Integrating Stereoscopic Video in 3D Games

Jonas Schild[1], Sven Seele[2], and Maic Masuch[1]

[1] Entertainment Computing Group, University of Duisburg-Essen,
Forsthausweg 2, 47057 Duisburg, Germany
{jonas.schild, maic.masuch}@uni-due.de
[2] Bonn-Rhein-Sieg University of Applied Sciences,
Grantham-Allee 20, 53757 Sankt Augustin, Germany
sven.seele@smail.inf.h-brs.de

Abstract. Recent advances in commercial technology increase the use of stereoscopy in games. While current applications display existing games in real-time rendered stereoscopic 3D, future games will also feature S3D video as part of the virtual game world, in interactive S3D movies, or for new interaction methods. Compared to the rendering of 2D video within a 3D game scene, displaying S3D video includes some technical challenges related to rendering and adaption of the depth range. Rendering is exclusively possible on professional hardware not appropriate for gaming. Our approach, Multi-pass Stereoscopic Video Rendering (MSVR), allows to present stereoscopic video streams within game engines on consumer graphics boards. We further discuss aspects of performance and occlusion of virtual objects. This allows developers and other researchers to easily apply S3D video with current game engines to explore new innovations in S3D gaming.

Keywords: Stereoscopic rendering, S3D video, game engine, 3D gaming.

1 Introduction

The use of stereoscopic 3D (S3D) vision in digital entertainment technologies has increased significantly over the past few years [8]. Based on the overwhelming success of the movie Avatar and the following boost in market revenue for 3D cinema in general [12], companies try to push S3D content into other market segments like blu-rays, TV sets, TV channels, and live broadcasts [10]. Similarly, the digital games industry tries to benefit from the current interest in S3D as well. Nintendo has released the first autostereoscopic hand-held console with its 3DS [14]. Sony is pushing S3D gaming on its PlayStation 3 [22], backed up by its series of 3D television sets. Influential developers like Crytek realize S3D render techniques [20] and Nvidia has introduced 3D Vision for S3D gaming on the PC [3,11].

The addition of stereoscopic depth is believed to create a whole new experience in all of the aforementioned fields and it will eventually create a mass consumer product that will be taken for granted, just as color replaced black-and-white television [5,9,10]. One motivation for displaying 3D video in games is the integration of 3D video as contemporary media technology: In the future, people expect integrated video content to be displayed in S3D on a regular basis. This also affects video being displayed in interactive game environments, e.g. advertisement billboards in sports games, or on TV screens in a game world as in Grand Theft Auto 4[1]. This would require rendering stereoscopic video on a virtual screen within a 3D game world.

A second motivation arises from interactive video in games. Interactive movies, such as Rebel Assault[2] or The X-Files Game[3], were prominent in the 1990s and allowed combined interaction with pre-filmed and virtual objects. As pre-filmed material could hardly provide enough freedom for innovative concepts, recent games rather incorporate live video: The Sony EyeToy[4] and Microsoft Kinect[5] provide live video-based interaction using computer vision methods and depth-aware cameras. The game YooStar2[6] combines live-filmed video with pre-filmed footage. BallBouncer explores multi-user gaming with live video in a cinema setting [19]. However, these examples provide only monoscopic video, as of yet.

Stereoscopic vision could allow for new forms of interaction with such content, as has been demonstrated in the domain of augmented and mixed reality [17]. A crucial problem remains the correct alignment of virtual and real objects and the coherent perceptual problems [13]. A first commercial application of stereoscopic augmented reality is the AR Games set that is bundled with every Nintendo 3DS device [15]. The alignment of virtual and filmed objects is based on marker tracking and works well within a specific distance. The stereo camera configuration is fixed.

Apart from such ideal prerequisites, our work assesses the necessary steps to integrate arbitrarily configured pre- or real time-recorded S3D video into the virtual environment of digital S3D games, i.e. on a virtual screen. The next section gives an overview of our technical platform and discusses the use of stereo cameras. Subsequently, we use this platform to explore the integration of the video data into a commercial 3D game engine. As current consumer hardware and Direct3D do not support rendering of S3D video, we propose a workaround: Multi-pass Stereoscopic Video Rendering (MSVR). Our approach further explores the problem of parallax adaptation and gives a practical solution to deal with such non-trivial setups. Lastly, we discuss the solution and pinpoint the need for automatic methods of parallax recognition and occlusion handling.

2 Technical Platform

For our approach, we used a typical gaming PC running Microsoft Windows 7 Professional, with Nvidia GeForce GTX 470 graphics card, Intel Core i5 Quad-core processor, and eight gigabyte of memory. Our implementation of MSVR (see Section 3.1) was incorporated into Trinigy's Vision Engine, a professional game engine that supports video textures and custom render methods [21]. The engine provides a C++ API for development and supports Direct3D-rendering, which is required to use the Nvidia 3D Vision driver. It is widely used for commercial game titles, such as The Settlers 7, Stronghold 3, or Arcania: A Gothic Tale [21].

[1] Rockstar Games, Inc., 2008.
[2] LucasArts Ltd., 1993.
[3] Fox Interactive, 1998.
[4] Sony Computer Entertainment, Inc., 2003.
[5] Microsoft Corp., 2010.
[6] Yoostar Entertainment Group, Inc., 2011.

2.1 Camera Setup and Support

The MSVR framework supports not only pre-recorded files, but also live captured S3D video. For this purpose we integrated access to several camera systems: DirectShow-based cameras, two PlayStation Eye cameras (PSEye) based on the CL-Eye Platform Driver [1], and a pair of professional high definition cameras. For using any of these approaches, the following issues have to be considered:

DirectShow can be easily modified to support two USB cameras, but as of yet it only allows the use of two different cameras and does not support multiple cameras of the same model. However, when capturing S3D videos, it is recommended to use cameras as similar as possible in every aspect as any differences have a strong impact on binocular fusion [12].

The use of Playstation Eye cameras is also cumbersome. First, the camera's wide shape does not allow for small interaxials. Second, even two PSEyes bought together from the same shop included models with fairly different serial numbers. Thus, the received images differed strongly in terms of color, brightness, and compression artifacts, which is not suitable for good S3D vision.

In our setup we used a stereo camera rig based on the microHDTV camera element provided by the Fraunhofer Institute IIS. The cameras are very compact and support HD and Full HD resolutions at 24 to 60 Hz (synchronized). Each camera can be adjusted within six degrees of freedom. The cameras can be configured via Ethernet (image resolution, frame rate, gain, white balance, etc.) and support image transmission via HD-SDI MCX. The stereo camera rig is connected to a professional video capture board (DVS Centaurus II[7]) via HD-SDI. The data can be accessed using a SDK.

But even when incorporating only professional cameras, a common interface still allows for simple and fast exchange of input components which is valuable during development when the rig is not available. This can be crucial as stereo camera rigs are very sensitive to physical force and often require recalibration.

2.2 Nvidia 3D Vision

The 3D Vision package includes a pair of shutter glasses, an emitter used for synchronization, and the 3D Vision driver. It further requires a display running at a refresh rate of at least 120 Hz and a GeForce graphics card of the eighth generation or later. Alternatives exist with DDD's Tridef [2] and iZ3D [6]. One major advantage of 3D Vision is the automatic stereoization of Direct3D applications. The driver intercepts all calls to the Direct3D API and uses them to create S3D content which is then displayed on the monitor in synchronization with the shutter sequence of the glasses. Besides following some basic guidelines to enhance the stereoscopic quality, no additional effort is required of the application developer and the user can easily control the stereo effect [11].

Obviously, it is straightforward to display videos as textures placed on arbitrary geometry. Here the automatic stereoization becomes an issue: the driver alters the part of the standard render pipeline after clip space into two separate paths, one for each eye. This is simply achieved through a translation in parallel to the screen plane,

[7] DVS Digital Video Systems GmbH.

which is automatically added to appropriate vertex shaders [11]. Everything but the vertex positions will be identical for each view, including the source texture. Thus, depending on the depth in the scene, each image will only contain an offset version of the same texture. Therefore, the resulting image will be stereoscopic, but the video texture will only contain monoscopic information.

The driver should be able to provide two different textures, one for each view. This technique is only available with the professional version of the 3D Vision driver on high-end Quadro cards which support quad-buffered stereo where the entire render pipeline is traversed separately for each eye [16]. This allows for full control over the right and left view in the application. Since quad-buffered stereo is only available for professional hardware, it cannot be used for game development where only consumer gamer hardware can be assumed. This lack of control requires a workaround for displaying S3D video content in a game scenario. Fortunately, the 3D Vision driver offers ways to display S3D videos and photos [3] on which we based our workaround: MSVR.

3 Integration of Stereoscopic Video into a Game Scene

In this section we describe our approach of integrating S3D video into a professional game engine. We first describe our rendering method MSVR, which was briefly introduced in [18], and give further insights on the implementation. The following subsection proposes a solution for adjusting different parallaxes of S3D video content, the underlying occlusion mesh, and game engine content. For aligning depth properties of the video with those of the game scene, we give a first solution by manually setting the video parallaxes. We also show how video content can be split up and distributed within the game scene. The last subsection pinpoints problems of occlusion with virtual objects and video content.

3.1 Multi-pass Stereoscopic Video Rendering (MSVR)

The engine's default render module is extended to process the S3D video content. The content is made available to the application as a video texture, e.g. using standard DirectShow filters. Then two quads of identical size and location are created, one for each eye. Since the quads are used to display the video, they will be referred to as canvas objects. The video files contain both images in a certain alignment depending on the video format. Therefore, care needs to be taken when assigning texture coordinates to both canvas objects. As a result, the left eye image is mapped to the left canvas object and the right eye image to the right canvas object. The virtual camera and the two canvas meshes are positioned within the game scene.

During the first render pass the entire scene is rendered into the backbuffer without any geometry but the left canvas object. Before clearing the backbuffer, its content is copied onto a Direct3D surface. In the second pass the scene is rendered with only the right canvas object present and the backbuffer content is again copied onto another Direct3D surface.

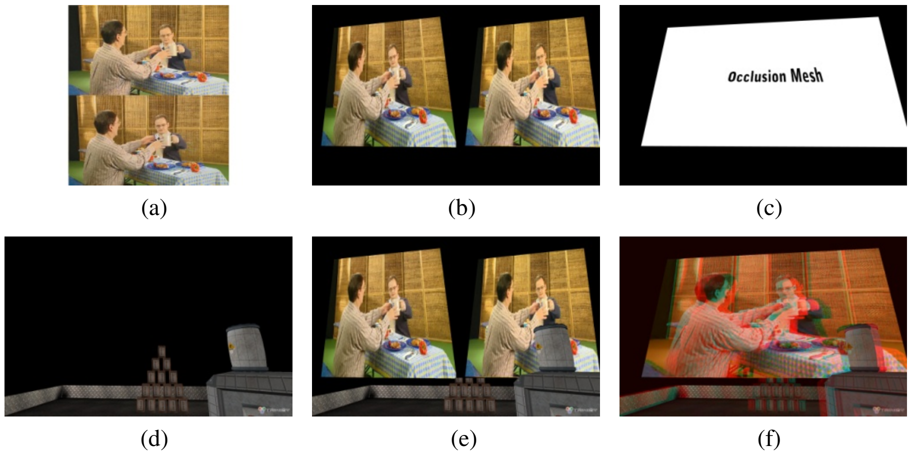

Fig. 1. The image sequence shows the general application flow and the composition of the different elements of the MSVR method. (a) A frame from the S3D video. (b) The textured canvas surfaces as stretched into the backbuffer. (c) Illustration of the rendered occlusion mesh for clarification. (d) A scene as rendered by the engine's default renderer. (e) The composition of the game scene and the stereo canvas in non-stereo mode. (f) The same composition in anaglyphic stereo mode.

During application initialization, a third Direct3D surface, called stereo surface is created which has twice the width of the backbuffer and its height plus an additional row of pixels. The previous backbuffer contents, saved onto the individual surfaces, are copied side-by-side onto the stereo surface. The additional row is used to insert a stereo tag that directs the 3D Vision driver to present the backbuffer's content in S3D [3]. To fit both views into the backbuffer they are copied using Direct3D's StretchRect method. If the display buffer is now rendered with the 3D Vision driver turned off, the screen will simply show two distorted versions of the images, as illustrated by Fig. 1b.

Since the S3D content is only copied into the backbuffer and not actually rendered, no depth information will be present in the depth buffer. Thus, if the third pass would be to render the scene, everything would be rendered in front of the canvas objects. This could mean a complete occlusion of the video content by scene geometry. Thus, the third step is to render an additional quad mesh of the same size and position as the canvas objects (see Figure Fig. 1c). The depth information produced by this render pass will now remain in the depth buffer for the following passes. This ensures that during the fourth pass, in which the rest of the scene is rendered via the engine's default render call, occlusions are correctly determined for the video display surface.

Displaying the scene without the 3D Vision driver will result in a combination of the in-game objects and the distorted S3D image pair (see Fig. 1e). If the driver is activated, the third and fourth pass will be stereoized automatically. The result will be a S3D game scene with an integrated canvas object that shows a S3D video, exemplified by Fig. 1f.

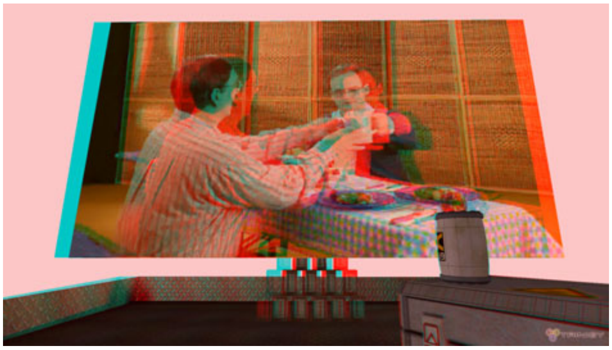

Fig. 2. The scene illustrates the three different depth settings and respective parallax differences caused by unadjusted MSVR: Note that the occlusion mesh's parallax causes a black border and that due to the differences between scene and video parallaxes it becomes difficult to fuse the images.

As with all multi-pass approaches, MSVR obviously creates performance overhead. Compared to adding mono video to a stereo scene, adding the stereo video about halves the frame rate to 30 fps in fullscreen mode. Rendering the additional occlusion canvas, however, does not alter the performance considerably (-1 fps).

Another issue results from the different screen parallaxes produced by the individual steps of the MSVR approach. First, stereoscopic image pairs are rendered to the same backbuffer locations. As a result, the displayed image shows the inherent video parallax as determined during the capture process. Second, the rendered occlusion mesh is subject to the 3D Vision driver's stereoization process. Thus, its parallax depends on its placement in the scene. Finally, the game scene exhibits parallaxes according to its depth layout as extracted by the 3D Vision driver. In our original approach [18], these three types of parallaxes were uncorrelated (see Fig. 2).

3.2 Adjustment of the Occlusion Mesh Parallax

One important issue in MSVR is a correlation between the canvas locations and the stereoization of the occlusion mesh: As the mesh is part of the scene, it is duplicated by the 3D Vision driver based on the current interaxial distance and its depth in the scene. For this reason, there will be a parallax between each eye's images of the occlusion mesh. However, the individual canvas objects are rendered to the exact same position in both views causing no parallax, except the one inherent to the video content.

To demonstrate this problem, think of the occlusion mesh as a rectangular cutout in the game engine's scene, allowing an observer to look through it and see the previously prepared backbuffer, containing the S3D image. For one eye, the cutout will coincide exactly with the stereoscopic content. For the other eye, the cutout will be moved horizontally by the 3D Vision driver, but the video texture will remain fixed. Thus, parts of the cutout reveal the blank backbuffer, ultimately resulting in a black border on one side of the canvas. The width of the border will scale with the amount of screen parallax depending on the virtual camera interaxial and the canvas' depth in the scene.

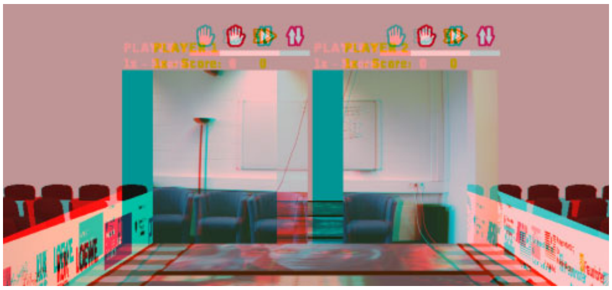

Fig. 3. Occlusion mesh border on two split video canvasses: Since the parallax of the occlusion mesh is adjusted by the 3D Vision driver, but that of the S3D video canvas is not, a black border is perceived by one eye

The effect is demonstrated in Fig. 3. Using anaglyphic glasses and looking at the image with the right eye closed, reveals the mentioned border on the left side of each canvas object while the right eye's view will be correct. Because of this border, it becomes difficult to properly fuse the images.

To accommodate for this offset either the position of the occlusion mesh or that of the canvas object in one view has to be adjusted. Since the occlusion mesh is duplicated by the Nvidia driver, the only way to adjust its parallax is to position it at a different depth and adjusting its size to still appear perspectively correct. While the idea behind this approach is simple, the actual implementation would pose several problems. For instance, adjusting the size of the mesh object would require locking its vertices; a very costly operation. Doing so every frame would cause a drastic decrease in performance.

A more elegant way could be to create a custom vertex shader for the mesh, which scales the output position of each vertex. The 3D Vision driver uses the homogeneous w-coordinate of the position vector to determine the vertex's parallax offset and with it its perceived depth. To achieve the desired depth the vertex position vector simply needs to be scaled accordingly. Since the driver operates in clip space, which is right before perspective divide is applied, this scaling operation will affect the parallax computations performed by the 3D Vision driver, but not the actually rasterized locations determined from the vertex position.

However, the purpose of the occlusion mesh is to correctly position the video element within the scene, so adjusting its depth (be it manually or with a vertex shader) would be counterproductive. Therefore, the canvas object needs to be rendered at the correct depth.

Of course, the issue could also be resolved by cropping the occlusion mesh, so that the border would simply not show. This would reduce the visible part of the images proportionally to the amount of cropping. At the same time, it would still leave the inherent disparities unchanged so that there would be no correlation between the parallax of the S3D video content and the stereoscopically rendered scene.

Instead, the offset between both canvas objects should match the parallax of the occlusion mesh to account for the horizontal shift and its depth in the scene. To accomplish this it is crucial to understand how the Nvidia driver calculates the parallax of a vertex at scene depth d. The used function is defined as follows:

$$parallax(d) = separation \cdot \left(1 - \frac{convergence}{d}\right) \quad (1)$$

$Convergence$ and d are both distances to the cameras given in world coordinates. The concept of convergence, as used by the 3D Vision driver, does not concur with the common understanding of converging eyes or camera axes. While it would be logical in the common model to represent convergence in form of an angle, it does not make sense for the 3D Vision driver since the cameras will not be toed-in, but simply translated to avoid producing vertical parallaxes, using an asymmetric viewing frustum. Thus, the convergence value is expressed by a distance measure, defining at which scene depth the convergence plane is situated. All vertices on this plane will experience no transitional offset, which means they will have zero disparity making them appear at the same depth as the display screen.

The separation is given in percentage of the image width and thus, so is the parallax value. Therefore, the formula needed to be adjusted, since it was necessary to know the parallax in units of the world coordinate system to correctly place the canvas. First, it was necessary to know the width of the image (buffer) n_w in world coordinates. To calculate this value, the distance to the near clipping plane n_d and the horizontal field of view of the virtual camera (fov_x) were necessary. Both could be obtained from the Vision Engine's render context. The width was then simply given by the following trigonometry function:

$$n_w = 2 \cdot n_d \cdot \tan\left(\frac{fov_x}{2}\right) \quad (2)$$

Knowing the width of the near clipping, the parallax can be converted into world coordinates by scaling it with the result of Equation (2).

$$parallax_s = parallax(d) \cdot n_w \quad (3)$$

This $parallax_s$ gives the offset in world coordinate units at screen depth. The final step left is to scale the parallax, so it can be applied to an object at the specified distance d. Since all values are given (the distance to the near clipping plane, the parallax at screen depth, and the distance to the object), the theorem of intersecting lines is used to determine the sought after parallax value as indicated by the following equation:

$$parallax_d = \frac{parallax_s \cdot d}{n_s} \quad (4)$$

Knowing the distance between the stereo canvas and the cameras, Equation (4) can be used to determine by how much the occlusion mesh needs to be translated horizontally. During each update of the stereo canvas, this value is used to translate the left eye's canvas accordingly to avoid the border problem.

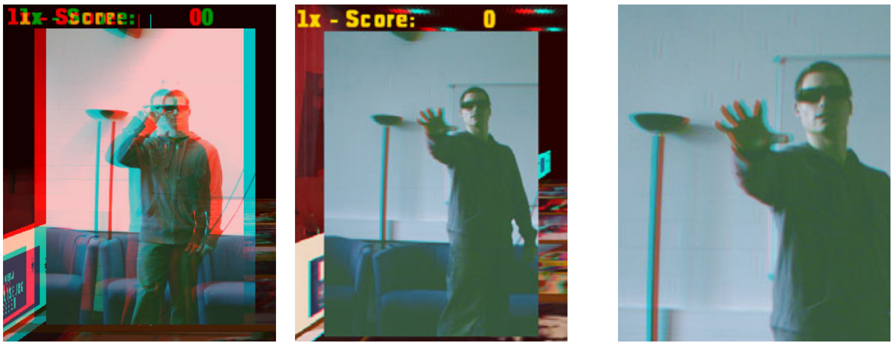

Fig. 4. Screen Parallax: Examples of adjusted screen parallax. **(a)** Due to constant adjustment and the zero parallax calibration, the disparity of the player's image is always almost exactly equal to that of the occlusion mesh. **(b)** Screenshot taken during the zero parallax calibration explaining the concept of the artificial convergence plane. **(c)** A magnified part of (b).

3.3 Manually Fitting Video Content Depth to the Game Scene Depth

While the parallax of the stereo image borders can be correctly adjusted with the aforementioned approach, the disparities within the stereo content would still be completely uncorrelated with those of the rendered game scene. Unless the video supports complete depth information (i.e. a depth map), a complete adaptation of the video depth range into the virtual depth budget is not possible. Our approach therefore concentrates on adjusting the depth of a particular depth plane from the video in respect to a given depth plane within the game scene. As our prototype incorporates live 3D video, we explore this problem using video from the stereo camera setup mentioned before.

Our approach is based on a manual setup using a zero parallax setting. First, the stereo canvas is positioned at the convergence plane, defined by the 3D Vision driver's convergence value. Consequently, the horizontal parallaxes of the occlusion mesh and of the adjusted S3D video frames are zero (see Fig. 4 b+c). Now the video content is shifted manually until objects at a specific depth within the video also show zero disparity. For our example, this artificial convergence plane was set at the position of a person standing in front of the camera. The additional horizontal offset determined in this way is added to the result calculated by the parallax equation:

$$parallax_o = parallax_d + offset \tag{5}$$

In this way the image and scene disparities would approximately coincide at all times. An example of this is shown in Fig. 4a. Since the calibration was performed manually, small errors were obviously inevitable, but if done properly, the errors are not visible to the naked eye. A challenging but very interesting task for future developments would be to perform this step in software automatically.

3.4 Splitting the Canvas

Our system allows splitting the camera image to display several parts of the image on different canvas objects (cf. Fig. 3). One application could be to identify the faces of multiple players within a video and distribute the according parts throughout the game scene.

To accomplish a split into two parties for example, two meshes instead of one are created for each eye, each with half the width of the original meshes. Additionally, the texture coordinates associated with each vertex are altered so that one split contains the left half of the video texture and the other split the remaining half.

Consequently, each split can be set completely independent within the scene. Of course, it is necessary to still treat corresponding splits, for example the left split of the left and right eye's canvas, in correspondence with the aforementioned considerations about parallax and offset to achieve the correct depth impression.

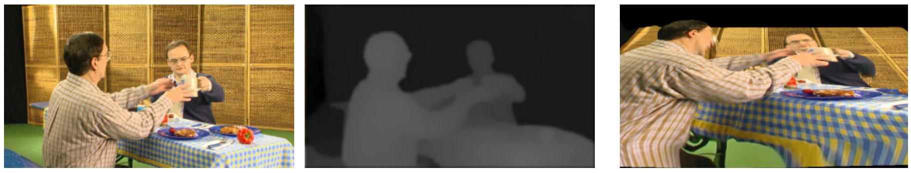
(a) (b) (c)

Fig. 5. Using MSVR for stereoscopic rendering of the video-plus-depth format (a+b). The generated occlusion mesh can be textured with the video (c) or stays hidden for z-buffer test with the game scene.

3.5 Problems of Occlusion with Objects and Video

Using S3D video in games may incorporate interaction of virtual objects with certain features of the video content, positioned at arbitrary depth layers. But, as the occlusion mesh of the video is fixed, virtual objects that shall be positioned in depth next to a video feature at large positive parallax, might be occluded. Thus, virtual objects are not enabled to extend into the depth of the video.

One simple approach would be to render such virtual objects with disabled depth test and in the correct order. In a complex interactive setup, this implicates a thorough design of positioning and animating scene objects which clearly restricts game design. Furthermore, virtual objects might occlude video features of smaller parallax values, possibly confusing depth perception.

Another approach is the use of non-planar occlusion meshes (see Fig. 5). Currently, a depth map of the video is required (as in video-plus-depth or depth-enhanced-stereo formats). It is used to create a displacement occlusion mesh. The vertices are shifted according to the depth information. The visual occlusion quality depends on the resolution of the depth map. In the future, we consider automatic extraction of depth information and updating displacement meshes through vertex shaders. For generating the required depth data in a live video situation, depth cameras and other systems as Microsoft Kinect seem promising.

4 Conclusions and Future Work

We presented our approach of rendering S3D video content in a professional game engine and discussed problems and solutions for integrating it into a S3D scene. While the multi-pass rendering approach only concerns a single technical platform, it allows other programmers, scientists, and hobbyists to develop custom stereoscopic visuals on consumer hardware. Besides stereoscopic video, any other independently arranged scenes can be rendered onto the canvasses, to circumvent the quad-buffered stereo barrier. Beyond this practical scope, the presented workings of occlusion handling and parallax integration are general problems that significantly influence the quality of S3D video display within a game scene. Our concepts give a first solution for manual adjustment and pinpoint the need for automatic methods of parallax adaptation and to generate depth data of a video. For the latter problem, future 3D video formats will hopefully offer depth meta data, as this is a major requirement for generating multi-view stereo data for autostereoscopic displays. This development highly depends on depth-measuring cameras [4] or automatic depth map extraction algorithms [7] in the domains of image processing and computer vision.

Possible future applications of S3D video are the integration as a contemporary medium within a virtual game world, and interactive S3D movies. The use of stereoscopic video may provide for better interaction opportunities: slightly changing the viewport through position shift or zooming in or out could provide for more dynamic visual exploration of a filmed scene. Depth estimation or the use of pre-rendered depth maps can allow for dynamically mixed pre-rendered/-filmed graphics with real-time animated objects. The use of live-feed S3D cameras can provide for stereoscopic augmented reality entertainment. Using MSVR, this design space can be explored using typically available graphics hardware and powerful game engine software.

Acknowledgments. This work has been co-funded by the BMWi through the Prime3D project. Video images used are partly courtesy of Fraunhofer HHI. Vision Engine educational license was kindly provided by Trinigy. We would like to thank DVS for providing the capture card, Fraunhofer IIS for the cameras, and all our partners for their valuable support.

References

1. Code Laboratories. Cl-eye platform driver, `http://codelaboratories.com/products/eye/driver/`
2. Tridef, D.: Bringing 3D to mainstream, http://www.tridef.com/home.html
3. Gateau, S.: The in and out: Making games play right with stereoscopic3d technologies, `http://developer.download.nvidia.com/presentations/2009/GDC/GDC09-3DVision-The_In_and_Out.pdf`
4. Hahne, U., Alexa, M.: Depth Imaging by Combining Time-of-Flight and On-Demand Stereo. In: Kolb, A., Koch, R. (eds.) Dyn3D 2009. LNCS, vol. 5742, pp. 70–83. Springer, Heidelberg (2009)
5. IJsselsteijn, W., de Ridder, H., Freeman, J., Avons, S.E., Bouwhuis, D.: Effects of stereoscopic presentation, image motion, and screen size on subjective and objective corroborative measures of presence. Presence 10(3), 298–311 (2001)

6. iZ3D. The Future Of Gaming Is Here, http://www.iz3d.com/
7. Kim, H., Yang, S., Sohn, K.: 3D Reconstruction of Stereo Images for Interaction between Real and Virtual Worlds. In: Proceedings of the 2nd IEEE/ACM International Symposium on Mixed and Augmented Reality (ISMAR 2003), pp. 169–178. IEEE Computer Society Press, Washington, DC, USA (2003)
8. Kroeker, K.L.: Looking beyond stereoscopic 3D's revival. Communications of the ACM 53(8), 14–16 (2010)
9. Lambooij, M.T.M., IJsselsteijn, W.A., Heynderickx, I.: Visual discomfort in stereoscopic displays: A review. In: Woods, A.J., Dodgson, N.A., Merritt, J.O., Bolas, M.T., McDowall, I.E. (eds.) Proceedings of the Stereoscopic Displays and Virtual Reality Systems XIV, vol. 6490. SPIE, San Jose (2007)
10. Lang, M., Hornung, A., Wang, O., Poulakos, S., Smolic, A., Gross, M.: Nonlinear disparity mapping for stereoscopic 3d. In: ACM SIGGRAPH 2010 Papers, p. 75:1-75:10. ACM Press, New York (2010)
11. McDonald, J.: Nvidia 3d vision automatic, http://developer.download.nvidia.com/whitepapers/2010/3DV_BestPracticesGuide.pdf
12. Mendiburu, B.: 3D Movie Making – Stereoscopic Digital Cinema from Script to Screen. Focal Press, Burlington (2009)
13. Milgram, P., Drascic, D.: Perceptual effects in aligning virtual and real objects in augmented reality displays. In: Human Factors and Ergonomics Society Annual Meeting Proceedings, pp. 1239–1243. Human Factors and Ergonomics Society (1997)
14. Nintendo: Nintendo 3ds, http://www.nintendo.com/3ds/
15. Nintendo: 3DS AR Games, http://www.nintendo.com/3ds/built-in-software#/4
16. Nvidia Quadro quad buffered professional stereo technology, http://www.nvidia.com/object/quadro_stereo_technology.html
17. Ohta, Y., Tamura, H. (eds.): Mixed Reality: Merging Real and Virtual Worlds. Ohmsha & Springer, Tokyo (1999)
18. Schild, J., Seele, S., Fischer, J., Masuch, M.: Multi-pass rendering of stereoscopic video on consumer graphics cards. In: Symposium on Interactive 3D Graphics and Games I3D 2011, p. 209. ACM, New York (2011)
19. Sieber, J., McCallum, S., Wyvill, G.: BallBouncer: Interactive Games for Theater Audiences. In: 13th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, Dunedin, New Zealand, pp. 29–31 (2008)
20. Taube, A.: Stereoskopie für Spiele-Entwickler. Making Games Magazine 8(1), 12–19 (2011)
21. Trinigy: The vision game engine, http://www.trinigy.net/en/products/vision-engine
22. Thorsen, T.: "Nearly" 30 3D PS3 games due in (2011), http://www.gamespot.com/news/6305285.html