computer graphics & vision

Inffeldgasse 16/2, A-8010 Graz

www.icg.tu-graz.ac.at

TUG

ARift

Augmented Reality for the Oculus Rift DK1

Nikolaus Heran, Markus Höll, Vincent Lepetit

Inst. for Computer Graphics and Vision
Graz University of Technology, Austria

Technical Report
ICG–CVARLab-TR–001
Graz, February 22, 2016

contact: Nikolaus Heran nikolaus.heran@tugraz.at

Abstract

In this work we investigate the creation of an augmented reality environment using the Oculus Rift DK1 combined with two webcams, similar to what has been done by Steptoe et al. in [1]. First we recapitulate the definition of augmented reality and the general obstacles to overcome. Upon which we discuss the state of the art and move on to the specifics of this implementation of augmented reality. Finally we describe ongoing and propose future work building upon this realization.

Keywords: *Report, Technical report, template, ICG, AR, ARift, Augmented Reality, Oculus*

1 Introduction

Virtual and augmented reality have been topics in research and popular culture for multiple decades [2, 3] but recent developments in hard- and software have sparked new interest in these subjects in a wide audience. Especially the upcoming release of head mounted virtual reality devices as the Oculus Rift and HTC Vive move this topic closer to the homes and lives of consumers than ever before. According to these companies the hardware for the end users will be just as affordable as a new desktop gaming computer. Both devices are aimed at providing access to a virtual reality but not augmented reality. Given the close relation of virtual and augmented reality a conversion to augmented reality can be achieved with very few downsides.

In this report we discuss the necessary steps for creating such a rig and software for it. To create the illusion of virtual reality three requirements must be met: the creation of a virtual world, measurements of the device's position and orientation and finally a convincing animation of the virtual world with the measurements in mind [3]. Advancing from virtual reality to augmented reality requires capturing the real scene and melding it with the already created virtual reality. To capture the real scene we mounted two cameras (IDS uEye UI-122-1LE-C-HQ) with fisheye lenses to the front of the device. The next step in the conversion process is creating an augmented reality application. This software superimposes virtual and real scene. Especially in the border regions the used fisheye lenses show significant distortion. To alleviate this problem and heighten immersion the cameras need to be calibrated. We used the OCamCalib matlab toolbox by Scaramuzza et al. [4] for this calibration. To improve performance the images are being undistorted on the graphics card. Our implementation was created for the Oculus Rift DK1 because it was already available at the start of the project. However, the software can easily be adapted to work with other hardware as well.

The specifics of the creation of the virtual world are described in chapter 2. Details about the calibration process are discussed in section 4. A summary and observations of the achieved results can be read in chapter 6. Finally ongoing and future work with the described approach concludes this report in chapter 7.

2 Creating a virtual world

There is a wide variety of methods to create a virtual world known from computer graphics, but almost all of them use homogeneous coordinates. Like many others we chose to add a fourth component to the position vec-

tor as well. Using homogeneous coordinates allows us to write rotations, translations, scalings and projections as matrix multiplications [5, 6].

The virtual world is populated by virtual objects. Each object consists of at least three vertices, where each vertex also has a normal vector associated with it. Therefore a directed triangle is the most primitive model that can be created in our application. A texture can be applied to the object by defining texture coordinates for each vertex and interpolating on the texture in between. Each object is placed using a position vector $\mathbf{x} = [x, y, z, w]$ and oriented using a rotation vector $\phi = [\phi_x, \phi_y, \phi_z]$. Furthermore each object can be scaled along each of the three dimensions.

To display the virtual world on the screen every vertex $\mathbf{x}$ is transformed into screen coordinates $\mathbf{x}_{\text{screen}}$ in the following way:

$$\mathbf{x}_{\text{screen}} = \mathbf{PVMx} \tag{1}$$

In equation 1 the projective transformation is denoted by $\mathbf{P}$. $\mathbf{V}$ contains the viewpoint transformation (i.e. where the virtual camera is and where it is looking at). The models placement, orientation and scale in the virtual world is introduced with $\mathbf{M}$.

3 From a virtual world to virtual reality

Achieving the illusion of looking into a virtual world, either by using a hand held device or through a head mounted display (HMD), requires that the virtual objects behave as if they were real. The most basic aspect of this is that the device's movement and rotation must be transported into the virtual world and be applied to the virtual camera(s). Measurements about the device's position $\mathbf{X} = [X, Y, Z]$ and rotation $\phi_R = [\phi_{X,R}, \phi_{Y,R}, \phi_{Z,R}]$ are therefore needed. The measurement coordinate system is depicted in Figure 1.

Given these measurements the virtual cameras can be translated and rotated in correspondence. Only the relative motion $\Delta_t\mathbf{X}$ and rotation $\Delta_t\phi_R$ between two consecutive frames are required. For a 2D display this is already enough to achieve the illusion. For HMDs further steps are required to achieve immersion. It is detailed by Rolland and Hua in [2] and Kooi and Toet in [7] that various techniques exist to gain a 3D perception of the virtual scene. All of them require to present each eye with a slightly different image. Thus the virtual scene has to be rendered twice.

The movement of the virtual cameras has to be in agreement with the movement of the wearers head. Each eye has a fixed offset from the heads rotation center $\mathbf{x}_{head,t}$. This offset is denoted by $\vec{\mathbf{x}}_{l,eye}$ for the left eye and

by $\vec{\mathbf{x}}_{r,eye}$ for the right eye. To determine the 3D positions of the virtual eyes (cameras) $\mathbf{x}_{l,eye,t}$, $\mathbf{x}_{r,eye,t}$ for frame t the relative head rotation $\Delta_t\phi_R$ and translation $\Delta_t\mathbf{X}$ is applied as can be seen in equation 2.

$$\mathbf{x}_{*,eye,t} = \mathbf{T}_t\Phi_t\mathbf{T}_{*,eye}\mathbf{x}_{head,t-1} \tag{2}$$

Where $\mathbf{x}_{head,t}$ denotes the heads position at frame t, Φ_t is a homogeneous rotation matrix constructed from $\Delta_t\phi_R$ and $\mathbf{T}_t$, $\mathbf{T}_{*,eye}$ are homogeneous translation matrices constructed from $\Delta_t\mathbf{X}$, $\vec{\mathbf{x}}_{*,eye}$ respectively.

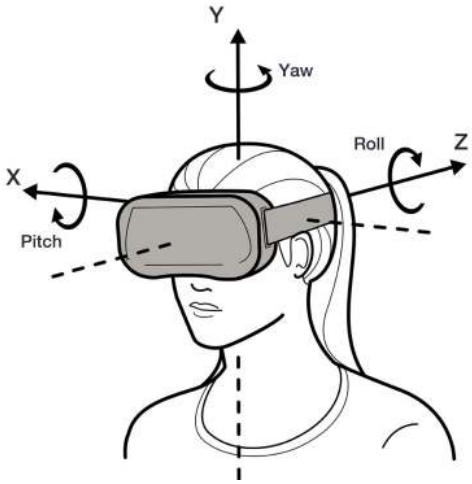


Figure 1: Measurement coordinate system for the orientation and movement of the device. Pitch corresponds with $\phi_{X,R}$, yaw with $\phi_{Y,R}$ and roll with $\phi_{Z,R}$. Image taken from [8].

Now the position of each virtual camera is known, but the viewing directions of each of the cameras still has to be determined. Since the object the eyes focus on is in a slightly different place in each eyes image the standard projection in equation 1 has to be replaced by an off axis projection as described in [9]. Figure 2 shows the principle of the off axis stereo projection.

4 Camera calibration

Before the camera images can be incorporated into the virtual scene an undistortion step is needed. In order to acquire the undistortion function the used webcams need to be calibrated. We chose to use the approach described by Scaramuzza et al. in [10]. This method calibrates the full system including external and internal parameters and has an open source Matlab toolbox

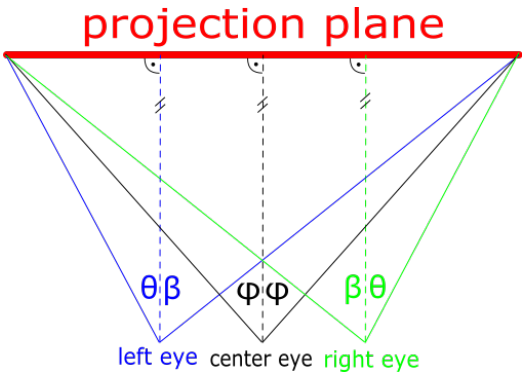


Figure 2: Top view of the principle of off axis stereo projection. The center eye (black) depicts a standard projection, the left eye (blue) and right eye (green) depict the used stereo projection mapping.

publicly available. Another important motivation for choosing this approach is the simplicity of its usage. The user only needs to take several images of a calibration pattern of known geometry from various viewpoints for a calibration to be possible. Except for a Matlab licence no additional expensive equipment is required. One of the images used in calibration can be seen in Figure 3.

As described in [10] the imaging process of a fisheye lens camera can be split into three parts:

1. A central projection of a scene point $\mathbf{X}$ to a direction vector $\mathbf{p}$.
2. A non-perspective optics transform by a function g mapping the direction vector $\mathbf{p}$ to a position on the camera sensor $\mathbf{u}'' = [u''_x \; u''_y]^T$.
3. A digitization of the position on the camera sensor $\mathbf{u}''$ to a pixel position $\mathbf{u}'$ on the final image.

The first point can be described with equation 3 where $\mathbf{P}$ denotes the (standard) projection matrix.

$$\lambda \mathbf{p} = \lambda \begin{bmatrix} \mathbf{u}'' \\ g(\|\mathbf{u}''\|) \end{bmatrix} = \mathbf{P}\mathbf{X} \tag{3}$$

The mapping of the direction vector to a position on the camera sensor is defined in [10] as a Taylor series missing the second term and can be seen in equation 4.

$$g(\|\mathbf{u}''\|) = a_0 + a_2\|\mathbf{u}''\|^2 + \cdots + a_N\|\mathbf{u}''\|^N \tag{4}$$

In the third part of the imaging process the fact that sensor and lens axis are not necessarily aligned is taken into account. Fixing this alignment exactly requires an Homography however as shown in [11] an Affine transformation, as can be seen in equation 5, supplies a good approximation.

$$\mathbf{u}'' = \mathbf{A}\mathbf{u}' + \mathbf{t} \tag{5}$$

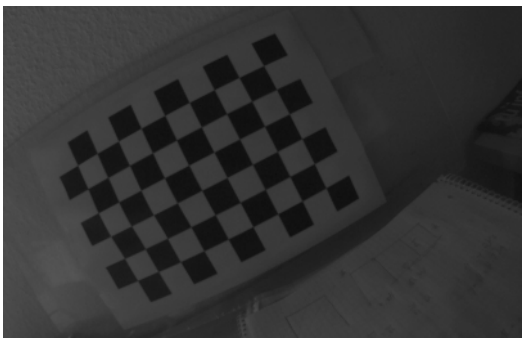
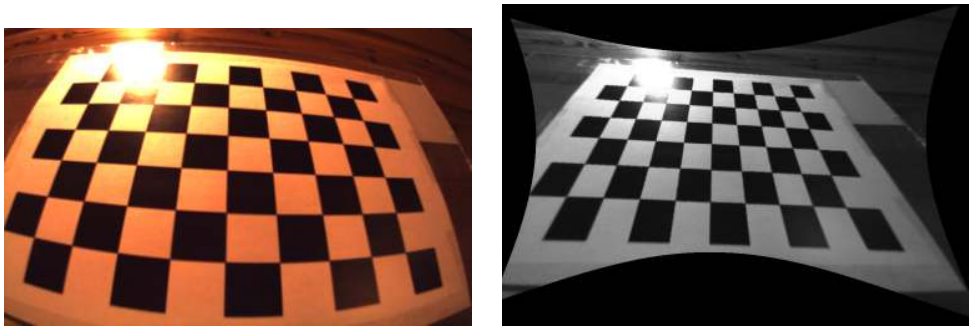


Figure 3: Example for an image used in calibration of the webcams.

5 Melding the real and the virtual

The last remaining steps to achieve augmented reality is to meld the already created virtual reality with the images from the cameras and displaying them on the screen. To achieve this we first render the undistorted camera image of the respective eye orthographically. Then the virtual scene is rendered on top for the same eye as described in the previous sections. Once the images for



(a) Image before undistortion. (b) Undistorted image in grayscale.

Figure 4: Results of the undistortion step.

both eyes are rendered they are passed on to the Oculus SDK that applies a barrel distortion and displays the images on the screen. The barrel distortion is needed to counteract the pincushion distortion introduced by the lenses inside the HMD.

A final calibration of the HMD is needed to get rid of misalignment of the camera images for the left and right eye. To do this the wearer looks at the calibration pattern and adjusts the offset of the left and right camera images using the keyboard until no blurry offset remains visible.

6 Conclusion

Combining all the described steps reveals the final HMD augmented reality pipeline after all calibration has been done:

1. Get rotation readings from HMD driver
2. Capture images on eye cameras
3. Retrieve image from first eye's camera
4. Undistort image on GPU
5. Render image with offset orthographically
6. Move first eye's virtual camera according to rotation readings
7. Render virtual scene for first eye on top
8. Repeat steps 3 to 7 for the other eye
9. Apply barrel distortion to both eyes' images
10. Display on the HMD screen

We have shown that it is possible to upgrade a virtual reality HMD to an augmented reality capable device using two front mounted cameras. Immersive 3D augmented reality can be achieved using our application and such a device. The hardware modifications can be seen in Figure 5. Our application supports multiple textured virtual objects of any geometry based on triangles. Objects can be animated using key frames and linear interpolation in between. It incorporates head rotation measured by the accelerometers in the HMD. Ambient and diffuse lighting is used to illuminate the virtual objects. Using the undistorted camera images our application creates an augmented reality by combining these with the virtual world. Figure 6 shows a

still frame from the final result of our work. A short overview of the program code can be found in appendix A.

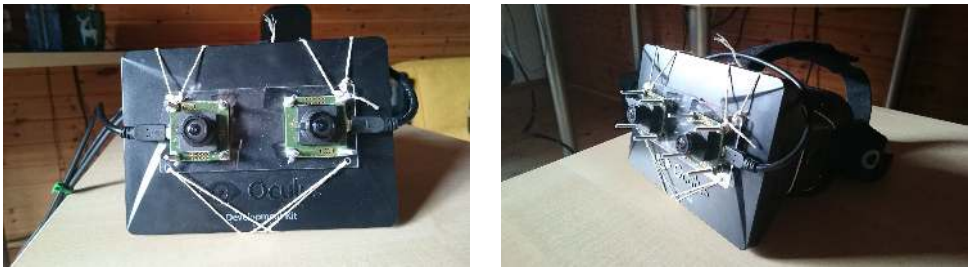


Figure 5: Hardware modifications.

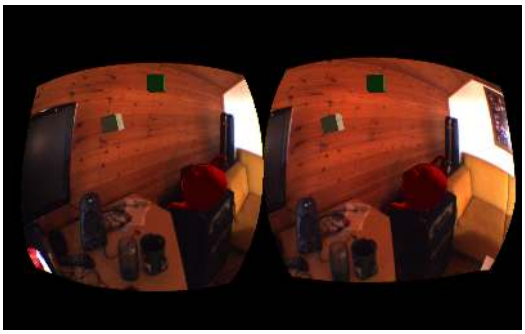


Figure 6: Still frame of the image displayed on the Oculus DK1 screen.

7 Ongoing and future work

Currently only the HMD rotation is measured and taken into account when moving the virtual cameras. Since almost any natural pose change incorporates some head translation the immersion suffers if it is not taken into account. Therefore a big improvement can be made when the HMDs translation is also measured and transferred into the virtual world. Measuring the HMDs position could be achieved by the computer vision method LSD-SLAM [12] that can be applied to one or both cameras video streams. The upside of this method of measurement is that it does not require any further hardware.

Another point that can be improved is the brightness and capture frequency of the cameras. The currently used IDS uEye UI-122-1LE-C-HQ cameras only provide about 25 frames per second in an averagely lit indoor environment without loosing too much brightness to impair vision. This is

not enough to provide a smooth appearance of movement, especially for fast head rotations. The best but also most expensive way to alleviate this flaw would be to use cameras with higher ISO. Furthermore, the framework for loading and saving virtual objects and textures can be improved, since it does become slow for many or simply single large objects and especially for large textures. The currently used wavefront format is human readable but also very slow to load. Using a more compact binary format would vastly improve startup time.

References

[1] W. Steptoe, S. Julier, and A. Steed, "Presence and discernability in conventional and non-photorealistic immersive augmented reality," in *Mixed and Augmented Reality (ISMAR), 2014 IEEE International Symposium on*, pp. 213–218, IEEE, 2014. ii

[2] J. Rolland and H. Hua, "Head-mounted display systems," *Encyclopedia of optical engineering*, pp. 1–13, 2005. 1, 2

[3] I. E. Sutherland, "A head-mounted three dimensional display," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pp. 757–764, ACM, 1968. 1

[4] D. Scaramuzza, A. Martinelli, and R. Siegwart, "A flexible technique for accurate omnidirectional camera calibration and structure from motion," in *Computer Vision Systems, 2006 ICVS'06. IEEE International Conference on*, pp. 45–45, IEEE, 2006. 1

[5] L. G. Roberts, *Machine perception of three-dimensional soups*. PhD thesis, Massachusetts Institute of Technology, 1963. 2

[6] J. Bloomenthal and J. Rokne, "Homogeneous coordinates," *The Visual Computer*, vol. 11, no. 1, pp. 15–26, 1994. 2

[7] F. L. Kooi and A. Toet, "Visual comfort of binocular and 3d displays," *Displays*, vol. 25, no. 2, pp. 99–108, 2004. 2

[8] Oculus VR LLC, *Oculus Developer Guide*. 3

[9] R. Kooima, "Generalized perspective projection," *School of Elect. Eng. and Computer Science*, pp. 1–7, 2008. 3

[10] D. Scaramuzza, *Omnidirectional vision: from calibration to robot motion estimation*. PhD thesis, Citeseer, 2008. 3, 4

[11] D. Scaramuzza and R. Siegwart, *Monocular omnidirectional visual odometry for outdoor ground vehicles*. Springer, 2008. 5

[12] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *Computer Vision–ECCV 2014*, pp. 834–849, Springer, 2014. 7

A Program code description

The augmented reality application was written in C++ for Microsoft Windows 7 & 8 using Microsoft Visual Studio 2013 Community. This project depends on the Oculus SDK version 0.4.4 for interfacing with the HMD, the IDS uEye API for utilizing the cameras efficiently and the Microsoft Windows 8.1 development kit for DirectX11. The code is publicly available under https://github.com/MaXvanHeLL/ARift.

At the start of the program first the GraphicsAPI and the ARiftControl objects are created. Following this a new thread is started that encapsulates the whole application. The main function then continues to keep running in an endless loop until ARiftControl.keepRunning() returns false. In the other thread the OculusHMD instance, the DirectX window and rendering is set up. Once the setup is successfully completed the main loop of the application initiated. The main loop starts off with checking for windows system messages (keystrokes, mouse movements, application closure commands,...). If there is a keyboard stroke incoming the handling of said input is handed over to the ARiftControl instance. Then images are captured on both cameras and saved to memory. Finally the virtual and real scene is rendered by the GraphicsAPI object instance before the loop repeats.

If the application received a WM_QUIT message the main loop is broken and the shutdown is initiated. During the shutdown first the GraphicsAPI instance is destroyed. Then the ARiftControl is informed that the loop in the main function shall be shut down. Before the thread is destroyed the OculusHMD instance is destroyed. The ARiftControl instance is destroyed in the main function as soon as the main loop ends.

In the following all implemented classes and their purpose are described shortly.

ARiftControl
: Retrieves and interprets the user interface input. Currently only keyboard input is considered.

IDSuEyeInputHandler
: Manages the control of the cameras via interfacing with the camera driver. Image capture and retrieval as well as camera settings can be changed using this class.

OculusHMD
: Provides an abstraction layer for the Oculus HMD SDK. It initializes the HMD and retrieves the rotation measurements. Once the scene

rendering in GraphicsAPI is complete the OculusHMD class passes the images on to Oculus SDK for predistortion and display.

GraphicsAPI
Creates, updates and destroys the DirectX scene. In the constructor the scene is set up, the models are loaded and placed. While the application is running the scene is updated according to user input and HMD rotation measurements. When the application is prompted to close this class handles the destruction of the scene and the cleanup.

Camera
The parameters and calculations needed for a virtual camera for standard projection are enclosed in this class.

HeadCamera
This class is derived from Camera. The calculations needed to get from the head rotation update to the camera parameter update are provided by this class.

Model
Represents a model in the virtual scene. A model's vertices, scale, position, rotation and texture are pooled together in this class. It further allows to specify and progress a model's animation. The loading of a model from a wavefromt file is also part of this class' methods.

Texture
This class represents a texture in the virtual scene. It provides methods to load it from a *.dds file and updating it from a camera image.

RenderTexture
Initializes, updates and destroys a render target that can be accessed as a texture. All rendering is done to such a RenderTexture whose texture is then passed on to the Oculus SDK.

BitMap
To meld virtual and real scene the camera images need to be rendered, this class provides the required functionality. It is a virtual object - a rectangle - that is always placed so that it fills the respective eyes full field of view. The camera images are then rendered using an undistortion shader onto this rectangle as a texture. Finally a orthographic projection is applied to it in GraphicsAPI.

Shader
Loading, compiling and applying shaders is encapsulated here. The

application compiles and loads a lighting pixel and vertex shader for the virtual objects and a undistortion pixel shader for the camera images from hlsl shader files.

EyeWindow
For debug purposes a window for each eye can be provided.

Lighting
Information about the ambient and diffuse lighting that is calculated in the lighting shader is saved here.