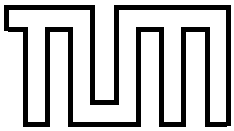
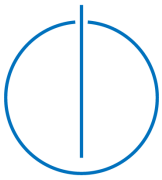


Technische Universität München
Fakultät für Informatik

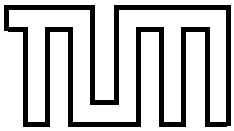


Systementwicklungsprojekt

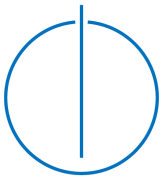
”Window into a virtual world” screen concept

Homemade CAVE environments

Nicolas Heuser



Technische Universität München
Fakultät für Informatik



Systementwicklungsprojekt

”Window into a virtual world” screen concept

Homemade CAVE environments

Nicolas Heuser

Aufgabenstellerin:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Marcus Tönnis, Dipl.- Inf.
Abgabedatum:	31. 7. 2008

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 31. 7. 2008 Nicolas Heuser

Zusammenfassung

Bei Visualisierungen in CAVEs und auf Powerwalls befindet sich der Benutzer im allgemeinen nicht zentriert vor dem Bildschirm. Die Darstellung von dreidimensionalen Objekten muss dann die relative Position des Benutzers zum Bildschirm berücksichtigen. Mit der Bezeichnung Fenster in eine virtuelle Welt"wird diese Beziehung ausgedrückt. Der Benutzer kann den Bildschirm als ein Fenster oder Rahmen betrachten, das die dahinterliegende virtuelle Welt zeigt. In diesem Projekt wird ein generelles Anzeigekonzept präsentiert, das ein breites Spektrum an Anwendungsmöglichkeiten, von großen CAVEs und Powerwalls bis hin zu tragbaren LCD- oder TFT- Bildschirmen, ermöglicht. Die Erweiterung der Ubitrack Bibliothek, zur Unterstützung dieses Darstellungskonzeptes, ermöglicht die einfache Wiederverwendbarkeit in anderen Projekten.

Abstract

3D visualizations in CAVEs and Powerwalls are situations where the view frustum, describing the projection of the scenery onto the screen depending on the viewers position to the screen, is in general asymmetrical. The wording ”window-into-a-virtual-world” is used to express a dynamic recalculation of the view frustum. Based on the position of the viewer and the pose of the screen so that the view on the display appears as showing the virtual scenery behind the area occluded by the display. This project implements a generic ”window-into-a-virtual-world” screen concept, that allows establishing a wide range of applications, ranging from CAVE environments over powerwall setups to portable LCD screens. Through extension of the Ubitrack library by a new pattern, that models the view frustum, easy incorporation into further systems is enabled.

Contents

List of Figures

1 Introduction
Window into a virtual world

In both Augmented Reality (AR) and Virtual Reality (VR), displays are a very important interface between the applications and the user. VR is the presentation of an interactive, computer generated world. The user is only interacting with the virtual environment. AR is extending reality with context based virtual information, while the user is supposed to interact the real world. The information is shown on one or more display devices. For example a regular computer display, a head-mounted display, large projectors or an array of any of these. Virtual objects and environments have to look and behave realistic when the user is interacting with it. So, when the user is moving, the virtual objects have to be shown according to the user's position in the virtual world and the display's position and orientation in both, the virtual and the real world. Today many different display setups and configurations are available to enhance the user's experience while working wit VR and AR applications. There are two main categories of screen setups: static and moveable displays. Static displays, like the regular computer screen, are fixed in space and if, like in most regular use cases, the user is also static in front of the screen, only the displayed virtual scenery is moving and rotating around the user. Head movement of the user is typically ignored. Moveable displays are most commonly attached to the user, for example to the user's glasses, and allow the use of the display while the user is moving. The display paradigm, presented in this work, allows independent positioning and orientation for both, user and display, and treats the display like a window into a virtual world that is positionally linked to the real world [13]. Consider standing in front of a window, which is your display. Moving to the left reveals more of the right scenery behind the window.

(a) View from left. (b) Middle view. (c) View from right.

Figure 1.1: Example views through a window with moving viewpoint.

Figure 1.1 shows a regular window with 3 different view positions. Note that not only the scenery shown through the window is changing, but also objects in front of the window

pass by. The metaphor, to move the virtual view position the user has to move his head, is a lot more intuitive to use than to maneuver with a mouse, keyboard or other input devices.

3D perception is based on the parallax of the eyes. Each eye sees a slightly different image, the only difference being a small positional offset. Then our brain can reconstruct the depth information of the scenery based on comparing the position of identical features in both images and on intrinsic information the user has about certain objects, like the size of a human or a car.

To display virtual information at the right position, it's necessary to know the position of the user's eye and of the display, which can be achieved by tracing known features of the user and the display. This process is called "tracking" and emits the pose of the user's head and of the display. Powerwalls and CAVEs, large, multi-display devices for visualization, are subject of many scientific researches and are also in used productive environments, as they allow a much more immerse experience of virtual worlds [11]. CAVE environments are still rare, not only because of the high hardware costs, but also because CAVEs are only small scale solutions, enabling the user to move freely in a small room, typically less than 5m x 5m.

With increasing performance gains of end-user graphic hardware and decreasing costs of video projectors, it becomes more realistic for small businesses and scientific institutions to build their own Powerwall and CAVE systems. On the other hand, there is still a lack of applications for these setups. This is partly based on the reason that developers don't have access to those systems and on their high costs. Tracked see-through displays, either hand-held or head-attached, for tracked users will allow powerful augmentation of the real world, but were not used or tested for this work.

This work presents a very generic approach to present 3-dimensional information on a display, depending on the display's position and orientation in real and in virtual space, honoring the viewer's position relative to the display. An implementation of this approach in a real world application is shown as an example.

2 Fundamentals

This section is covering the fundamentals of display environments and tracking, needed for this project. At first, a glance at common display scenarios for AR and VR is given. Later we present how the position and orientation of objects in space can be retrieved and how a virtual world can be matched and aligned to the real world.

2.1 CAVE automatic virtual environment

CAVE is a recursive acronym for CAVE automatic virtual environment. The first CAVE environment was demonstrated in 1992 at the Siggraph conference by Thomas A. DeFanti, Daniel J. Sandin, and Carolina Cruz-Neira (see [19] and [12]). Today CAVEs are a synonym for a box shaped room with two or more rear projections per side wall. Common setup are five-sided: ceiling, floor and 3 walls, but the number of sides can vary. Low cost CAVEs might even have only two sides [18]. Simple versions assume the user is standing at a fixed position inside the room and can only look around. In most modern systems, the user can walk freely inside the CAVE and the displayed images are updated according to his position, so the user has the illusion to really walk through a virtual environment.

For an immersive experience, stereoscopic rendering is very important. For obvious reasons projectors have to be outside the CAVE (as seen in figure 2.1), so rear projection canvases have to be used, which should also preserve the polarization of the light. Polarized glasses can then be used to present different images to for each eye. As for every wall and eye one projector is needed, most systems only support one concurrent user. Stereoscopic rendering is further discussed in section 5.7.

2.2 Powerwall

The term Powerwall is mostly used for very large displays. Most common they are constructed by creating an array of several smaller displays or projectors. To not have the users cast shadows onto the wall, these systems are mostly rear projected, which again, increases the cost of the system, as better canvases are needed. Regular Powerwalls also support only one simultaneous user due to the fact that the shown content is drawn from the perspective of the viewer and other spectators will note a parallax error when looking at the display.

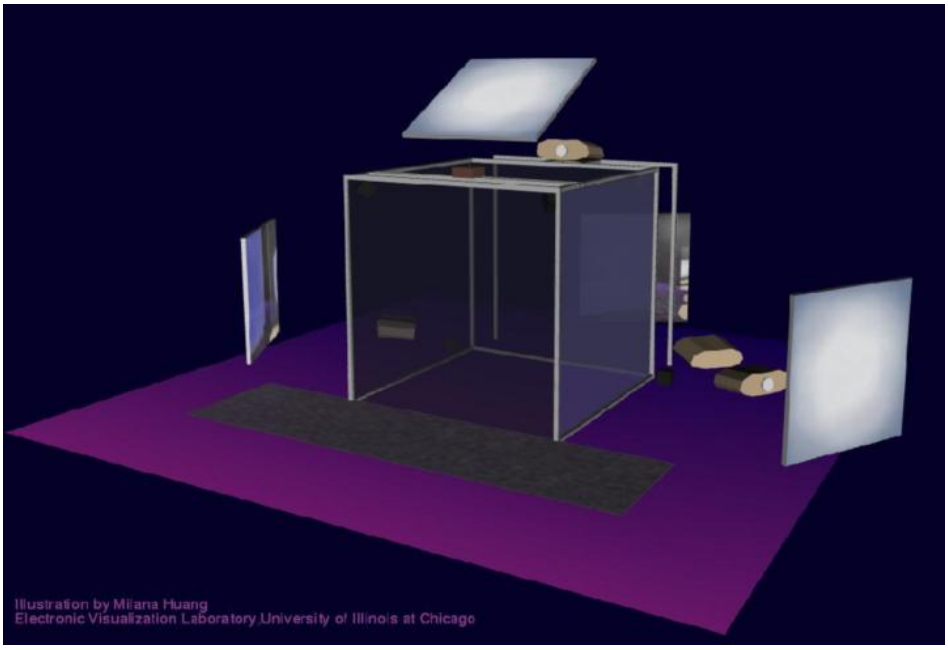


Figure 2.1: Schematic view of a four sided CAVE. Illustration by Milana Huang, Electronic Visualization Laboratory University of Illinois at Chicago.

Figure 2.2: Man standing inside a CAVE.

2.3 Uses for CAVEs and Powerwalls

The range of possible applications for CAVEs and Powerwalls is huge. But due to the high cost of acquisition, these systems are mostly used in a professional engineering context. Especially with stereoscopic rendering, designers can better visualize 3d models without the need of creating a real life model. The immersive experience of 3D world or objects helps to better understand the structure and the composition of models.

2.4 Tracking

Every virtual 3D world has a point called origin. All coordinates in the virtual world are described with a spatial transformation from the origin. For simplicity let our origin be $(0, 0, 0)$. There is no known origin for our real world, but one can simply define an arbitrary point to be the world origin and express all real world dependencies relative to this origin. If an application is aware of the position and orientation of the user's head in the real world, we say the user is head-tracked. Gathering this information can be done in many different ways today with differences in accuracy and latency. Many modern tracking technologies use hardware attached to the user to perform or aid the tracking process.

The deployment environment, as well as the testing environment, has a tracking system from the A.R. tracking GmbH, which uses light reflecting marker balls and several infrared cameras for tracking. If two cameras can see one ball unambiguously, the position of this ball relative to the cameras can be calculated by triangulation. If several balls are attached to one marker and visible to two cameras, the system can also determine the orientation of the target. The system calculates the pose of a given marker target with 60Hz and pushes these values via network to one or more consumers. The approach forces the user to attach markers to his body to be tracked. For this work, markers attached to glasses or to a helmet, were used to extrapolate the position and orientation of the viewer's eyes with a certain offset.

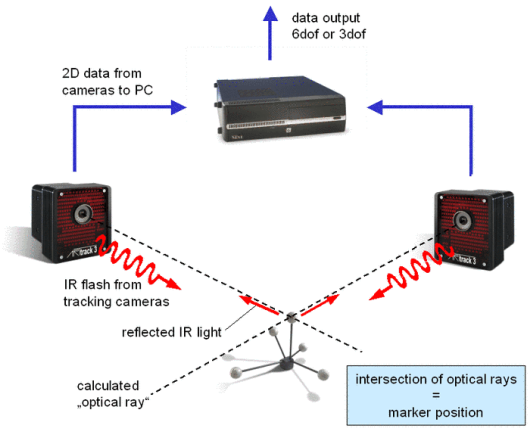


Figure 2.3: Optical tracking with A.R.T. Image is courtesy of A.R. Tracking GmbH [9]

2.5 Registration

If we want to combine virtual and real objects, the coordinate systems of both worlds have to be matched and aligned, so that the virtual information can be placed precisely in space. This process is called room calibration. We now have a transformation from real world coordinates into virtual coordinates and vice versa.

Now the tracking process reports the position of the tracked features relative to our real world origin, which are not always at exactly the position we are actually interested in. To get the position of the eye of the user, a marker is attached to the user's head and the offset from the marker to the eye is measured. This offset is added to the tracked head's position to get the real position of the eye. This step is called eye calibration.

As with eye calibration, the marker attached to the tracked screen might be placed at anywhere on the device. At screen calibration the relative offset from the screen marker to the 4 screen corners is determined. We also get the dimension of the screen.

Figure 2.4 shows a spatial relationship graph (SRG) [15] for the screen calibration process. A SRG is a directed graph, with nodes representing the objects in the scene. The edges between the nodes represent the translational and rotational offset between the corresponding objects. Traversing an edge in the opposite direction inverts the corresponding transformation.

The position of three screen corners and of the marker relative to the world origin is measured. But the position of the screen corners should be relative to the screen marker. By traversing the edges in the graph from the screen corner to the screen target, we get the relative offset of the corners to the marker. For head calibration the offset from the target, that is attached to the head, to the users' eyes is measured. As we are interested in the focal point of the eye, which lies inside the eye, the offsets must be estimated or advanced calibration methods like SPAAM [21] must be used.

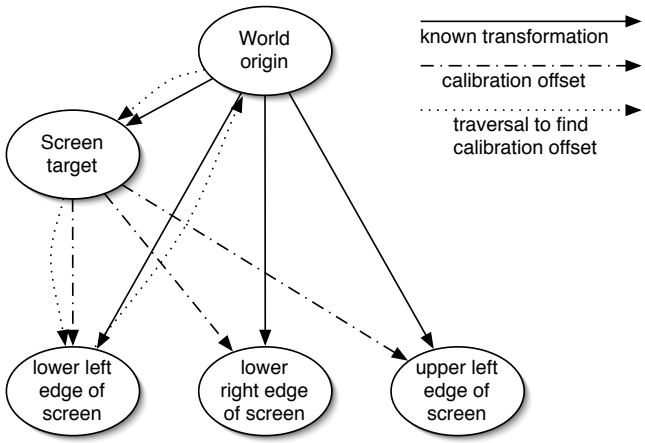


Figure 2.4: Spatial relationship graph of the screen calibration process.
Nodes represent physical objects and the edges represent the corresponding spatial transformations.

3 Specifications and requirements

This section will specify general requirements for the algorithm. Later an overlook of the target environment for deployment and the used development environment will be given.

3.1 Software Requirements Specification

To provide the illusion of reality in computer graphics, the scenery must be drawn in realtime with a preferably high frame rate. People perceive a stuttering of motion at low frame rates differently and in static scenes it's less obvious, but below 15 frames per second, movements generally do noticeably stutter. So wording "realtime" is referred here as being fast enough to render the scene without a noticeable stuttering. Computation time for a rendered frame depends on the complexity of the geometry shown, the amount of vertices and polygons. The computational overhead for this screen concept should be as small as possible. Today several major 3D graphic APIs exist on the market, most famous being OpenGL [5] and DirectX [2]. As there are only minor differences in the math used for by these APIs, we present the general concept without the use of any 3D APIs.

As example application for this work, the Zusbau tool is used. Zusbau is a in-house development of the BMW AG [10] and is not publicly available. It is a 3D visualization application for car models during the design process to verify and evaluate surfaces. The main focus is on accuracy and quality of the visualization. The application consists of a client and a server part. The server contains the UI and controls the display clients which are responsible for the 3D presentation. The server UI is a 2D top down view of the scenery with the car model in the center. Eye and target point are indicated by icons and a numerical view of the coordinates. All parameters can be changed interactively in realtime. Connected clients show the scenery based on the view point and the target point honoring special screen parameters to enable Powerwall support with translated projection planes. Client/server communication is socked based and therefore Zusbau can be run distributed on several computers.

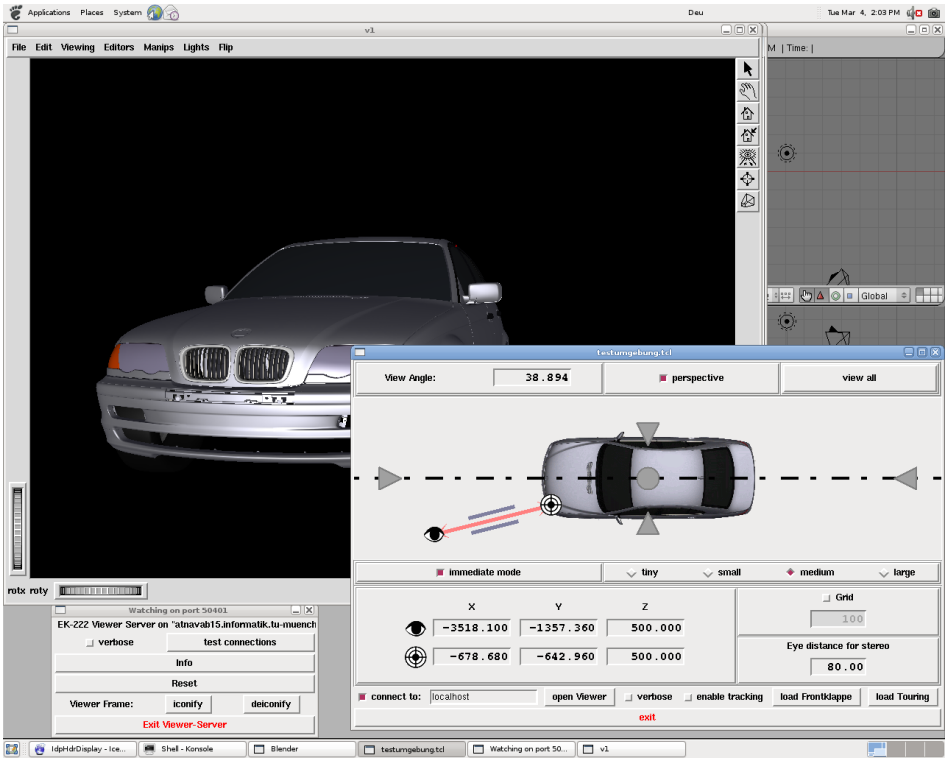


Figure 3.1: Screenshot of Zusbau control server and viewer client. Image is courtesy of BMW AG.

Communication is bidirectional. The user can use the mouse to change view direction and position on the client. These changes are propagated to the server and shown in the UI. All other connected display clients are updated accordingly.

Zusbau is to be extended to enable it to display the scenery according to the viewer's position relative to the screen, allowing Zusbau to be used in CAVE environments and on Powerwalls.

3.2 Test and deployment environment specification

As in most software engineering projects the destination environment for deployment and the development and testing environments differ.

3.2.1 Development and testing environment

For development of the example implementation, two setups where used. One setup was using one tracked TFT display with an external tracking system, similar to the one of the final environment for deployment. The other one was a regular notebook with built-in webcam for marker-based headtracking. Figure 3.2 shows the hardware used for this setup. While the ART system was very close to the actual target environment and allowed a very wide

working space, using a notebook with build-in webcam was a very good way of testing during development, when the ART system was not available.

At first the algorithm was implemented as a stand alone component and tested with an external renderer. The actual implementation into Zusbau was done in a second step. The test application was extended to support stereoscopic anaglyph rendering. This allowed to test 3D perception with very low cost paper glasses.

The marker attached to the glasses allows the calculation of the location and orientation of the users' eyes when visible on the picture from the webcam.

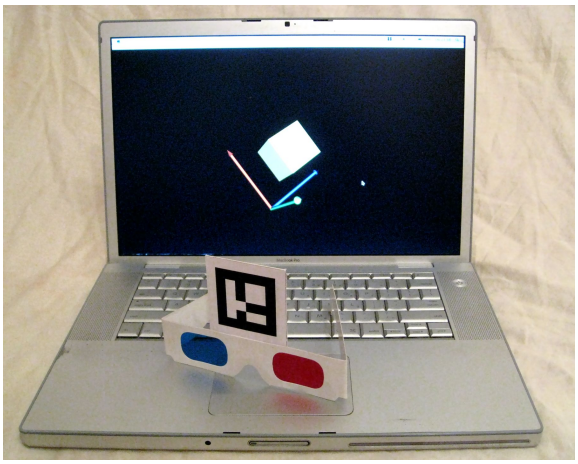


Figure 3.2: Notebook with build-in webcam and glasses with attached marker.

3.2.2 Deployment environment

The target deployment environment is a five sided CAVE, rear projected with ART head-tracking. Sides are approximately 3 meters long. For each side, two projectors with different polarization are used to accommodate for stereo separation. Each projector is attached to one computer which is only responsible for rendering that particular screen. The CAVE user has to wear tracked spectacles with polarized glasses, so that each eye can only see the image displayed by one projector. In this setup only one user can use the CAVE simultaneously.

4 Offaxis projection

Consider taking a photograph of a person. The resulting image is a 2D representation of the scenery in front of the camera, depending on the position of the photographer, the view direction of the camera, aperture, shutter speed etc. In the moment of taking the photograph, the geometry of the scene in front of the camera is projected through the lens onto the film. Such a photography is similar to a snapshot from a virtual world. Here a rendering process maps virtual 3D objects to a 2D screen presentation. Geometrical objects in 3D graphic consist of simple polygons and triangles, which themselves, are defined by their corner points, also called vertices. The next sections show the mathematical background to setup virtual cameras.

4.1 Cameras

There a two different types of projection. Perspective projection is similar to the way we see objects, as they appear to be smaller the larger the distance to the viewer is. With orthogonal projection the size of the objects after the projection is independent of the distance to the viewer. For AR and VR information should be displayed in a natural way, so we restrict the discussion to perspective projection. The virtual camera has a plane to project the geometry onto, the screen plane. A rectangular area, the viewport, from this plane is selected for the display. A so called view volume is created by rays starting from the eye point going through the edges of the viewport. To achieve the illusion of depth, if two objects overlap from the eye point, these two objects must drawn in the correct order. If an object further away is drawn over objects closer to the viewer, this will obviously destroy the depth illusion. A so called "depth buffer" can be used to store the depth information of every rendered pixel. New pixels are only rendered if they are in front of, or closer to the eye point, than the previous rendered pixel at this position. The information in the depth buffer is then updated accordingly. The precision of the depth buffer is important for distinguishing between surfaces near each other. The precision is affected by the used data type for the depth buffer and two planes, a near clipping plane and a far clipping plane, which are perpendicular to the line of sight of the eye. The clipping planes are not needed for the projection itself. The volume resulting from the pyramid cut by the two clipping planes is called view frustum (see figure 4.1). All objects inside the view frustum will be rendered to the screen. Together with the eye point and the projection plane the construct is called camera model.

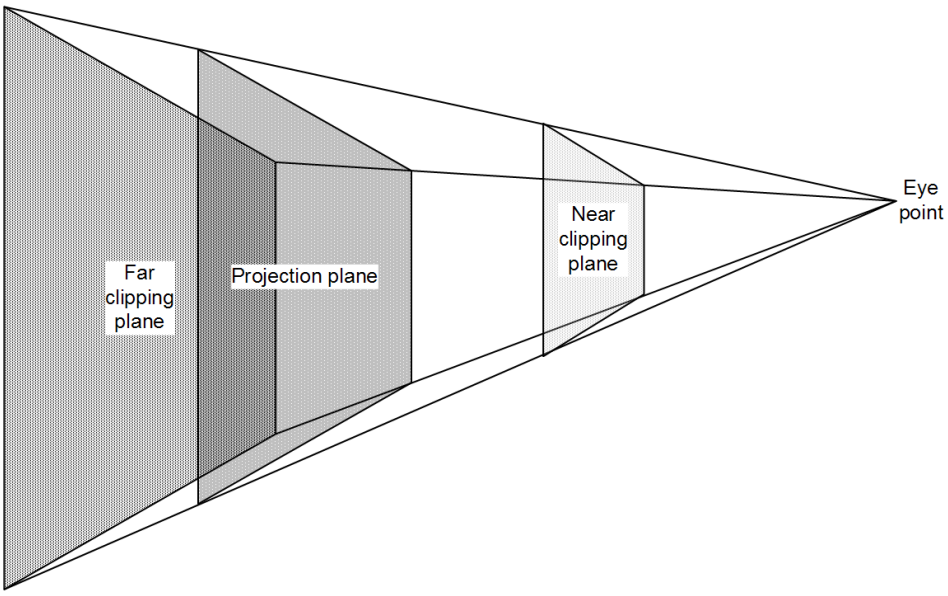


Figure 4.1: Perspective viewing volume with projection plane, near and far clipping plane.

4.2 General camera model

Let's get back to the photograph analogy. When taking a photograph, you first consider what you want to see on the photo and from where. The camera is positioned on a tripod and pointed at the scene. This is called viewing transformation, had has to be done with the virtual camera as well. Let $\vec{E}$ be the eye point, the focal point of the camera, and $\vec{U}$ a normalized vector pointing upwards from the eye point, $\vec{L}$ pointing left and the view direction $\vec{D}$, the direction the camera is pointing. So $\vec{L}$, $\vec{U}$ and $\vec{D}$ span a right handed coordinate system. The resulting Matrix $R = (\vec{L}|\vec{U}|\vec{D})$ is orthonormal. Now we can obtain the view matrix M_{view}:

$$M_{view} = \left(\begin{array}{c|c} R^T & -R^T\vec{E} \\ \hline \vec{0} & 1 \end{array}\right) \tag{4.1}$$

4.3 Projection

The photographer now chooses a camera lens or adjusts the zoom. The moment a photograph is taken, the shutter opens and the film is exposed to the incident light of the scenery projecting a picture of the scene onto the film. Technically this can be described with a projection matrix that maps 3-dimensional points onto a 2-dimensional plane. As the target API is OpenGL (see section 3.1) we use the matrix 4.2 from [8, p.131].

$$M_{proj[-1;1]} = \begin{pmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.2}$$

$left$, $right$, top and $bottom$ are the distances from the axis ray to the near clipping plane edges. $near$ and far are the distances of the corresponding clipping planes to the eye, needed for the depth buffer.

OpenGL maps the values in the depth buffer to the range [-1;1]. For DirectX or other APIs mapping the depth values to [0;1] the matrix would be:

$$M_{proj[0;1]} = \begin{pmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.3}$$

4.4 Offaxis view frustum

The previously discussed camera model is a special case of the offaxis frustum calculation. Regular screen concepts require the user to be centered in front of the screen at a fixed distance. In head-tracked environments, the user is moving around in the room and the view frustum has to be adapted accordingly. The effect is like looking out of a window and moving around except that in our model the window can also be moving. See figure 1.1 for a real world example.

For the perspective projection, a matrix is needed to project the vertices after the model- and view-transformations onto the screen plane. For this, the screen position, orientation and dimension is needed. This can be calculated from three of the four screen corners. Below the lower left (P_{ll}), lower right (P_{lr}) and upper left (P_{ul}) edges are used. The screen width is given by $width = |P_{lr} - P_{ll}|$ and height is defined by $height = |P_{ul} - P_{ll}|$.

For a tracked display the screen corners are saved relative to the screen target during screen calibration (see section 2.5). For this calculation they need to be transformed back into world coordinates as shown in figure 4.4. Notice the similarity to figure 2.4, only the direction we traverse the graph has changed.

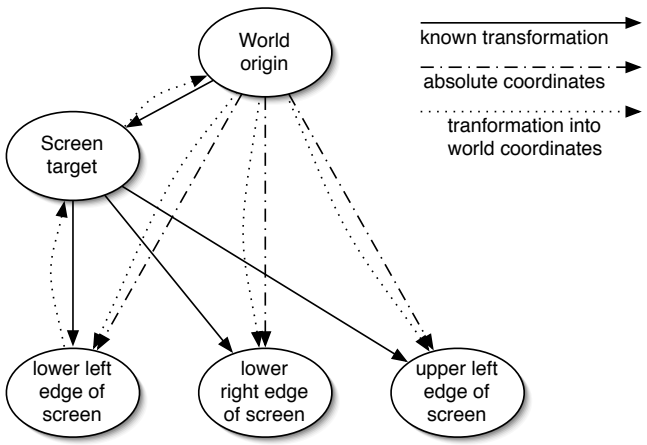


Figure 4.2: Getting the absolute position of the screen corners.

We need a way to express the rotation of the screen relative to the viewer. This can be done by taking to perpendicular vectors in the screen plane and other vector being perpendicular to the first two. These three vector form a coordinate system local to the screen. For simplicity let the first two vector be the edges of the screen:

$$\begin{aligned} X_s &= \| P_{lr} - P_{ll} \| \\ Y_s &= \| P_{ul} - P_{ll} \| \\ Z_s &= X_S \bullet Y_S \end{aligned}$$

These three vectors are a rotational transformation from the world coordinate system into the local screen coordinate system M_s.

$$M_s = \begin{pmatrix} \begin{pmatrix} X_s \end{pmatrix} & \begin{pmatrix} Y_s \end{pmatrix} & \begin{pmatrix} Z_s \end{pmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

The relative position of the viewpoint E to the lower left corner of the screen is $E_s = E - P_{ll}$. In the next step the distance from a ray, which is perpendicular to the screen plane going through the viewpoint, the view vector, to the edges of the near clipping plane needs to be calculated. This can be explained with the theorem of intersecting lines. The distances of the edges of the viewport to the view vector is scaled to match the near clipping plane. The distance of the viewpoint to the screen is $d = E_s \bullet Z_s$.

$$\begin{aligned} L &= E_s \bullet X_s \\ R &= width - L \\ B &= E_s \bullet Y_s \\ T &= height - B \\ left &= -L * near/d \\ right &= R * near/d \\ bottom &= B * near/d \\ top &= T * near/d \end{aligned}$$

Figure 4.3 shows these distances in a schematic overview of a perspective view volume.

Now equation 4.2 or 4.3 can be used to calculate the projection matrix for the given frustum.

To get from model space into view space the following product needs to be calculated:

$$M_{total} = M_{proj} M_{view} M_{model}$$

M_{model} is the transformation of the geometry in model space and used to place the virtual objects in space. M_{proj} is already known, so we need M_{view}. The rotational part of M_{view} is M_s^{-1} and the eye position has to be taken into account. So M_{view} is build like:

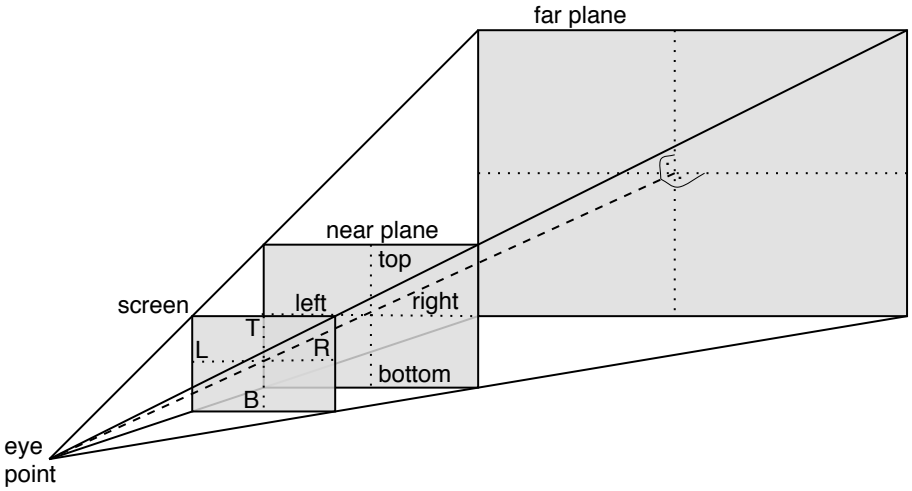


Figure 4.3: Distances of the view vector to the near .

$$M_{view} = \left(\begin{array}{c|c} M_s^{-1} & -\vec{E} \\ \hline \vec{0} & 1 \end{array} \right)$$

Also note that M_s is an orthonormal rotational matrix, so instead of inverting it, it can just be transposed:

$$M_{view} = \left(\begin{array}{c|c} M_s^{T} & -\vec{E} \\ \hline \vec{0} & 1 \end{array} \right)$$

4.5 Limitations

The human field of view is fixed and quite limited. In cases where the user of a CAVE or Powerwall approaches the screen and gets so close that the display becomes larger than the user's field of view, the shown geometry will appear distorted. This edge case is not yet handled in the implementation but the impact in a CAVE should not be too visible as the area most CAVE users use is far smaller then the actual dimensions of the CAVE.

5 Notes about the implementation

How we did it

As already mentioned, the development and testing was done on linux platforms with off-the-shelf hardware, but most of the code is written platform independent and should be portable to other platforms easily. All mathematical operations were implemented in C++ with the uBlas Library [1]. The core component for offaxis projection has no dependencies to any 3D API.

5.1 Zusbau

The implementation was not done in the complete Zusbau system, but on a stripped down version which basically just contained the 2D control UI and the viewer process. In the full version, Zusbau is connected to a database containing information about the various meshes of the models and scenes can be composed by selecting the models. The models themselves are located on a network storage which has to be mounted on all computers involved in the control / display process. Most of the database access features to combine model elements were left out to ease integration and to have Zusbau work without a running database instance with mesh information. Except one checkbox no additions were made to the Zusbau UI. The checkbox controls wether tracking information should be accepted or not. Switching to tracked mode also switches to perspective projection, as it's the most common projection method when using a CAVE or Powerwall.

Zusbau is programmed completely in Tcl [7] with a special Inventor extension package for the scene graph management and itcl [4] for the graphical user interface.

5.2 Inventor

Open Inventor [17] is a 3D retained mode API developed by SGI to abstract the 3D rendering process. Inventor is an API standard with different available implementations. All objects, e.g. models, lights, cameras, are placed in a scene graph as so called nodes with a set of attributes. The scenery is then drawn by traversing the scene graph and rendering the nodes. Zusbau is using a special Inventor package for the visualisation process. Regular inventor files (.iv) are used to load the models and most of the interaction with the 3D geometry and camera manipulation is done with inventor nodes. Some non-standard additions have been made the Inventor package, to better support the rendering of shadows, etc. The camera model of the inventor standard has no support for arbitrarly positioned and oriented screens

like we needed for our implementation. We have an extra function in the inventor viewer object that allows to modify the viewing and projection matrix of the viewer directly. This way arbitrary frusta can be injected into the rendering pipeline.

5.3 Ubitrack

Ubitrack is a tracking framework developed at the Technische Universität München and is distributed under the LPGL license with the purpose of abstracting the tracking process from the application. Ubitrack supports various tracking system, including the A.R.T. tracking system and provides a foundation for mathematical operations in 3D space. Ubitrack also maintains a central spatial relationship graph that describes coordinate frames, trackers and trackable objects [16]. Tasks for Ubitrack are defined in .utql files, a XML file describing the various steps of the process with patterns. These patterns operate on the SRG and allow to derive new spatial relationships or modify existing ones based on one or more inputs. This work extended Ubitrack to support offaxis view frusta. Therefore, a plugin was created, which takes two inputs, the eye pose and the screen pose and outputs a projection/view matrix ready to be used in OpenGL-based render back-ends. For the example renderer in Ubitrack, a component was written to receive the projection matrix from the component for offaxis projection and injects it into the rendering pipeline. Although not a requirement, all Ubitrack components for this project compile and run on the Windows platform.

5.4 SWIG

SWIG [6] stands for "Simplified Wrapper and Interface Generator" and is a software development tool to generate interface wrappers for various scripting languages, including Tcl. Ubitrack already had a wrapper for Java so it just had to be extended to also generate wrappers for Tcl. This way changes in the Ubitrack library can easily be adapted by simply generating new wrappers instead of changing the wrapper code manually. Note that not the complete functionality of Ubitrack has been exposed to Tcl, just the basic frontend interfaces.

5.5 Offaxis projection pattern

Figure 5.1 shows the a spatial relation graph for this pattern which computes an offaxis projection matrix based on the two dynamic input values, eye position and screen position and the static parameters from the screen calibration process. The component is designed to either update the matrix if a new input value arrives or to look for new input values if the projection matrix is requested. To actually set the projection matrix, an extra component for Ubitrack has been created to inject the matrix into the Ubitrack rendering pipeline, for other renderers this has to be adapted accordingly.

For easy integration into Zusbau, not only the projection matrix is exported, but also the raw tracking information of the viewer. This way the 2D map view can easily be updated to show the latest position of the tracked viewer. This is realized with so called data sinks in the Ubitrack dataflow, which collect the data and pass it to receiver objects in the Tcl application.

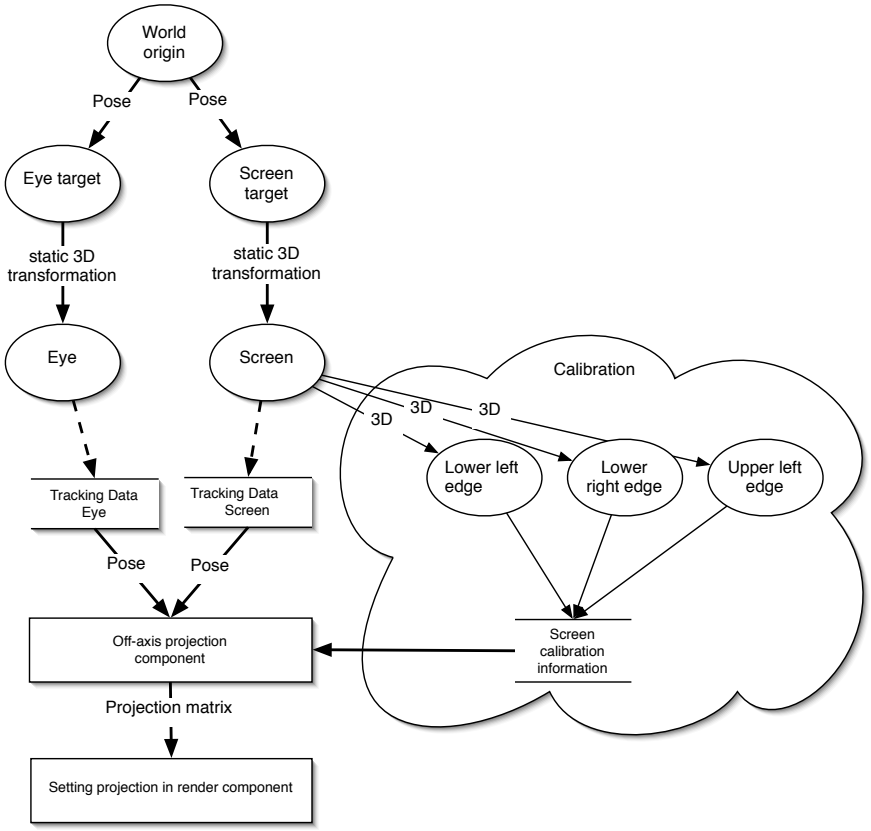


Figure 5.1: Spatial relationship graph for the Ubitrack pattern for offaxis projection.

The creation of a Tcl wrapper object that allows callback from Ubitrack into Zusbau would have not been trivial to implement, so the receiver objects have to pull their information instead. A standard A.R.T. system publishes tracking data with 60 Hz and in the current implementation we query the object with 60 Hz. Increasing the frequency of the pulling process did not noticably increase the responsiveness of the system.

5.6 Scaling

3D rendering API mostly don't have a convention how units map to real measurements. By default the Ubitrack library maps meters to units 1:1, whereas Zusbau is using one unit for one millimeter. Rotations are naturally not affected by different scaling factors, so only positions have to be modified. Ubitrack can scale tracking coordinates by patterns so the transformation can be controlled from .utql- files without any modifications in the source code. Or the scaling can be done by manipulating the model-view matrix accordingly before drawing the geometry.

5.7 Stereoscopic rendering

Stereoscopic rendering is a technique to create the illusion of depth for a rendered scene by using two different 2D images showing different perspectives of the same scene [20]. The deviation of the two images should be the same, or at least similar, as the eye distance of the viewer. Head mounted displays often use two displays, one for each eye to achieve this effect.

So called "shutter glasses" use glass containing liquid crystal which darkens when voltage is applied. When synchronized with the refresh rate of the display or the projector each eye can see alternately a different image. To avoid a flicker effect the refresh rate of the display needs to be very high though.

Instead of shutter glasses, glasses with polarized glass can be used where the polarization of the glass for one eye is rotated 90° to the other lens. The screen can then display two images simultaneously each with a different polarization. Polarized lenses are quite expensive and most setups use two projectors and pricy canvases which preserve the polarization of the light.

Anaglyphic 3D uses colored glasses to separate the pictures, by the cost of color quality. This is a very popular method because of the very cheap hardware costs. Viewing glasses are inexpensive and there is no special display hardware required.

Ubitrack and Zusbau do not support implicit stereoscopic rendering, but can be used to achieve this effect by using two display instances, one for each eye, honoring the eye offset. During development of this project a Ubitrack component for frame sequential rendering was proposed and partially implemented, but as this is not part of this work it will be not further discussed.

6 Performance

All tests were done on an AMD Athlon XP 2400+ with 512MB RAM and a NVidia GeForce FX 5600 graphics card with 128 MB RAM. The operating system was a Linux operating system, kernel 2.6.22 with binary NVidia drivers [3] to enable hardware 3D acceleration. GCC [14] 4.3.1 was used for compilation with the flag "-O2" to enable code optimization by the compiler. Due to dependencies of the Ubitrack library, all data structures are 64 bit aligned. The code itself was not optimized especially for performance. To measure the computational impact of the calculation of an offaxis projection a test application was written. The benchmark calculated an offaxis frustum 100.000 times with random parameters. The elapsed time was measured with the system function *clock_gettime*. One iteration of the benchmark took an average of 15 microseconds. The cpu cache was cleared intentionally before every iteration to better simulate real scenarios, where the amount of data, touched for processing, between two consecutive frames is by far larger than the cpu cache.

Most of 15 microseconds are used for the matrix multiplications of the model and view matrix, which is also necessary when not using offaxis frusta. For comparison, the setup of a regular projection matrix was measured as well and took an average of 12 microseconds. The introduced overhead for the calculation of one offaxis projection is only 3 microseconds. Due to it's dynamic nature, the offaxis projection has to be recalculated for every frame. Considering a frame rate of 60 frames per seconds, which is the usual maximum refresh rate of TFT displays, this results in a total of $60 * 15 = 900$ microseconds per second dedicated to offaxis frustum creation.

Parameter retrieval was not incorporated in the benchmark, as the multi-threaded nature of the Ubitrack library would not allow a reliable measurement of these times. Also, the parameter retrieval time itself is strongly dependent on the combination of used patterns in the workflow.

In use cases like with CAVEs, a Powerwall or head-mounted displays, not all transformations are static and could be pre-calculated and reused to speed up these special cases.

7 Conclusion

The use of displays as windows into virtual worlds becomes much more intuitive when the viewpoint is independent from the display. It allows new metaphors for navigation in virtual worlds by exploiting the human skill to navigate in a three-dimensional world. By either moving the head or display, the user can disclose parts of the scenery that where hidden before.

Demo incorporation into several applications, ranging from marker-based webcam setups with notebooks over portable TFTs and large projection walls shows the broad spectrum of possible setup scenarios.

As noted in section 3.2.1, for development a marker was attached to simple red/green anaglyphic glasses and together with a webcam a very small Powerwall was constructed. This construction was very cheap but not very unobtrusive and not usable in an everyday work environment or for public exposition. The new component for offaxis projection in Ubitrack allows the fast and cheap setup of various different display scenarios from Augmented- and Virtual Reality. Together with anaglyphic rendering this allows an easy start for small projects, demos and showcases in the field of AR/VR.

Future work could include the direct integration of of registration and calibration into Zusbau to get a better user experience for the operator. The integration of a meta pattern for stereo view into Ubitrack would be desirable to ease the setup of stereoscopic displays. For performance one could create special versions of the offaxis pattern for special cases where static transformations exist between one or more components, for example static displays for CAVE environments. The runtime of the algorithm can probably be further reduced by optimizing the vector and matrix multiplications involved.

With the advent of mobile devices with 3D accelerated graphic, integrated cameras and high-speed internet access, it will be interesting to see what applications, with the ”window-into-a-virtual-world” display paradigm, can be realized on these devices.

Bibliography

[1] Boost c++ libraries - boost basic linear algebra. http://www.boost.org/doc/libs/1_34_1/libs/numeric/ublas/doc/index.htm, last visited 6/10/2008.

[2] Directx homepage. http://www.microsoft.com/directx/, last visited 5/28/2008.

[3] Nvidia homepage. http://www.nvidia.com, last visited 7/23/2008.

[4] Object-oriented programming in tcl/tk. http://incrtcl.sourceforge.net/itcl/, last visited 04/28/2008.

[5] Opengl. http://www.opengl.org/, last visited 5/28/2008.

[6] Simplified wrapper and interface generator. http://www.swig.org, last visited 5/10/2008.

[7] Tcl developer site. http://www.tcl.tk/, last visited 6/20/2008.

[8] *OpenGL Reference Manual, Release 1*. Addison-Wesley Longman Publishing Co., Inc., November 1992.

[9] Advanced realtime tracking GmbH. Advanced realtime tracking GmbH Homepage. http://www.ar-tracking.de/, last visited 4/14/08.

[10] BMW AG. BMW Deutschland. http://www.bmw.de, last visited 07/27/08.

[11] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the cave. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142, New York, NY, USA, 1993. ACM.

[12] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The cave: audio visual experience automatic virtual environment. *Commun. ACM*, 35(6):64–72, 1992.

[13] Steven Feiner, Blair MacIntyre, Marcus Haupt, and Eliot Solomon. Windows on the world: 2d windows for 3d augmented reality. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 145–155, New York, NY, USA, 1993. ACM.

[14] GNU Project. GCC, the GNU Compiler Collection. http://gcc.gnu.org, last visited 7/23/2008.

[15] Daniel Pustka, Manuel Huber, Martin Bauer, and Gudrun Klinker. Spatial relationship patterns: Elements of reusable tracking and calibration systems. In *Proc. IEEE International Symposium on Mixed and Augmented Reality (ISMAR'06)*, October 2006.

[16] Daniel Pustka, Manuel Huber, Florian Echtler, and Peter Keitler. UTQL: The Ubiquitous Tracking Query Language v1.0. Technical Report TUM-I0718, Institut für Informatik, Technische Universität München, 2007.

[17] SGI. Sgi - developer central open source. `http://oss.sgi.com/projects/inventor/`, last visited 5/15/08.

[18] Po wei Lin, Zhi geng Pan, Jian Yang, and Jiao ying Shi. Implementation of a low-cost cave system based on networked pc. Technical report, State Key Lab of CAD and CG, Zhejiang University, 2002.

[19] Wikipedia, the free encyclopedia. Cave automatic virtual environment. `http://en.wikipedia.org/wiki/Cave_Automatic_Virtual_Environment`, last visited 4/7/2008.

[20] Wikipedia, the free encyclopedia. Stereoscopy. `http://en.wikipedia.org/wiki/Stereoscopy`, last visited 5/10/2008.

[21] N. Navab Y. Genc, M. Tuceryan. Practical solutions for calibration of optical see-through devices. In *ISMAR*, 2002.