOrientations
  SFString    [out]    gesture
}
```

The node is identified through its `userID`, which is assigned to a tracked person. If no such person exists, the `userID` can be reset to a standard value. A binary image `userMask` is provided to identify the pixels that have been assigned to the same user ID, which can be used for stencil operations. The eventOut slot `gesture` provides a basic string of a recognized gesture or pose from the user. Detailed information about the gesture is not handled inside OpenNI. Instead, OpenNI refers to proprietary middleware, which at this moment is not yet standardized. However, a discussion for specific gesture sensors could be relevant for future work.

Skeleton tracking is achieved with the help of two eventOut fields: `jointPositions` and `jointOrientations`, as can be seen in figure 1. These values can be used for character animation, though in our tests we noticed that the `jointOrientations` provided by OpenNI are too unstable for this purpose. Another problem is the simplification of the reconstructed skeleton, which has far less joints than a common rigged character. To animate complex H-Anim characters [Web3DConsortium 2008a], we used CCD[1] to calculate joint orientations based on the recognized joint positions. This approach allows stable animations based on the skeleton information also for skinned H-Anim figures.

[1] Cyclic Coordinate Descent – an inverse kinematics approach that solves the IK problem through optimization [Canutescu and Dunbrack 2003].

(a)



(b) (c) (d) (e)

Figure 2: *(a) Rotated 3D view with virtual trees. The depth measurements were transformed to the world coordinate system and mapped with the colors of the 2D camera. Left: Calculated camera positions (orange: color camera, blue: depth camera). (b) Raw depth image. At black pixels the Kinect depth camera could not acquire depth values. (c) Raw color image. The square markers on the brick model are used to track the camera pose. (d) Color image augmented with virtual trees (without occlusion) (e) Correct occlusion handling by mapping the depth measurements on the color image.*

4 Computer Vision

This section describes the methods and mathematical foundations for using real-time depth imaging for Mixed Reality applications. First the camera parameters need to be estimated with camera calibration algorithms. The camera parameters are prerequisites for camera tracking algorithms which estimate the camera pose in real-time. Camera tracking is the core technology of Mixed and Augmented Reality applications because the camera pose is needed to seamlessly integrate real with virtual elements by augmenting the video stream with correctly positioned virtual objects. Furthermore we specify the mathematical foundations for calculating 3D positions from the raw depth measurements and mapping them between the coordinate systems of the depth and the color camera. Mapping the depth measurements to the viewpoint of the color camera is a prerequisite for occlusion handling in the color image.

Please note that the Kinect instantiation of the OpenNI framework provides methods for automatic viewport mapping. These methods can be used to create and use Mixed Reality applications with small programming effort. The automatic viewport mapping methods use default calibration values which are not calculated for the specific Kinect device. Nevertheless, in our experience they provide a good approximation. The occlusion or shadow accuracy can be enhanced further by replacing the default OpenNI mapping with the equations given in this section, which use the manually calculated camera parameters of the specific device. The equations specified in this section are not restricted to Kinect depth cameras but can also be applied to any combination of depth cameras with color cameras. Therefore they make it possible to use any other depth imaging devices for Mixed Reality applications as well, for example a time-of-flight camera rigidly coupled with a 2D color camera [Kahn et al. 2010].

4.1 Camera calibration

The pinhole camera model mathematically describes the projection of real 3D objects onto camera images with the intrinsic parameters focal length (f_x, f_y), principal point (c_x, c_y) and skew (s) [Forsyth and Ponce 2002]. This basic model can be extended by distortion coefficients which account for the radial distortion of the camera lens. The estimation of the intrinsic parameters of custom 2D cameras is a state-of-the-art computer vision technique for which open source libraries [OpenCV 2009] and tools such as the GML Camera Calibration Toolbox [CalibrationToolbox 2011] are available. To calibrate a 2D camera, several images of a chessboard calibration pattern are acquired. Then the calibration algorithms detect the positions of the chessboard corners in the 2D camera images and use these 2D positions to estimate the intrinsic parameters.

The Kinect contains an infrared projector (which projects an infrared pattern onto the scene) and two cameras (a color camera and the depth camera). For realizing MR applications, both cameras

need to be calibrated intrinsically. The default calibration procedure can be used for the color camera. However, the checkerboard pattern is not visible in the depth image. Therefore we make use of the fact that the depth camera is an infrared camera (which detects the infrared pattern of the projector to calculate depth values for each pixel). Instead of acquiring the depth image, the raw infrared image of the depth camera is acquired with the OpenNI interface. By covering the infrared projector with opaque tape to remove the infrared pattern from the image and illuminating the chessboard pattern with an external infrared lamp, the chessboard becomes visible in the infrared images of the depth camera. Then the custom intrinsic calibration procedures for 2D cameras can also be applied on the infrared images of the depth camera.

Due to the fact that the Kinect device contains not only a depth camera but also a color camera it is very well suited for realizing Mixed Reality applications: Whereas the depth data provides the basis for real-time occlusion calculation and shadow mapping, the position of both cameras can be calculated by tracking the position of the 2D camera with camera tracking algorithms and then inferring the position of the depth camera from the relative transformation $(\Delta R, \Delta t)$ between the two cameras. As well as the intrinsic parameters, the relative transformation between the cameras only needs to be calculated once in an offline calibration step. The KinectAutoCalibration tool written by Engelhard [Engelhard 2010] makes use of the OpenCV library [OpenCV 2009] to calculate the relative transformation between the depth and the color camera.

4.2 Camera tracking

For a seamless integration of virtual objects into camera images the pose (position and orientation) of the camera needs to be know. The position and rotation of the camera are calculated with camera tracking algorithms. The most commonly used method for camera pose estimation is marker tracking. These fiducial markers are attached to the scene and have a contour which is easy to detect with image processing algorithms. Image 2(c) shows two square markers which are attached to the brick model. The coordinates of the markers are specified in the world coordinate system and the markers are tracked with the computer vision framework InstantVision [Becker et al. 2007]. Other publicly available tools such as ARToolKit [ART 2008] also contain easy to use marker tracking algorithms.

Marker based camera tracking has the drawback that the camera pose can only be calculated if markers are visible in the current camera image. Therefore we combine markerless camera tracking with structure from motion 3D reconstruction [Bleser et al. 2006]. While the camera is moved, this approach detects characteristic image features with a Shi-Tomasi corner detector [Shi and Tomasi 1994] and tracks these 2D image features with a Lucas Kanade tracker [Wuest 2008]. Then the 3D positions of the tracked features are reconstructed online via triangulation [Bleser et al. 2006]. Finally the 2D-3D correspondences between the 2D image coordinates of the tracked features and their reconstructed 3D coordinates are used to calculate the camera pose if no marker is visible.

When a color camera is combined with a depth camera to create more realistic MR applications (e.g. when the Kinect is used for real-time depth imaging), not only the pose of the color camera needs to be known but also the pose of the depth camera. The pose of the depth camera can not be calculated directly from the depth image because image features like markers are not visible in the depth image (see figure 2(b)). Therefore the pose of the depth camera is calculated from the tracked 2D image of the color camera with equation 1. $(\Delta R, \Delta t)$ is the relative transformation between the two cameras which was calculated in the offline calibration step.

$$\begin{aligned} R_D &= R_C \cdot \Delta R \\ t_D &= R_C \cdot t_C + \Delta t \end{aligned} \tag{1}$$

4.3 Conversion from depth values to 3D points

To calculate occlusions or shadow mapping for augmentations in the color image, the depth measurements of the depth camera need to be mapped onto the color image. Therefore the raw depth values are first transformed to 3D points and then projected onto the 2D image of the color camera. Equation 2 transforms the 1D depth value d_{cam} of a pixel (p_x, p_y) in the 2D image coordinate system of the depth camera to a 3D point $p_{CCS(D)}$ in the depth camera coordinate system CCS(D).

$$p_{CCS(D)} = \begin{pmatrix} (p_x - c_x) \cdot \frac{1}{f_x} \cdot d_{cam} \\ (p_y - c_y) \cdot \frac{1}{f_y} \cdot d_{cam} \\ d_{cam} \end{pmatrix} \tag{2}$$

The origin of the camera coordinate systems is the camera center. Thus points in the camera coordinate system are specified relative to the camera position and orientation which changes whenever the camera is moved. To obtain the 3D world coordinates of 3D points (which are independent of the camera pose), the 3D points are transformed to the world coordinate system with equation 3.

$$p_{WCS} = (R_D)^{-1} \cdot (p_{CCS(D)} - t_D) \tag{3}$$

Finally equation 4 is used to project 3D points from the world coordinate system to 2D coordinates $p_{ICS(C)}$ in the camera image of the color camera. Here K_C is the camera calibration matrix of the color camera which contains the intrinsic parameters that were calculated with the intrinsic calibration procedure described in section 4.1. ICS(C) is the 2D image coordinate system of the color camera.

$$p_{ICS(C)} = K_C((R_C \cdot p_{WCS}) + t_C) \tag{4}$$

Figure 2(b) and 2(c) show the depth and the color image acquired by a Kinect camera. The raw depth measurements were transformed to the world coordinate system and finally mapped onto the color image. The 3D points in the world coordinate system and their color mapping as well as the calculated poses of both cameras are visualized in figure 2(a). Furthermore, occlusion was calculated with the depth values mapped onto the color image (which will be described in more detail in section 5.1).

4.4 InstantIO

Interfacing between input/output devices in general (and specifically the computer vision component) and the X3D-based scenegraph is not done directly, but through a layer called InstantIO [Behr et al. 2005], which provides a general sensor abstraction for IO devices. In a similar fashion to X3D, the layer provides a rich set of nodes with slots that either send or receive data to other nodes via a routing mechanism. Device nodes (or backends respectively) in InstantIO generally do not refer to a specific piece of hardware, but rather to a group of similar devices. For instance, a *video* node can provide input from a webcam, a TV device or any other video stream the underlying system detects.

We connect InstantIO, which is an integral part of the Instant Reality framework [IR 2010], to the X3D-based rendering system either with a specialized `X3DDirectSensor` node or alternatively for

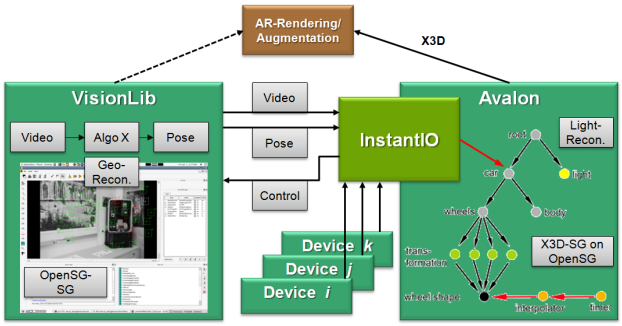


Figure 3: *Connection between tracking system (left) and rendering (right) in the Instant Reality framework via InstantIO (middle).*

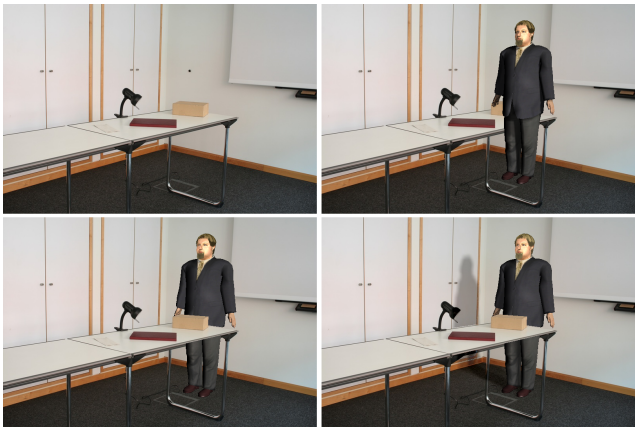


Figure 4: *From top left to bottom right: (a) real image of a room; (b) real scene augmented with virtual character (note that the character appears to be before the table); (c) augmentation with additional occlusion handling (note that the character still seems to float on the floor); (d) augmentation with occlusion and shadows.*

convenience with a sensor node called `IOSensor` initialized with a type that specifies the sort of input that is expected (e.g. video or joystick). The system dynamically maps fields to the slots provided by the InstantIO hardware abstraction, which can then be rerouted to other X3D nodes and fields. Our computer vision subsystem is handled analogously to any other IO device: as shown in figure 3, the IO system provides several (dynamic) fields that configure the tracking mechanism and allow grabbing the results such as the background video image or the intrinsic camera matrices.

5 Advanced MR Rendering

Several effects for MR simulations require reconstruction of scene elements or the viewing area from the main camera perspective, such as occlusion of virtual geometry from real objects, virtual shadow mapping onto real geometry or even simulation of physical interaction. Traditionally, this has been achieved by reconstructing a 3D model a priori and then tracking it alongside the MR application to access real scene geometry. The obvious disadvantage of this method is its inflexibility. For many use cases, a simple depth image can remove the need for prior knowledge of the scene.

5.1 Occlusions

A simple but powerful effect for MR applications is occlusion of virtual objects through the real scene. Typical MR scenes leave a weird impression on the user when the perceived depth isn't correctly reflected by the application, because the virtual object always appears on top of things (compare top right image in figure 4 vs. bottom left one). For rasterization-based rendering engines several different possibilities exist to handle occlusions, but all of them require multipass rendering [Diefenbach and Badler 1997].

The most trivial approach employs invisible "ghosting" objects, which are geometrically registered 3D models of real objects, for instance the table in figure 4. This method only requires having some basic render state control, which explicitly allows rendering the reconstructed 3D models of real objects first (which potentially occlude virtual ones) – but only into the depth buffer, since the color buffer is already correctly filled with the incoming video stream.

Hence, with a few extensions of the traditional X3D Appearance and Material nodes, which already were proposed in [Jung et al. 2007], occlusions in AR/MR scenes can be easily incorporated, given that the (real) 3D objects, for which occlusions should be handled, already exist as (X)3D model. The bottom left image in figure 4 shows an application of color masking in combination with using a defined rendering order for achieving the occlusion effects. The corresponding X3D code (in VRML encoding) is shown below.

```
Shape {
  appearance Appearance {
    sortKey -1
    colorMaskMode ColorMaskMode {
      maskR FALSE
      maskG FALSE
      maskB FALSE
      maskA FALSE
    }
  }
  geometry IndexedFaceSet { ... }
}
```

As can be seen, only two fields of the geometry's Appearance are parameterized. The *ColorMaskMode* allows writing to the depth buffer without writing any color values to the framebuffer. This node thereby permits control over color masking – the color channel is written if the corresponding mask field is true. The field `sortKey` (whose default value is "0") is explicitly set to "-1", which means that this object is rendered before all others. This is necessary to initially stamp out the depth buffer without setting the color – otherwise another object (depending on the rendering order) might already have written its color to the framebuffer.

The second approach for dealing with occlusions between real and virtual objects, as shown in figure 5, directly employs the depth map of the real scene as generated by ToF cameras or similar devices, since this nicely fits with standard rasterization techniques. First, a window-sized, view-aligned quad needs to be rendered, whose appearance contains both image streams, the depth and the color map of the real scene. As can be seen in the shader part of the following X3D code fragment, this can be easily achieved with a shader that initializes the depth buffer with the given depth map via `gl_FragDepth`, while the color buffer is initialized with the standard camera image. Real objects hence are not required to be reconstructed in advance, but – as is shown in the screenshots – can even be highly deformable objects like e.g. humans with clothes.

```
PolygonBackground {
    appearance Appearance {
       texture MultiTexture {
          texture [
```

```
            DEF depth PixelTexture {}
            DEF color PixelTexture {}
        ]
    }
    shaders [
        ComposedShader {
            exposedField SFInt32 depth 0
            exposedField SFInt32 color 1
            parts [
              ShaderPart {
                type "fragment"
                url "
                  uniform sampler2D depth;
                  uniform sampler2D color;
                  void main()
                  {
                    gl_FragDepth = texture2D(
                       depth, gl_TexCoord[0].st).r;
                    gl_FragColor = texture2D(
                       color, gl_TexCoord[0].st);
                  }
                "
              }
            ]
          }
        ]
    }
    positions [ 0 0, 1 0, 1 1, 0 1 ]
    texCoords [ 0 0 0, 1 0 0, 1 1 0, 0 1 0 ]
}
```

For convenience, we therefore propose a new background bindable node, the *PolygonBackground*, whose most important field is `appearance`, an input-output slot of type SFNode that can hold an X3D *Appearance* node. By first rendering the AR scene including its real depth, the GPU then automatically compares the rendered depth values of virtual scene geometry, such as the bandit character in figure 5, with the depth map captured by a ToF camera or similar device. In doing so, pixels of the virtual object *behind* real pixels can be filtered out to simulate real occlusion.

While the fields `positions` and `texCoords` further define the polygon on which the appearance is applied, another field called `doCleanup` allows specifying, whether the depth and stencil buffer should be cleared after applying the material. Especially helpful for AR applications are furthermore the SFVec2f field `fixedImageSize` for keeping the aspect ratio when rendering e.g. images by defining the desired ratio via *(width, height)*, as well as the `mode` field of type SFString, which defines whether the background texture fits vertically, horizontally or in the smaller direction and only takes effect if `fixedImageSize` is set.

The last approach here uses differential rendering [Debevec 1998], which is also capable of handling shadows and inter-reflections, and will be very shortly discussed in section 5.3.

An "inversion" of MR is also possible: the `UserSensor` provides a `userMask` binary image which can be used as a stencil mask to extract a person in front of the camera and insert it into a regular X3D scene. Since stencil operations and the like are still not part of the X3D specification, we refer to the corresponding Appearance node extensions as proposed in [Jung et al. 2007; IR 2010].

5.2 Lighting

Virtual light sources can be used to illuminate real objects. Depending on the lighting model, a normal vector for a real surface and its distance to the light source are already sufficient, provided material parameters of the real surface are known. One can extract a surface normal for a pixel of reconstructed position p and its neighbors $p_{(i,j)}$ by calculating the gradient from the depth image

Figure 5: *The occlusions between real objects (given by the sensor's depth and color image – here both rooms with people inside) and virtual objects (in this case the bandit character) are automatically handled during rendering by the graphics hardware by first writing the acquired depth image into the frame buffer and then testing the rendered fragments' depth values against that value.*

through a series of cross products. Light influence calculated by a shader can be added to the pixel color in the real image. Differential rendering (see section 5.3) in contrast can cope with arbitrary lighting conditions, but requires knowledge of the real scene.

Correct reflectance behavior of the real surface depends on the reconstruction of material properties, which is not covered by the scope of this paper. We instead refer to the lengthy discussion in [Kühtreiber 2010], which covers on- and offline BRDF reconstruction in Augmented Reality applications. In addition, various illumination methods for MR classified according to the amount of radiance and geometry known from the real environment can be found in [Jacobs and Loscos 2004].

5.3 Shadows

With the *EnvironmentLight* in [Jung et al. 2010] a special X3D lighting node was proposed which allows fitting the lighting conditions to a given input environment map. The environment map can be streamed from a fisheye camera image capturing the real lighting conditions of the MR scene. The median-cut algorithm is proposed by [Debevec 2005; Madsen and Laursen 2007] as a method to extract geometric light sources from the environment map, including their direction, color and intensity. This information provides the necessary reconstruction of real light sources to render drop shadows of a virtual object onto the real scene.

In [Jung et al. 2007], it was shown how differential rendering – a rendering technique that adds the difference of two synthetic images to the real input image – can be used in conjunction with a shadowing algorithm for X3D applications to project virtual shadows into real scenes. This method requires the real scene to be reconstructed (which even nowadays is still done in an offline step) and tracked alongside the virtual object. By rendering this reconstructed scene two times, once with and once without the virtual object, the *difference* between both images extracts the shadows (as well as additional virtual lights and inter-reflections between real and virtual objects) from the rendering, which can be multiplied with the real image (see figure 4). A drawback of this method is that the real scene needs to remains static during runtime, since it is not feasible to track all real objects (especially deformable ones).

This requirement can be avoided: we use the depth map given by a depth-sensing camera to render shadows from the virtual object in a classical shadow mapping fashion, which nicely fits with the

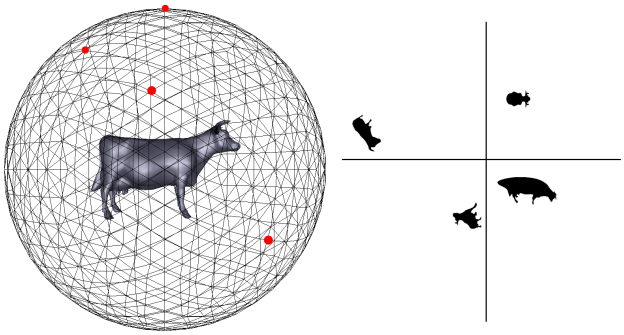


Figure 6: *Left: an OOF shell is sampled around the object, e.g. at the highlighted vertices on the sphere geometry. Right: for each sample a binary occlusion function is saved and compressed by projection into spherical harmonics.*

camera's depth map. A virtual or real light source (i.e. tracked or reconstructed) is used as viewpoint to render a depth map of the virtual object D_o. For each pixel of the real background image a corresponding depth value is transformed into light projection space and compared to the depth value in D_o. Depending on the comparison, the rendered pixel of the real background image is either shadowed or unshadowed. More complex shadowing techniques such as PCSS soft shadows [Fernando 2005] can be implemented with the same input.

5.4 Object Occlusion Fields

To simulate virtual shadowing on real geometry without reconstructing specific light sources or the real geometry we propose a modification of Object Occlusion Fields (OOF) [Zhou et al. 2005]. In [Franke and Jung 2008] it was argued that two node additions to X3D – *SphericalHarmonicsGenerator* and *SphericalHarmonicsMatrix* – are sufficient as a minimal framework to enable Precomputed Radiance Transfer (PRT) in X3D applications, but some of the more involving operations were left open for future discussion.

A basic requirement of PRT is that objects remain rigid so that their transfer functions on the surface can be precomputed. OOF's relax the rigidness requirements *between* objects so that they can behave dynamically under affine transformations and therefore might affect other objects with precomputed transfer (for instance by blocking off light). Shells around the object O with sparse sample positions q save the binary occlusion information introduced by the presence of the object as spherical harmonic vectors $\mathbf{t}^q$, as displayed in figure 6. Since each shell is essentially a sphere around the object, we upload these sampled coefficient vectors $\mathbf{t}^q$ as low resolution latitude-longitude textures to the GPU. Access on the GPU to the OOF texture automatically handles interpolation for unsampled spaces on the OOF shell.

At runtime, after each transformation, each PRT object (i.e. each surface with precomputed transfer vectors) is tested for collision with an OOF-shell of another object (this can be realized through a simple radius collision test for each OOF shell and does not require the help of a collision sensor). Colliding geometries will adjust their surface transfer $\mathbf{t}^p$ at surface point p by combining it with the occlusion coefficient vector $\widehat{\mathbf{t}}^q$ (i.e. $\mathbf{t}^q$ in the global frame) via triple product on the GPU (please refer to the appendix for clarification). The local transfer is now modified by the change introduced through the colliding OOF. The newly generated transfer vector can be combined in a double product with the light vector (generated from an *EnvironmentLight* through a *SphericalHarmonicsGenerator*) that is rotated with $\mathbf{R}_p^{-1}$ into the local tangent space. Equation 5 presents the final formula to merge diffuse transfer with a colliding OOF transfer vector and relight the scene with light coefficient vector $\mathbf{l}$ accordingly.

$$\begin{aligned} t_k &= \sum_j \sum_i t_i^p \widehat{t}_j^q C_{ijk} \\ L_{o,p}(\omega_\mathbf{r}) &= \left\langle \left(\mathbf{R_p^{-1}} \cdot \mathbf{l}\right), \mathbf{t} \right\rangle \end{aligned} \tag{5}$$

Applying this solution to a MR simulation with access to a light probe or a (fisheye) light camera and a depth camera does not work at first. The incident light captured by the light probe or (fisheye) light camera is transformed into a coefficient vector. A simple dot product of a surface transfer vector from a virtual object with the light vector yields the correct shading under real incident light, however the surface onto which the virtual object will cast its shadow is unknown and therefore transfer coefficients cannot be determined. Instead, we assume real surfaces to have diffuse reflection behavior and precalculate basic diffuse transfer vectors $t_i = \int_\Omega (\mathbf{n} \cdot \mathbf{l}_\omega) \Phi_i(\omega) d\omega$ for regular sample normal directions $\mathbf{n}$ and save them in a cache.

At runtime, we derive world space positions p of each pixel in the real image with the help of the depth map provided by the `DepthSensor` and calculate a normal vector $\mathbf{n}^p$ for it based on its neighbors. If the position p is inside a shell, we interpolate an occlusion vector $\widehat{\mathbf{t}}^q$ from the OOF and select a transfer vector of the real surface point $\mathbf{t}^p$ from the cache we have built up earlier by indexing into it with $\mathbf{n}^p$ and apply equation 8 to calculate the combined transfer vector. The double product of the light vector and the combined transfer vector yields a shadow attenuation value that can be multiplied with the fragment's color.

6 Conclusion and Future Work

We have presented three sensor extensions to X3D to enable support for depth sensing as generic `DepthSensor` devices or OpenNI compatible devices as `NISensor` with special support for several tracking algorithms reflected in a `UserSensor`. Several powerful and important effects for MR simulations rely on reconstructed environment information, which can be provided through a depth image. This depth image has to be matched with the color image of the main camera, which is either done automatically if a specialized device such as the Microsoft Kinect provides both cameras, or can be done manually for separate cameras mounted together. In combination with X3D, this greatly enhances development cycles for AR/MR applications.

A depth image is highly beneficial for MR applications: drop-shadows from virtual geometry can be inserted into the real scene easily with support from shadow mapping capabilities in X3D (e.g., as presented in [Jung et al. 2007; IR 2010]) or with image-based lighting algorithms combined with PRT and OOF rendering. Virtual light sources can be used with the existing depth information to illuminate real objects. Through depth imaging, prior knowledge and manual reconstruction of the real scene geometry is not necessary and provides an additional degree of freedom for the MR simulation. This is very important for AR/MR to decouple dependency on the tracked scene.

The proposed `UserSensor` provides an SFString field for the current gesture recognized by the subsystem. This event possibly shares common features with modern multitouch devices that are able to recognize touch gestures. Windows 7 [Microsoft 2010] and

Qt [Nokia 2010] for instance already have several gesture recognition algorithms implemented, such as zooming, panning, rotation or selecting. For single user interaction a common event system might be more consistent for the application developer.

References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMANN, N. 2008. *Real-Time Rendering*, 3 ed. AK Peters, Wellesley, MA.

ART, 2008. Artoolkit. http://www.hitl.washington.edu/artoolkit/.

BECKER, M., BLESER, G., PAGANI, A., STRICKER, D., AND WUEST, H. 2007. An architecture for prototyping and application development of visual tracking systems. In *Capture, Transmission and Display of 3D Video (Proc. of 3DTV-CON 07)*.

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.

BEHR, J., KNOEPFLE, C., AND PATRICK, D. 2005. A scalable sensor approach for immersive and desktop VR Application. In *HCI/VR International*.

BITMANAGEMENT, 2002. Drawgroup & drawop. 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 and Augmented Reality*, IEEE Computer Society, Los Alamitos, Calif., 56–65.

CALIBRATIONTOOLBOX, G. C., 2011. Gml camera calibration toolbox. http://graphics.cs.msu.ru/en/science/research/calibration/cpp.

CANUTESCU, A. A., AND DUNBRACK, R. L. 2003. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein Science 12*, 963–972.

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.

DEBEVEC, P. 2005. A median cut algorithm for light probe sampling. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Posters*, ACM, New York, USA, 66.

DIEFENBACH, P. J., AND BADLER, N. I. 1997. Multi-pass pipeline rendering: realism for dynamic environments. In *I3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 59–70.

ENGELHARD, N., 2010. KinectAutoCalibration. https://github.com/NikolasE/KinectAutoCalibration.

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

FORSYTH, D. A., AND PONCE, J. 2002. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference.

FRANKE, T., AND JUNG, Y. 2008. Precomputed radiance transfer for X3D based mixed reality applications. In *Proceedings Web3D 2008: 13th International Conference on 3D Web Technology*, ACM Press, New York, USA, S. Spencer, Ed., 7–10.

IR, 2010. Instant Reality. http://www.instantreality.org/.

JACOBS, K., AND LOSCOS, C. 2004. Classification of Illumination Methods for Mixed Reality. In *Eurographics 2004 - State of the Art Reports*, Eurographics Association, 95–118.

JUNG, Y., FRANKE, T., DÄHNE, P., AND BEHR, J. 2007. Enhancing X3D for advanced MR appliances. In *Proceedings Web3D '07*, ACM Press, NY, USA, 27–36.

JUNG, Y., WAGNER, S., BEHR, J., JUNG, C., AND FELLNER, D. W. 2010. Storyboarding and pre-visualization with x3d. In *Proc. Web3D 2010: 15th Intl. Conference on 3D Web Technology*, ACM Press, New York, USA, S. Spencer, Ed., 73–81.

KAHN, S., WUEST, H., STRICKER, D., AND FELLNER, D. W. 2010. 3D discrepancy check and visualization via augmented reality. In *9th IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 241–242.

KAMBURELIS, M. 2010. Shadow maps and projective texturing in x3d. In *Proceedings Web3D '10*, ACM, New York, NY, USA, 17–26.

KOLB, A., BARTH, E., KOCH, R., AND LARSEN, R. 2009. Time-of-flight sensors in computer graphics. In *EG 2009 - State of the Art Reports*, Eurographics Association, M. Pauly and G. Greiner, Eds., 119–134.

KÜHTREIBER, P. 2010. Brdf approximation and estimation for augmented reality. Tech. rep., Institute of Computer Graphics and Algorithms, Vienna University of Technology.

LEE, G., 2009. Supporting mixed reality visualization in x3d standards. http://web3d.org/membership/login/groups/Korea%20Web3D%20Forum/20090707-MR-Web3D-eng.pdf.

MADSEN, C. B., AND LAURSEN, R. E. 2007. A scalable gpu-based approach to shading and shadowing for photorealistic real-time augmented reality. In *GRAPP 2007: Proceedings*, INSTICC Press, 252–261.

MICROSOFT, 2010. Windows touch gestures overview. http://msdn.microsoft.com/en-us/library/dd940543%28v=vs.85%29.aspx.

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

NOKIA, 2010. Qt – cross-platform application and UI framework. http://qt.nokia.com/.

OCTAGA, 2009. Octaga – bringing enterprise data to life. http://www.octaga.com/.

OGGIER, T., LUSTENBERGER, F., AND BLANC, N. 2006. Miniature 3D ToF Camera for Real-Time Imaging. In *Perception and Interactive Technologies*, 212–216.

OPENCV, 2009. Opencv wiki. http://opencv.willowgarage.com/wiki/.

OPENNI, 2011. Openni framework. http://www.openni.org/.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87*, ACM Press, 283–291.

SCHILLER, I., BARTCZAK, B., KELLNER, F., AND KOCH, R. 2010. Increasing realism and supporting content planning for

dynamic scenes in a mixed reality system incorporating a time-of-flight camera. *Journal of Virtual Reality and Broadcasting 7(2010), CVMP 2008 Special Issue no. 4*.

SHI, J., AND TOMASI, C. 1994. Good features to track. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, 593–600.

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.

WEB3DCONSORTIUM, 2008. H-anim. http://www.web3d.org/x3d/specifications/ISO-IEC-19775-1.2-X3D-AbstractSpecification/Part01/components/hanim.html.

WEB3DCONSORTIUM, 2008. X3D. http://www.web3d.org/x3d/specifications/.

WUEST, H. 2008. *Efficient Line and Patch Feature Characterization and Management for Real-time Camera Tracking*. PhD thesis, TU Darmstadt.

ZHOU, K., HU, Y., LIN, S., GUO, B., AND SHUM, H.-Y. 2005. Precomputed shadow fields for dynamic scenes. *ACM Trans. Graph. 24*, 3, 1196–1201.

Appendix

In this section, we shortly introduce the term triple products. Given three functions $O(x)$, $T(x)$ and $E(x) = O(x)T(x)$ on a domain Ω, coefficients for any given basis Φ can be calculated with

$$O_i = \int_\Omega O(x)\Phi_i(x)dx \tag{6}$$

$$T_i = \int_\Omega T(x)\Phi_i(x)dx \tag{7}$$

and coefficients for $E(x)$ are derived in the manner of equation 8, where i, j and k are indices of the basis components.

$$\begin{aligned} E_i &= \int_\Omega \Phi_i(x)E(x)dx = \int_\Omega \Phi_i(x)\left(O(x)T(x)\right)dx \\ &= \int_\Omega \Phi_i(x)\left(\sum_j O_j\Phi_j(x)\right)\left(\sum_k T_k\Phi_k(x)\right)dx \\ &= \sum_j\sum_k O_jT_k\int_\Omega \Phi_i(x)\Phi_j(x)\Phi_k(x)dx \\ &= \sum_j\sum_k O_jT_kC_{ijk} \qquad (8) \\ &= \sum_j O_j\mathbf{T}_{ij} \qquad (9) \end{aligned}$$

The term C_{ijk} denotes a *triple product integral* or *tripling coefficient*. For spherical harmonics these values correspond to the well studied Clebsch-Gordan coefficients, but there is no general formula to analytically evaluate these coefficients for any basis Φ. Haar Wavelet tripling coefficients are presented in [Ng et al. 2004]. Depending on the method of calculation, surface transfer for any point p can be saved as either a transfer vector T or a transfer matrix $\mathbf{T}$, as shown in equation 9.