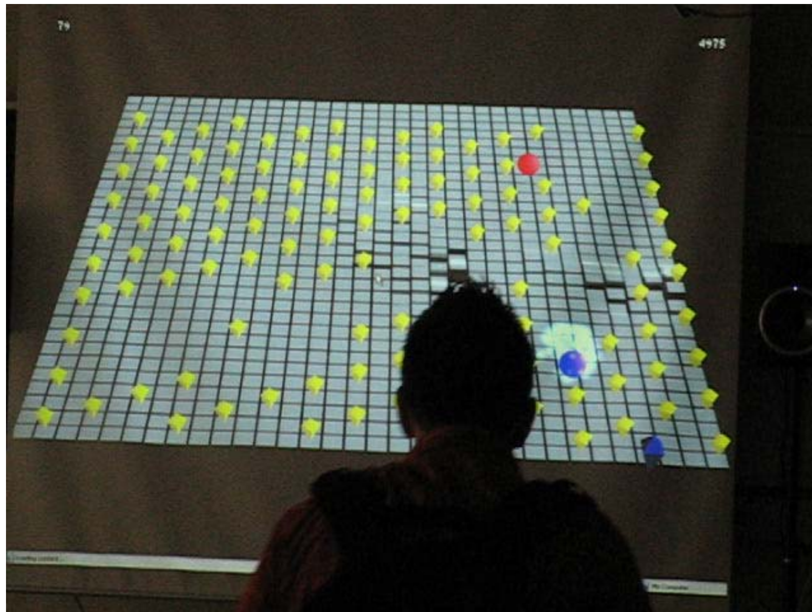
Aalborg University Copenhagen

Medialogy Cand. Sc. – 8th semester project

Project Report

Image tracking as interaction tool for a 3D virtual world in audience interaction context

An approach to networked multi-application environment



14/10/2005

Group number: 000

Student:
Smilen Dimitrov

Project Supervisor:
Stefania Serafim

AALBORG UNIVERSITET

Abstract / synopsis

In the summer of 2005, a team of Medialogy M.Sc. students, consisting of Vincent Agerbech, Brian Johansen, Allan Johansen and the author of this report, were assigned the task to prepare a game for the opening ceremony of the Medialogy studies in Copenhagen. The starting basis for the game was that it should allow audience interaction, and it resulted in a demo of a 3D world, where the interaction input was obtained via processing of live video of the audience – thereby satisfying the initial demands. The limitations of the project (like available space) required a set up of a TCP/IP networked environment, where each PC ran a specialized task, and communicated its results to the next recipient. The final recipient was eventually the 3D rendering application, which was developed in Virtools. The purpose of this report is to provide a description of the process of interaction of the audience with the 3D world, description of the parts of the input system for this interaction, and the details of the networking process between them.

Acknowledgments

I would like to thank the members of the project team, Vincent Agerbech, Brian Johansen and Allan Johansen for the great teamwork and working atmosphere during development of this project; project supervisor Stefania Serafim, and all involved with the AAUC Medialogy inauguration event, of which this project was a part of; as well as my parents, Vladimir and Todorka Dimitrovi, for their support during the course of my studies.

Table of Contents

1 Introduction 3
2 Related work in audience interaction 5
2.1 Summary 8
3 System description 12
3.1 The process of audience interaction 12
3.2 System Design 15
3.3 Usage of sound 19
4 3D interpretation algorithm 20
4.1 Images, functions and matrices...20
4.2 Surface plot of an image as 3D interpretation 23
5 Image processing algorithm for motion detection – difference based algorithm.............27
6 Input processing chain and 3D world interaction..33
6.1 Persistence effect..34
6.2 Catadioptric extension and radial distortion algorithm 35
7 Prototype gameplay...42
8 Comparison to other methods of virtual world interaction 46
9 Networking approach 50
9.1 Networking overview 50
9.2 Initial development and server/client structure...50
9.3 The camera from a network perspective 53
9.4 EyesWeb client and networking to the camera 54
9.4.1 Stream reception and extraction of frames 55
9.4.2 Decoding of the received frames 57
9.4.3 Further processing 59
9.5 Virtools server and networking to EyesWeb 60
9.5.1 Stream reception and extraction of frames 61
9.5.2 Decoding of the received frames 62
9.6 Relationship to OSI layers 63
10 Conclusion and perspectives...68

11 Appendix....73
Appendix A. Matlab example codes 73
Appendix B. Difference-based analysis example from EyesWeb 75
Appendix C. Code listing for client connection used in EyesWeb plugin 75
Appendix D. Code listing for client receiving thread used in EyesWeb plugin 77
Appendix E. Code listing for JPEG reading and decoding used in EyesWeb plugin....80
Appendix F. Code listing for server connection used in Virtools plugin 82
Appendix G. Code listing for server receiving used in Virtools plugin 84
Appendix H. Code listing for bitmap transfer used in Virtools plugin....85
Appendix I. Pinch/Spheric Distortion algorithm....87
Appendix J. Persistence effect algorithm 95
Appendix K. Screenshot of EyesWeb patch with plugins 98
Appendix L. Screenshot of Virtools patch with plugins 99
List of references 100

1 Introduction

In the summer of 2005, a team of Medialogy M.Sc. students, consisting of Vincent Agerbech, Brian Johansen, Allan Johansen and the author of this report, were assigned the task to prepare a game to be demonstrated at the opening ceremony of the Medialogy studies in Copenhagen. The starting basis for the game was that it should allow audience interaction, and it resulted in a demo of a 3D world, where the interaction input was obtained via processing of live video of the audience – thereby satisfying the initial demands. The limitations of the project (like available space) required a set up of a TCP/IP networked environment, where each PC ran a specialized task, and communicated its results to the next recipient. The final recipient was eventually the 3D rendering application, which was developed in Virtools. The purpose of this report is to provide a description of the process of interaction of the audience with the 3D world, description of the parts of the input system for this interaction, and the details of the networking process between them.

In spite of the fact that the game had its technical problems, which unfortunately appeared at the opening ceremony as well, it provided an interesting approach to interaction with a 3D world, especially in the sense that several persons can do it simultaneously – and a technical solution that demanded networking between several applications on different PCs. This document will therefore describe the process of interaction of the audience with the virtual world, the type of feedback the audience gains and to what extent it could be called immersive. It will discuss the basic idea behind the mapping of the input interaction and the 3D world, and how is it implemented in the networking environment. It will thereupon describe the details behind the data processing within the input interaction processing chain, and how it is relayed via the network.

The game was based on the interaction input was acquired by recording the audience participants. The audience members were recorded with a catadioptrically extended camera, providing a live video of a top (orthogonal) view of the auditorium (playground) along with the players. The auditorium boundaries were mapped to the boundaries of a surface, which composed the basis of a 3D virtual world where the game unfolded. Using a video beam projector, this 3D world was presented back to the audience on a large screen; whereas a sonic illustration of the game was fed back to the audience using a pair of speakers. The input video was analyzed for local motion, and areas in the top auditorium scene – where local motion occurred – could be identified in realtime. The game interaction consisted in

corresponding areas on the 3D world surface rising to follow those areas where local motion occurred – and therefore the players had the possibility to induce rise and fall of waves (or hills) in the 3D world, and move them as well – by executing a range of motions: from shaking a hand to walking across the auditorium. A model of an inert ball was placed in the 3D world, which could be displaced by an incoming "wave" – and the interaction of this ball with the other game objects collectable in the game, composed the arcade style of gameplay in the prototype. The prototype system was implemented using two Windows XP PC's, an Axis 206 network camera and a network switch connected establishing a local area network, where one of the PC's performed the image processing analysis of the camera video input, and the other rendered the 3D game world and the sound based on the processed video signal.

2 Related work in audience interaction

The starting point in the research for the game, is a paper describing several methods for audience interaction, “Techniques for Interactive Audience Participation[1]”. The authors describe several methods, some of which will be repeated here – as well as some guidelines for design of such products.

As noted in [1], involving the audience to participate is not a new concept, even in the traditional media - however, the authors feel that computer based real-time processing opens up possibilities for new types of “unscripted audience activities[1]”. There are several projects presented, all of which acquire the interaction input via real-time image processing. The original inspiration seems to come from the Cinematrix Interactive Entertainment System[9], presented at the SIGGRAPH conference in 1991. Cinematrix required the audience members to use interaction tools – reflective paddles that had a red and green side. The audience members interacted by showing a desired color of their paddles – which provided the scene that was recorded as video input. By processing the video feed, the amount and distribution of the two tracked colors throughout the scene can be obtained, and through that, the “audience members can participate interactively in activities such as maze navigation and opinion polling by displaying the red or the green side of their paddles[1]”.

Based on this system, the authors of [1] proceed with research in other applications of vision based audience participation. An important issue for them was to eliminate the need for individual audience members to use interaction tools such as the bicolor paddles used in Cinematrix, and they elaborate on three interaction modes that can be obtained by image processing of live video:

- Audience movement tracking
- Object shadow tracking
- Laser pointer tracking

All of the described systems, use video feedback as an output, which is projected on a screen positioned in front of the audience. Thus, all the described cases, require that the audience is oriented toward the projection screen, and focus at least a part of their attention to visually follow the results of the interaction.

Laser pointer tracking obviously does involve tools being distributed to the audience. On the other hand, the traces of the laser beam, due to their brightness, are reported to be relatively simple to track – and authors of [1] managed to scale the hardware used down to one Windows PC and one camera in relation to previous laser-tracking systems (which could involve multiple cameras and workstations). The laser tracking systems have been used in interactive education, audience-level polls and trivias, collaborative painting, as well as games – however, it is reported that laser pointing as a mode of interaction is "less intuitive" and that (in spite that several players equipped with pointers, up to the entire audience, can interact simultaneously with the system) "attempts to play games with … many people generally results in onscreen chaos, since the players cannot tell their laser points apart [1]".

Object shadow tracking again involves usage of tools by the audience to interact, however, here there is only one object that is shared among the audience members. The interaction tool is a beach ball, passed around by the audience members, which intercepts the video feedback projection beam when bounced in the air, and creates a shadow on the projection screen. The position of the shadow in relation to the rest of the projection screen is tracked, so the audience uses the ball's shadow as a cursor. The authors of [1] applied this method in a version of the Missile Command game, and they report that this mode of interaction is "immediately obvious to audience members" and that

> "although only several members control the game action at ay given time, if the activity is chosen carefully, the entire audience gets emotionally involved […] Audience members went to such lengths as throwing other objects around the theater in an attempt to generate additional shadows. [1]"

Audience movement tracking is the method that truly does not require usage of additional interaction tools by the audience members. This approach goes back all the way to a 1973, when the artist Myron Krueger coined the term "Artificial Reality" to describe the concept of "development of unencumbered, full-body participation in computer-created virtual experience [1]". Whereas other systems track motions of small groups of people (up to several individuals – where individual movements can be tracked), and it is reported that "previous systems have required estimating image features such as silhouettes, computing moments and orientation histograms with the assistance of specialized hardware, or placing reflective markers in the scene [1]". The authors of [1] propose a system based on template matching – a version of the popular game Pong – in which a camera frontally records the audience, providing the video input for the system. The audience leans left and right, and by doing so, it controls the movement of the pad in Pong. The system compares the current

frame of the video input with reference images for the audience leaning left, right or standing still – and based on that, the system decides which way the pad should be moved. the choice of leaning action eased the audience interaction a lot - the authors of [1] “found that audiences immediately understood the leaning technique”, most likely since “people have a natural tendency to lean left and right to express a desire to move an object in a particular direction [1]”. The template matching system also allows for detection of other kinds of audience gestures than simply leaning left and right.

Obviously, such an algorithm evaluates the audience scene as a whole – and an individual audience member cannot perceive her/his own interaction in the averaged feedback – or as the authors themselves note, the “technique does not uniquely identify each member of the audience [1]”. Other drawbacks found are the uneven influence of the audience members (dependent on their proximity to the camera), and the need to calibrate the system, both before the show and on the fly – since the number of people in the audience may change during the show and the template images must be updated for the algorithm to work.

Another project related to audience interaction – and probably the largest of its kind – is the Squidball[8] game; and although unknown during the development of our prototype, it shows some surprising similarities. This game was displayed at SIGGRAPH 2004, where some 4000 audience participated in its testing. It tracks inflated balloons of a reflective material 2.5m in diameter, used as motion capture spherical markers, within a 73 x 73 x 12 meters motion capture volume. It uses a state of the art Vicon motion capture system, composed of 22 MCAM2 cameras – and up to 20 markers can be tracked in 3D in realtime. The analog output of the cameras is brought to a dedicated Datastation, which digitizes the video signals and sends them compressed over a gigabit Ethernet network to a Vicon PC, which does the 3D motion tracking. The tracked 3D data was sent through network to another PC that hosted the game engine, programmed in Max/MSP and Java. The game engine provided a 3D virtual world that corresponded to the tracked physical space, where the balloons were represented as spheres. By throwing the balloons around, the audience would cause the corresponding movement of their virtual counterparts in the 3D world of the game. The gameplay thereafter can be described in the words of the Squidball authors “Players moved the weather balloons around the auditorium (whose space corresponded to a 3D space onscreen), in order to destroy changing grids populated by 3D target spheres[8]”.

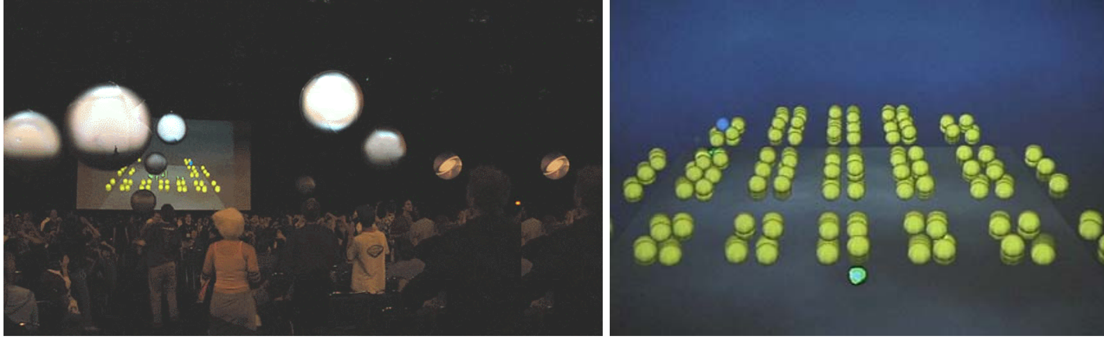


Figure 1. Audience interaction and example screenshot from the Squidball game (from Ref. [8])

Among some issues that are reported from this project, is the logistical problem of calibrating the system in such a large space, as well as the fact that audience in such a big room does not evenly distribute, so the balloons can end up in an empty patch.

The interesting part of similarity with our project is the choice to use a networking environment, which separates the image processing / motion tracking application, which extract the interaction data from a video feed, and the game engine, which applies that data within a virtual world, both as separate software and on separate hardware, making use of an Ethernet TCP/IP local area network to establish communication between the two. Another striking similarity is the virtual world that is rendered as feedback – both the rendering perspective (camera angle) and the fact that the foundation of the world is a 2D surface, which maps to the area the audience occupies in the auditorium playground.

2.1 Summary

All of the described systems use image processing of live video feed to provide audience interaction input. The type of image processing is however different, and it can range from motion capture, through template matching, to extraction of 3D coordinates; from using a single camera to using a system of 22 cameras; sometimes the audience image is captured, sometimes the interaction tools themselves that the audience uses, and sometimes a visual effect that the tools produce elsewhere. As such, live video processing seems the primary choice for providing audience interaction. The hardware used in this context can range from one camera + one PC, to a whole network of cameras and PC's. In all cases the output is a 2D video which is projected on a screen, placed in front of the audience – demanding the audience to both pay attention to the projected feedback, and to think about their interactive

movements in the given context. Thus, all of the audience members need to be faced toward the screen while interacting.

In regards to usage of interaction tools, most of the projects mentioned suggest usage of tools - however, using an audience movement tracking approach does not explicitly require one. However, it pays off to choose interactive actions based on how natural the audience feels they are, in the given context – for instance, the left/right leaning action seems very natural for every member in the audience to perform, given the context of Pong and the task of moving the bat left and right, and the fact that they need to be continually facing towards the projection screen while playing.

As far as application is concerned, audience interaction does not have to necessarily involve interaction with a virtual world – such as in live audience polls and trivias. However, most of the times when audience interaction is used within a context of a game, it becomes a tool for a navigation within a virtual world – whether a 2D one (like in the Pong example) or a 3D one (like in the Squidball example). All such systems require a certain space calibration – establishing the mapping between the space the audience occupies, the image that the cameras capture (possibly an image of the audience), and the output image that displays feedback of the current interaction performed.

The aforementioned projects also establish a distinction between whether the input of the whole audience is averaged, or the input of an individual member of the audience can be distinguished (as opposed to the averaged input). If the input of an individual member can be distinguished, then it becomes an issue whether this input can be uniquely identified (for instance, in the laser pointer project, chaos occurred since the players could not tell their pointers apart, although each laser point was tracked individually – and such input was not averaged). Finally, it is also of importance how many members of a given audience play at a given time – in the Squidball game, although performed before 4000 people, only a few at a time actually hit the balloons.

The authors of [1] have summed up their experience with audience interaction in the following design principles:

- System design
    - Focus on the activity, not the technology
    - You do not need to sense every audience member
    - Make the control mechanism obvious
- Game design
    - Vary the pacing of the activity
    - Ramp up the difficulty of the activity
- Social factors
    - Play to the emotional sensibilities of the crowd
    - Facilitate cooperation between audience members

In all the cases, a lot of emphasis is placed on the fact that the audience interaction should be intuitive and obvious to the audience – the audience should be able to pick up the rules of the game either through a short explanation or through playing of a tutorial-type level (which could also be used for a calibration phase, if needed). Once intuitive understanding is achieved, game narrative considerations come into play, and it is of importance to use the fact that the audience is also a social gathering – so the narrative can be used to facilitate physical cooperation in playing the virtual game. In that sense, audience interaction games would benefit from traditional team games (for instance, the ball shadow technique builds on the type of crowd ball playing known from beach volleyball) – however, the limitation of having to face the projection screen at all times, in order to follow the game feedback, makes direct translation of traditional team games rules a tad difficult.

The intuitive understanding is one of the underlying reasons why we chose to focus so much on the possibility of tracking individual effort within the audience – in spite of the "you do not need to sense every audience member" recommendation above. In essence, audience interaction games (both in the researched projects and in our case) find application mostly in an opening event of a sort. This means that usually there is a reasonably short amount of time allocated for both explaining the interaction and the game engine, and allowing the audience some time to actually experience the game. That becomes even more an issue in the opening ceremony context – a lot of people may have arrived at the event not even knowing they would be expected to participate in a game, so their motivations to get immersed in the game experience might not be the same as when an individual her/himself decides to play a game

on the home PC. A system that has to be explained before it can be demonstrated, might lead to initial frustration of the players, and their disappointment in the system might lead to lack of motivation. The possibility to indicate a personal interaction input by an individual member of the audience, means that the system can be demonstrated as it is explained – meaning not only shorter introduction times, but also the individual game participants will be able to receive a sort of an instant gratification to their actions, as they are being introduced to the game system – possibly motivating them to continue collaborating with one another in playing the game.

3 System description

3.1 The process of audience interaction

The initial demand for the game was that it should support audience interaction, that is a simultaneous input from several persons. After looking into some projects done previously which worked with the same problem[1], several approaches were discussed. One of the features deemed important, was that an individual within the crowd should be able to perceive the effect of her/his own particular interaction, as well as the effect of the other participants in the audience, simultaneously. Such a feature could be implemented in several ways – for instance, through usage of a hand-held tool (like a light or a laser pointer). This was however found to be a problem of logistic nature, and an additional requirement came up that the interaction should be possible to perform without usage of additional hand-held tools.

These limitations influenced the choice of live video processing as a technology to acquire user input – since live video of the audience, representing 2D information, can provide information both about a placement of the individual within the crowd, as well as information about the crowd. Consider the following diagram:

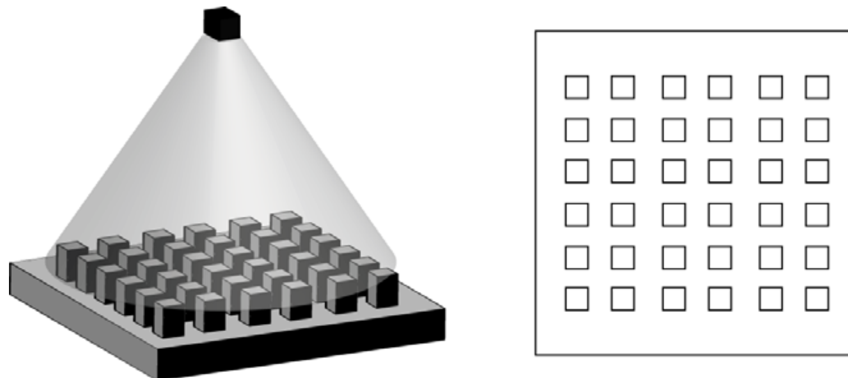


Figure 2. A 3D rendering of live video camera position and audience, and the expected idealized acquired image

The figure shows an idealized setup of a camera, mounted at a ceiling, which is recording the audience from the top. The image we expect to get is basically an orthogonal projection of the audience – neglecting of course distortions that occur in the camera lens. It is obvious from the image that the projection retains information about the placement of each

individual participant within the assembly, as well as information about the assembly in its entirety.

Eventually, the audience is supposed to receive visual feedback from the game, and its interaction with it. The output device that was used to provide this feedback was a LCD video projector, projecting on a screen. The following figure displays the intended setup of position and orientation of the audience in relation to the video feedback projection:

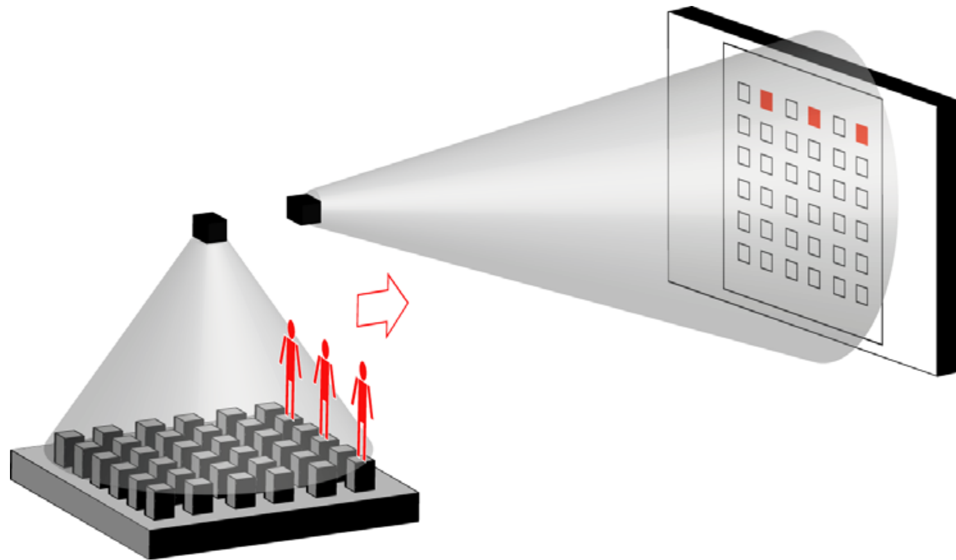


Figure 3. Idealized setup of audience, live camera, and video projection of feedback

Figure 3 displays a setup where each audience member, can identify their position in relation to the audience area boundaries. That is, an audience member positioned in the front right corner of the audience area, would perceive the position of her/his image on the projection screen as at the top right corner. Considering that the image is acquired from live video, motions are rendered as well – and this setup also indicates the mapping, perceived by an audience member, between her/his movements and the response on the projection screen:

- Left/right movement of audience member, on the audience area plane, is preserved as left/right motion on the projection image
- Forward/backward movement, on the audience area plane, is translated to up/down motion on the projection image.

Provided that a single 3D Cartesian coordinate system is used to describe the scene displayed in the image, if the audience area plane is parallel to the xy plane, then the plane of the projection image would correspond to yz plane, for instance. The mentioned mapping

was deemed as most natural mapping between a participant movement, and her/his own perception of the feedback, in the sense of the participant being able to establish the relationship between the actual physical area where she/he is positioned, and the image on the video feedback. In that sense, the participant would be able to navigate in the audience area space, by using the feedback – which is important since in order to play the game, the participants eyes must follow the projected image on the screen most of the time, and thus must retain their orientation towards the screen while moving – which means that they would have less possibility to navigate their movement by looking in actual physical space. This mapping is the basis for the interaction with the virtual world.

The next step was to find a proper mapping between the acquired image and a 3D world. The basic idea that the team worked on was that the acquired video image, representing in essence 2D information each frame, could be understood as a basis for a plane in a 3D world. So, this image can be mapped to a xy plane of the coordinate system of the 3D world, and each pixel's color can be used to generate additional information – which would eventually produce a 3D shape. The 3D world is observed through a camera with an off-axis placement, which can eventually be used to reproduce a perspective rendering of the audience area. The following image illustrates the concept:

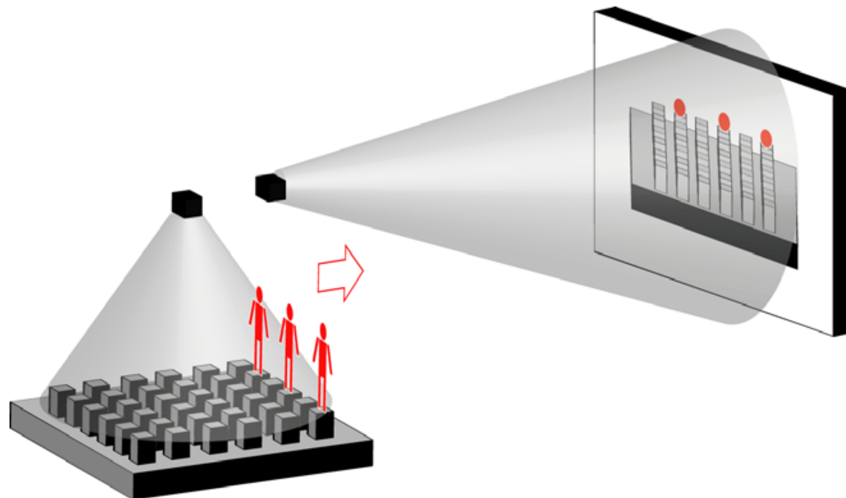


Figure 4. A setup of camera, audience, and projected 3D world, based on the acquired image

Figure 4 demonstrates the relation between the generated 3D world, and the acquired image in Figure 3. The red dots represent the position of each member of the audience, and we can immediately establish a correspondence between the rendered plane in the 3D world and the physical plane of the audience area. For a participant in the audience, this correspondence is also perceivable, although it is obvious that in relation to an audience

participant, he is "viewing himself" from the top – and thus does not have a first person, but instead a third person view over the audience area and himself as well. That however, does not pose a problem in interaction, since the correspondence between movement in the physical area and in the 3D world image still holds. In that sense, the participant would not experience a first-person immersion in the 3D world – but on the other hand, experiences a merge between the physical world (or actions performed in it) and the displayed 3D world, expressed through a 1-to-1 mapping between the audience area and the plane in the 3D world.

3.2 System Design

The previous section introduced the general concept behind the game interaction. At this point, we are still faced with two problems:

- the problem of acquiring input interaction from the live video feed, and
- the problem of mapping and applying that information in the 3D world.

The system design arose by addressing these problems. The problem of acquiring input interaction from a live video feed involves image processing – or, since we need to process images at the frame rate that they are produced, this means live video processing. As this task is conceptually different from the problem of applying information within a 3D context, the team decided to use two different applications to address these two different aspects.

The team chose EyesWeb[2] as platform for video processing, and Virtools[3] as platform for rendering of the 3D virtual world. Both of these applications are intended for the Windows operating system. The biggest problem that arises is of course, how to implement real-time data exchange between the two – as they are by default incompatible, considering their different functionalities and intent of use.

In essence, EyesWeb, just as any image processing application, works with data which essentially represents a 2D matrix, as a representation of an image. The most important link between the two applications, lies in the fact that almost every 3D authoring application supports import of 2D image data in one way or another, mostly intended to be used as a texture.

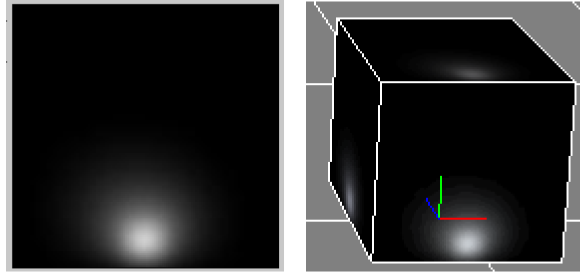


Figure 5. A sample 2D image, applied as a texture to the sides of a cube in Virtools

As it is obvious from the figure, using image data as a texture does not mean the same as using the 2D data to generate 3D data. Regardless of that, the importance is in the possibility of importing the image in the application – once inside, the image data can be extrapolated and be represented as 3D data.

The fact that both EyesWeb and Virtools can be extended through plugins via their respective software development kits (SDKs) in C++, means that they can be provided with TCP/IP functionality and thus operability in a networking environment. Further on, Virtools supports image data access on a pixel level – meaning that the interpretation of the image data as 3D can be programmed directly[1] within the 3D authoring environment. That means EyesWeb and Virtools would be able to talk via TCP/IP, exchanging image data – with Virtools being in charge to eventually map the image data information to the 3D world.

The networking environment would allow both applications to run on the same PC, via the loopback IP address 127.0.0.1 and by simply using distinct TCP ports to communicate. However, since both applications can be quite demanding in terms of usage of PC resources, and given that networking is a possibility, the team chose to use dedicated PCs for each platform – and thus separate the solution of the two problems mentioned above on two distinct machines. Thus each machine would use most available resources for its respective task (image processing and 3D rendering), and the only delay in the processing chain would be in the transfer of the intermediate image between EyesWeb and Virtools via the network.

In addition, the team had access to an Axis 206 IP network camera for the project. This camera has a built in web server, which provides the acquired image in the standard JPEG format – or in case of video, it sends a stream of consecutive JPEG images that represent the frames of the video, which is known as motion JPEG. In any case, due to this functionality, we

[1] If this feature was not present, then extendibility with SDK might allow for the same type of functionality.

can perceive the camera as a small PC of its own, and the output of the camera is supposed to be retrieved via a TCP/IP network – either local or Internet.

Thus all the hardware used in the project could be connected in a local area network:

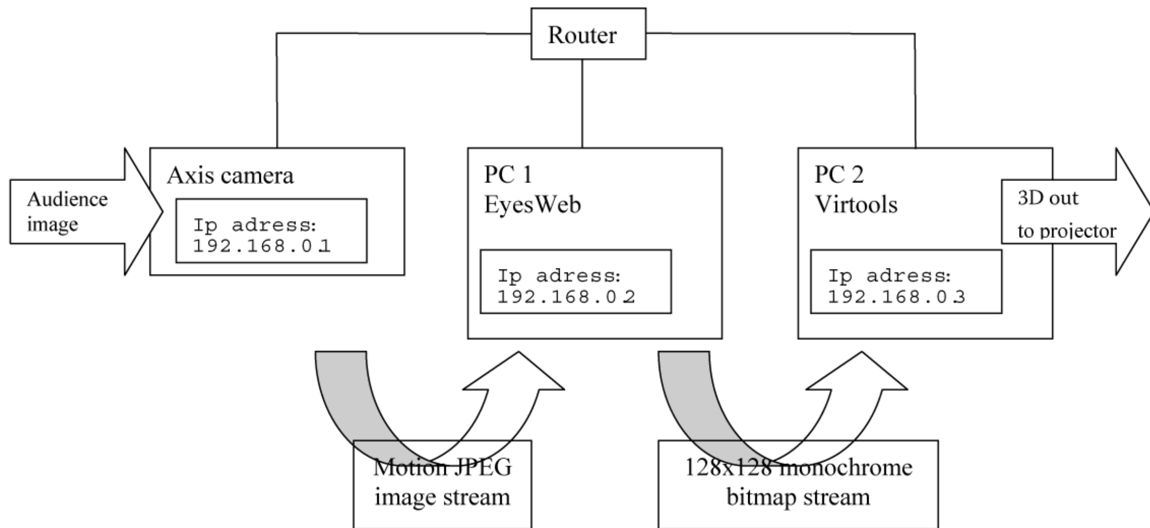


Figure 6. The LAN hardware setup

A more detailed description of the networking setup is described further on in the document. With this setup, the task of each individual machine (and application) is separated:

- Axis camera
  - captures the image
  - converts to JPEG format
  - sends to EyesWeb
- PC1 (EyesWeb)
  - Receives the image
  - Processes the image
  - Sends the result to Virtools
- PC2 (Virtools)
  - Receives the image (and interprets it as a texture)
  - Processes the image data
  - Applies image data to 3D world
  - Renders the 3D world state and sonic illustration
  - Provides the visual output to the projector
  - Provides auditory output to speakers

The formats of the intermediate data exchanged through the network are displayed in Figure 6. The Axis camera sends an m-JPEG stream to EyesWeb, whereas EyesWeb provides a 128x128 monochrome bitmap as an end result of its processing, and sends it as a stream to Virtools.

Having introduced the hardware setup, we may now proceed with the discussion of the image processing algorithm in EyesWeb, and the 3D image interpretation algorithm in Virtools. The 3D interpretation algorithm is the base upon which we build a 3D world based on the 2D image, so it will be discussed first in section 4: 3D interpretation algorithm. The image processing in EyesWeb is then used primarily to extract the information about the motion in the image – which is the acquisition of input interaction by the audience; and to format its results so that Virtools can receive them. This aspect of image processing is discussed in section 5: Image processing algorithm for motion detection – difference based algorithm. The image processing in EyesWeb, however, is not only concerned with motion detection – there are additional aspects that need to be implemented; together with the motion detection algorithm they represent the interaction input image processing chain. The input processing chain, and its eventual effect in the gameplay are discussed in section 6:

Input processing chain and 3D world interaction. Further details about implementation of the game engine will not be discussed in this report.

3.3 Usage of sound

Since the main discussion area of this report is focused on the visual modality, this section will be used to briefly discuss the usage of sound in the game prototype. In essence, sound was used to illustrate the current game context. Both music pieces and sound samples were obtained, which thematically sounded like the early arcade games. The music pieces were connected to start and end of a level; whereas the sound effects illustrated different events in the game, such as obtaining a collectable object. As such, sound did not directly illustrate an interactive action of a participant – unless it resulted with a certain game event. Such implementation of sound would certainly be desirable, as immediate auditory feedback of an interactive action will certainly enhance the visual feedback towards the audience participants, and hopefully increase the satisfaction and interest level a participant might experience, when first encountering the system. As it is obvious, most of the focus of the work was aimed at the visual processing, so it was left to Virtools to manage both the triggering and actual playback of the sound samples – as it was relatively the fastest method available to do at the time – although, in theory, we could have employed a third dedicated machine to render a more interactive sonic illustration of the 3D world.

4 3D interpretation algorithm

The 3D interpretation algorithm, which interprets a grayscale image as a surface in 3D, essentially represents one of the basic tools – surface plotting – being used in mathematics software for display of 2D functions. Before we look at this algorithm, it will be useful to recapitulate on the connection between images, functions and matrices.

4.1 Images, functions and matrices

Wolfram Research gives an excellent introduction to this problem in the online documentation for the "Digital Image Processing" package of the Mathematica application: "An image is a 2D signal, typically a brightness function of two spatial variables. Common examples are photographs, still frames of video, radar and sonar images, and x-rays. A digital image results from a sampling of the spatial domain and a quantization of the brightness values. The sampling produces a finite 2D array of values uniformly distributed over the field of view, while the quantization restricts the sample values to a finite integer range. These necessary operations convert real-world analog sensory data to a form suitable for computer processing and storage [32]". Furthermore, a greyscale or "a monochrome digital image f(x,y) is a 2D array of luminance (brightness) values [33]":

$$f(x,y)=\begin{pmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,0) & \cdots & f(M-1,N-1) \end{pmatrix} \qquad \textit{Eq. 1}$$

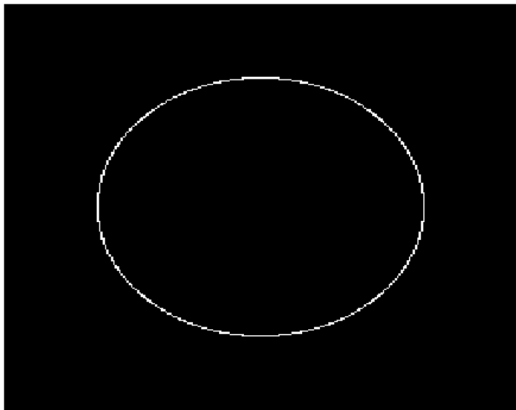
"with $f(x,y)\in Z$, where Z is the domain of the integers, and $0 \leq f(x,y)\in Z \leq$ L-1, where typically L=256. Each element of the array is called a pel (picture element), or more commonly pixel[33]". So, although the original meaning of a matrix is derived from systems of linear equations (see Ref. [34]), it is clear from this equation that "the raw image data of a monochrome image may be conveniently represented by a matrix [33]" (an example is provided in the next section). The image function thus assigns brightness as a real scalar to a set of points (x,y) that represents the domain, or the input set.

In the original meaning of a function as a map from an input to an output set [35], we might see the input set as an empty image – just a set of points with unassigned brightness (or brightness zero), and the output set as a set of points with correspondingly assigned brightness. This is important since it might help us evade some problems. Namely, the Cartesian equation for a circle in 2D is commonly written as

$$x^2 + y^2 = R^2 \qquad \textit{Eq. 2}$$

where R is the radius. Note however, that this is a condition that all points that constitute the circle must satisfy – it is not a description of brightness. To get an image out of it, we simply have to assign some brightness to the points of the circle, such that they can be identified from the background of an empty image. If we assume the brightness of a pixel of an empty image is zero, the image function for a circle could be written as

$$f(x,y) = \begin{cases} 1, \text{ for } \forall (x,y) \text{ such that } x^2 + y^2 = R^2 \\ 0, \text{ for all other } (x,y) \end{cases} \qquad \textit{Eq. 3}$$



which would result in the following image:

Figure 7. Rendering of image of $x^2+y^2=r^2$ (circle)

As opposed to that, simply using $f(x,y) = x^2 + y^2$ as an image function would assign brightness values to the entire domain – that is, the resulting image is:

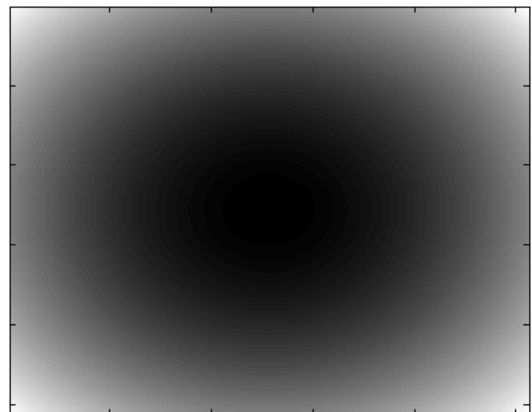


Figure 8. Rendering of image of x^2+y^2 (gradient)

This approach helps us describe an image of a line in a 2D context, which represents a degradation of one dimension (since a line has essentially a one dimensional geometry). The same thinking can be applied for surfaces, which essentially have a 2D geometry, within a 3D image. In a 3D image, of course, we would assign "brightness" to each point - which now is defined through three spatial coordinates (x,y,z), and is often correspondingly called a voxel in literature. Again, the standard way to describe a surface can be seen through a simple example of a plane parallel to the xy plane of the 3D Cartesian coordinate system – for instance:

$$z = 1\frac{1}{2} \qquad \textit{Eq. 4}$$

However, again that is a condition that has to be satisfied by all points that constitute the plane, so to identify them as such in an image function, we could write:

$$f(x,y,z) = \begin{cases} 1, \text{ for } \forall (x,y,z) \text{ such that } z = 1\frac{1}{2} \\ 0, \text{ for all other } (x,y,z) \end{cases} \qquad \textit{Eq. 5}$$

which would result in the following image:

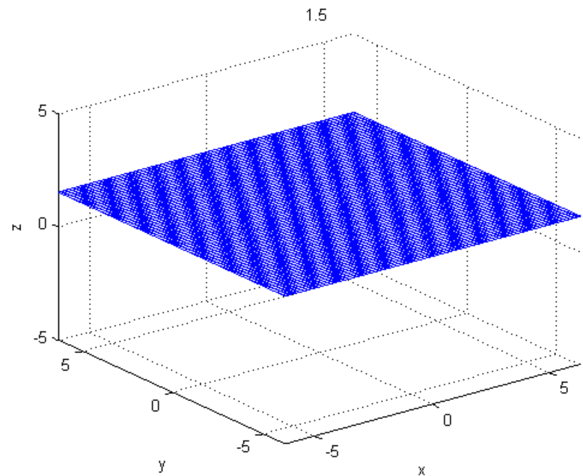


Figure 9. Graph of the plane z=1.5 in 3D Cartesian coordinate system

4.2 Surface plot of an image as 3D interpretation

Let's look at a real scalar 2D function, that can be written as

$$g = f(x, y) \qquad \text{Eq. 6}$$

which means that every distinct pair of (x,y) from the input set, relates to a single scalar value g of the output set. The coordinates x and y represent of course Cartesian coordinates, and thus define a 2D plane.

In the discrete case, provided the input and output set are finite and limited, the 2D function can be represented with a 2D matrix:

$$G = [g_{xy}] = f[x, y] \qquad \text{Eq 7}$$

so we have a simultaneous representation of the function output, as well as input set – since the position of a given function value g_{xy} in the matrix is determined by its input coordinates.

As an example, we can use a short example for the Matlab application. For instance, we can use Matlab to generate an 8 by 8 matrix, which would represent this function:

$$g = f(x, y) = 8 - (|x - 4| + |y - 4|) \qquad \text{Eq 8}$$

We would use the following code

```
for x = 1:8
    for y = 1:8
        G(x, y) = 8 -
(abs(x-4) + abs(y-4));
    end
end
```

which results with the matrix G:

```
2   3   4   5   4   3   2   1
3   4   5   6   5   4   3   2
4   5   6   7   6   5   4   3
5   6   7   8   7   6   5   4
4   5   6   7   6   5   4   3
3   4   5   6   5   4   3   2
2   3   4   5   4   3   2   1
1   2   3   4   3   2   1   0
```

As soon as we have the matrix G, since we have strictly scalar and positive values, we can map them to gray color pixel intensity. Thus, instead of writing numbers to represent the matrix values, we can display the matrix as an image, which in Matlab is easily achieved by running the commands:

```
>> figure;
>> colormap(grey)
>> imagesc(A)
```

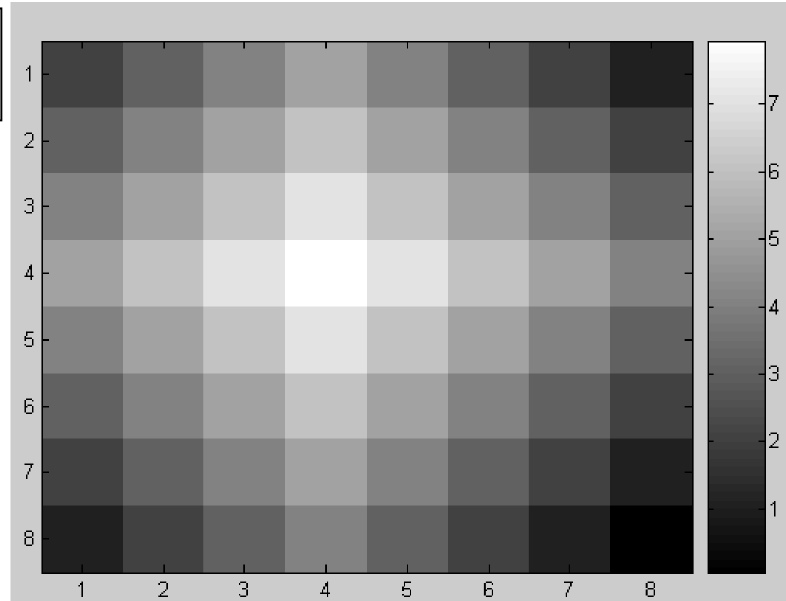


Figure 10. Matrix display as a grayscale 2D image

So we can see that the function values now assume the roles of pixels in the image, where the coloration of the pixel corresponds to the function value, while the geometry of the matrix is preserved.

However, in Matlab it is just as easy to obtain yet another view of this function. Namely, the xy input set can be seen as the xy plane in a Cartesian 3D coordinate system. The values of the function g can then be mapped to pixels in a 3D environment – their x and y coordinates correspond to the function input set, whereas the function value is taken to mapped to the height of the pixel placement within the 3D system. In this way, the pixels representing the function value will form a 2D surface in the 3D system – where the height of the surface at a given (x,y) corresponds to the function value g=f(x,y). To view such a representation in Matlab, we would simply call

```
>> figure;
>> colormap(grey)
>> surf(A)
```

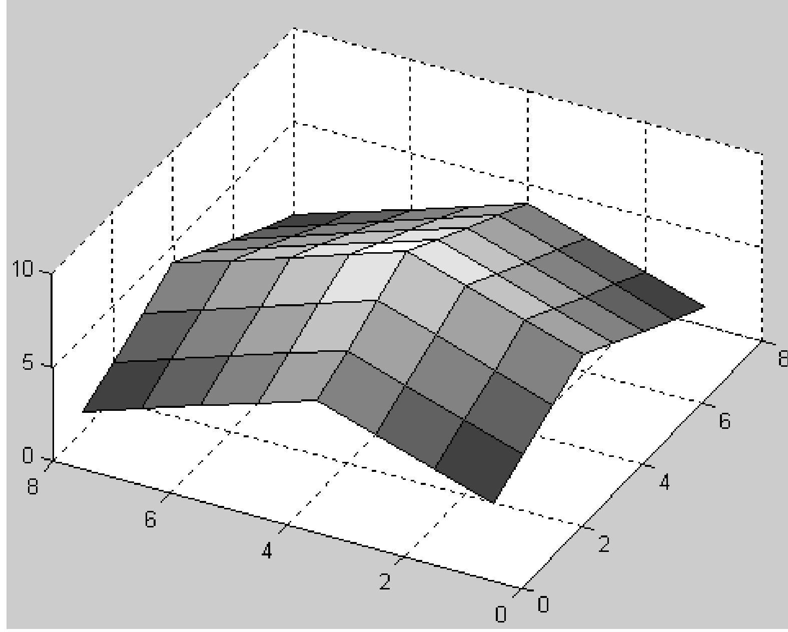


Figure 11. 3D surface representation of 2D data in Matlab

As discussed in the previous section, if the 2D image function is given by $g = f(x, y)$, say the used, then the corresponding 3D function $f_{3D}(x, y, z)$ that describes the surface with a height corresponding to the image might be written as

$$f_{3D}(x, y, z) = \begin{cases} 1, \text{for } \forall (x, y, z) \text{such that } z = g = f(x, y) \\ \quad\quad 0, \text{for all other } (x, y, z) \end{cases} \quad \text{Eq. 9}$$

or, since we use the function values also for the brightness of the surface in Figure 11, the corresponding description would be

$$f_{3D}(x, y, z) = \begin{cases} \text{g}, \text{for } \forall (x, y, z) \text{such that } z = g = f(x, y) \\ \quad\quad 0, \text{for all other } (x, y, z) \end{cases} \quad \text{Eq. 10}$$

In other words, the surface is a 3D model of all (x,y,z) points that satisfy the equation $z = f(x, y)$.

In essence, it is this very algorithm that was applied in the game (more details about the Matlab algorithm can be found in the documentation for surf[36] as well as the section for 3-D visualization[38]; see also [37] for more on function visualization). The 3D world that we generate based on an input image, is essentially a 3D surface representation of the image that is received. In that sense, the 3D world that is generated is basically a surface, whose elevation at a given (x,y) position is based upon the image received. As such, what the audience has control over, is the elevation of the surface at a given point. The values mapping in the game

is exactly the same as in this example – brighter pixels are mapped to greater height in the surface.

This is the reason why a monochromatic bitmap was used as the intermediate data between EyesWeb and Virtools – in order to achieve this kind of a surface 3D representation of an image, we simply do not need any color information, other than grayscale. This also helps save some bandwidth (the image size transferred would be 3 times greater if an RGB color bitmap was used).

More precise description of the use of this algorithm within the context of the game will be given further in this document; however, the details of its implementation in Virtools will not be discussed in this report.

5 Image processing algorithm for motion detection – difference based algorithm

As discussed in the previous section, all we need is a monochromatic image transferred to the 3D application, in order to create a 3D surface that represents our 3D world. Then, if our monochromatic image contains information that represents where in the auditorium has movement occurred, we can generate a 3D surface based on such video signal, which can be interactively shaped by motion in the scene. That part of the input image processing chain will be discussed here. In order to keep the bandwidth low, the decided size of the output bitmap was 128x128 pixels – which for a grayscale image (1 byte per pixel) amounts to 16KB per image. So, basically, this set the input and the output parameters of the image processing part of the chain, which was conducted by EyesWeb:

- Input – 320x240 RGB bitmap
- Output – 128x128 monochrome bitmap

The basic task for EyesWeb was, therefore, to extract interaction data from the input image, and encode it in the output image, before sending it to Virtools. The choice for the basic algorithm, used to extract interaction data from the acquired video image, fell on the method known as difference-based analysis. This algorithm was used to extract information about movement that participants in the audience performed. In the following text, we shall introduce this algorithm, and its relation to the motion of the audience and the effect on the generated 3D world.

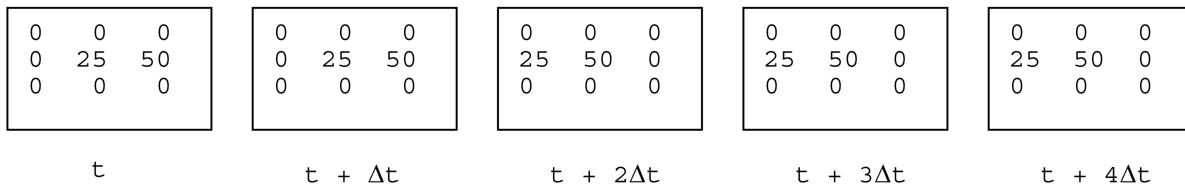
In essence, a difference-based algorithm is simply the difference of two consecutive image frames. In its simplest form, and the one used in this project as well, the images are grayscale, which allows us to treat the image as a 2D matrix of values between 0 and 127. As a mathematical equation, we may write the original animated image sequence as a scalar function of two spatial coordinates and time $g(t) = f(x, y, t)$. In that sense, the animated image we obtain through difference based analysis is the partial derivative of g(t) over time:

$$g_o(t) = \frac{\partial g(t)}{\partial t} = \frac{\partial f(x, y, t)}{\partial t} \qquad \textit{Eq 11}$$

In discrete terms, $g_o(t)$ and $g(t)$ are represented by arrays or streams of 2D matrices, which represent image frames of an animation. Thus, the matrix G_o which represents a frame at a given moment t of the output animation, obtained through difference-based analysis of the input image stream $G[t]$:

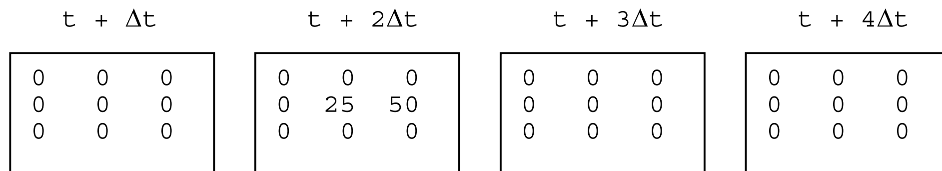
$$G_o[t] = G[t] - G[t - \Delta t] \qquad \text{Eq 12}$$

We can use Matlab to perform difference based analysis of the following matrix sequence:



t	t + Δt	t + 2Δt	t + 3Δt	t + 4Δt
0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
0 25 50	0 25 50	25 50 0	25 50 0	25 50 0
0 0 0	0 0 0	0 0 0	0 0 0	0 0 0

Which will result in the output sequence



t + Δt	t + 2Δt	t + 3Δt	t + 4Δt
0 0 0	0 0 0	0 0 0	0 0 0
0 0 0	0 25 50	0 0 0	0 0 0
0 0 0	0 0 0	0 0 0	0 0 0

since we do not allow negative values for the image matrix – so actually we are using

$$G_o[t] = \left| G[t] - G[t - \Delta t] \right| \qquad \text{Eq 13}$$

Both the input and the output sequence, plotted as grayscale images will look like

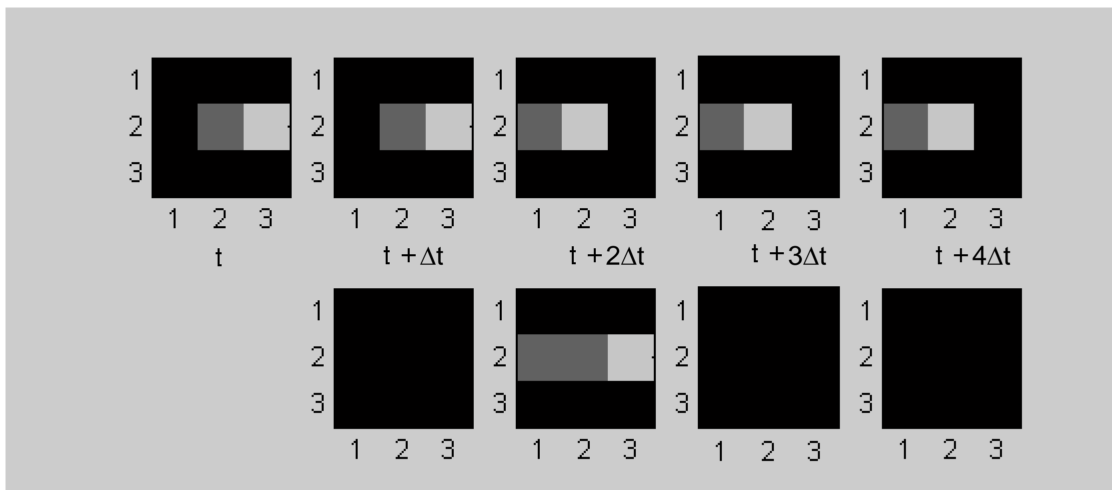


Figure 12. Image representation of original and difference-based processed matrices

The output sequence serves as an indication whether a change in the spatial distribution of pixel values has occurred from one frame to another – and thus serves as indication whether motion has occurred. The motion is also localized – even though the matrix size is too small to allow for a detailed overview, still we can tell that the motion occurred in the middle row of the image, just by looking at the output sequence. The aspect of the localization of motion can be more clearly seen on this animation sequence of 12 frames, where the input sequence is displayed above, and the difference-based sequence below the frame number:

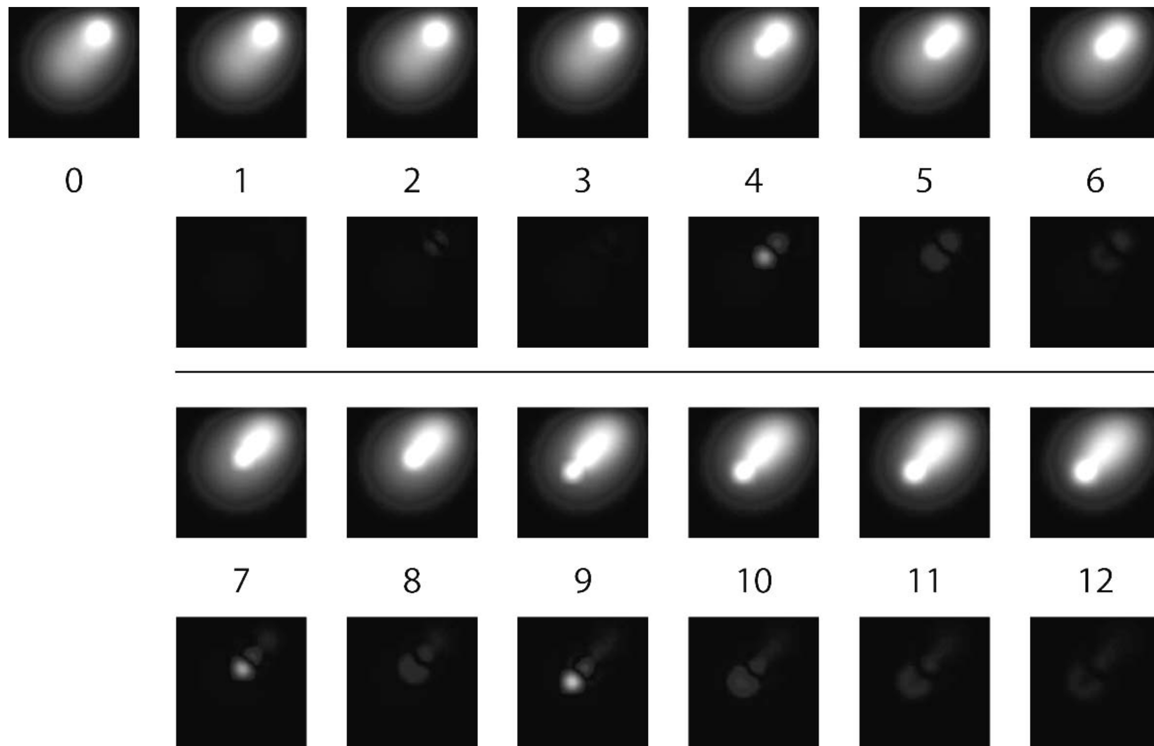


Figure 13. 12 frames of input and difference-based output animation

We can observe that the amount of detected motion (non-black pixels) in the processed sequence does not depend on the actual distribution of pixels in a given input frame – only on the amount of different pixels from frame to frame. The input sequence represents a moving circular area of bright pixels, leaving a blurred comet-like trail. For slight displacements of the original bright spot, which relate to slower motion, the corresponding difference-based frame shows mostly a dark image. For bigger displacements from frame to frame, which relate to faster motion, there are more bright pixels in the difference-based output – and as it can be seen (frames 7 and 9), the position of the bright spot can be approximately localized as well.

This algorithm can be applied regardless of the actual contents of the input sequence, so we may readily replace with any video stream – provided we have discarded the color information first, so we have a grayscale version of it that we use for the frame subtraction. Obviously, there is always a delay of at least one frame, but for an ideal frame rate of 30 fps, a delay of 33 ms should not make a big difference in the user experience, which was also observed during prototype demonstration - none of the participants seemed to complain about delay, even with worse actual frame rates than 30 fps.

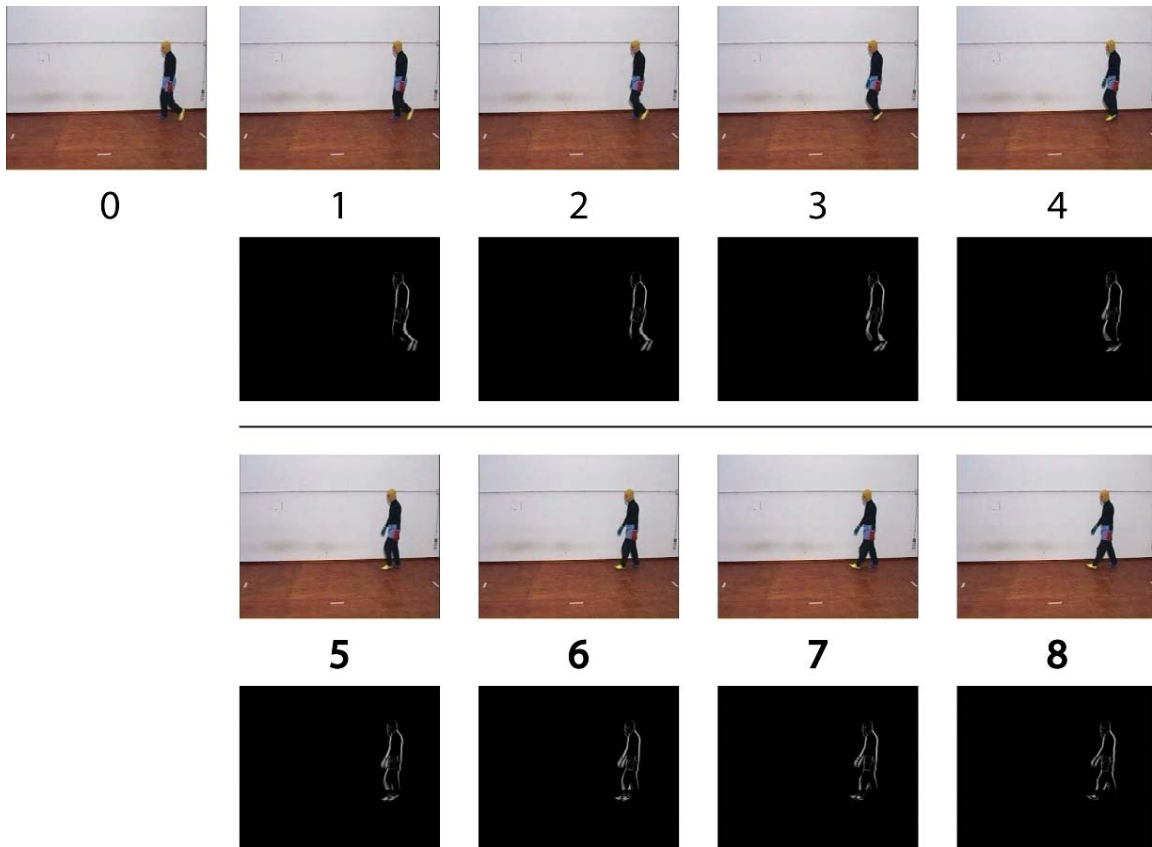


Figure 14. Difference-based analysis of an input video stream (EyesWeb example)

The difference-based processed output of the input video signal was then eventually downscaled to 128x128 pixels, providing a format and resolution similar to the image sequence presented on Figure 13.

So, the difference-based analysis is a simple means of tracking and visually localising movement in a 2D scene. As long as a member of the audience performs any movement within the captured scene, such that it causes a change of the distribution of pixel values in the captured video stream, it will be captured by the difference-based algorithm, and indicated by bright pixels in the area of the scene where the movement happened. This allows

for treatment of the difference-based algorithm as an input interaction algorithm, where the interaction tracked is simply any movement that an audience member performs, that can be registered on the captured video image of the scene. The implementation in EyesWeb is shown in Appendix B.

Simple as it is, the difference algorithm also has its drawbacks when it comes to its use as input interaction algorithm, in the sense of application in this project. Those can be summarized as follows:

- **The intensity of output pixels depends on the difference in brightness between the moving object and the background**. While developing, the best results were obtained when there was not too much light in the scene, and a small flashlight was used as the moving object in the scene. A scene which is considerably well lit, featuring a moving person, dressed in colours not too different from the background – or on the other hand, a scene which is rather poorly lit altogether – would result in quite less bright pixels in the processed output than the ones demonstrated in Figure 14.
- **The distribution of output pixels depends on the area that the tracked object occupies in the input scene – and thus on the camera placement in relation to the tracked object**. The frontal camera capture displayed in Figure 14, and the proposed top camera placement for the project in Figure 2, will give quite difference distribution and localisation of pixels that represent a person in the input video scene. Thus, a movement that a person performs will provide different change of distribution of pixels in the input scene, depending on the camera placement – and this will lead to different distribution of bright pixels in the difference-based output.

These issues were somewhat addressed by providing the audience members with white gloves as interaction tools. As all the movements that the members perform, that are detectable on the captured video scene, participate in the generation of bright pixels in the output – even involuntary movements influence the output. By using the gloves, the team attempted to create an area of high contrast from the background for each member of the audience, which is also more localised than the image of the entire body of an audience member in the scene. In that way, the influence of the rest of the body in the difference-based output scene (as more distributed pixels, but less bright) should be negligible in relation to the influence of the registered movement of the gloves (as more localised and more bright

pixels). Although this approach was helpful, it raised other problems when members of the audience happened to wear other clothing that also represented high contrast with the background.

Even without these considerations, the difference-based algorithm is quite sensitive to local changes of coloration in the scene. Local changes of natural light are detected, as well as change of artificial lighting in the scene; difference of pixel values in consecutive frames due to JPEG compression are also detected, and effectuated as noise throughout the output image.

The choice of a dedicated image-processing platform for tracking the camera input interaction allows for some intervention in the difference-based algorithm so these issues can be addressed. For instance, a more advanced image-tracking algorithm can be used, as well as gloves of some distinct colour which is not found in the background – so that the tracked movement from a given audience member can always be localised with the same amount regardless of their position in relation to the camera. The team decided however, to use the simple version of the difference based algorithm described here, partly to keep the prototype development simple, and partly because other issues needed to be addressed as well – discussed in the next section.

6 Input processing chain and 3D world interaction

Having said the above, we can now revisit the process of audience interaction and provide a more detailed description of the actual setup used in the game demonstration prototype.

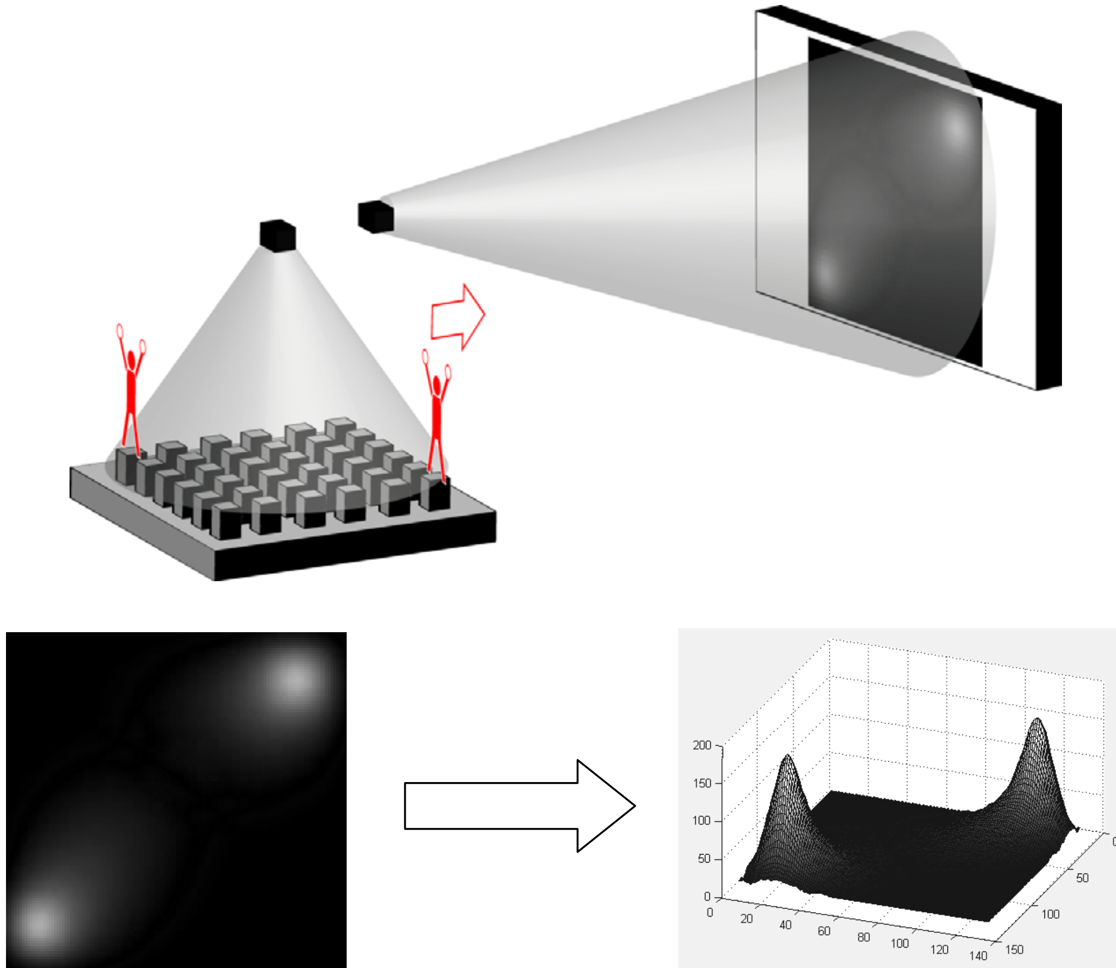


Figure 15. Scene setup, difference-based output and 3D Matlab surface plot based on it

Each player (audience member) assumes a distinct position within the captured playground area. Equipped with gloves, each player can cause change in the captured scene, and thus cause a bright pixel output in the difference-based processed video signal. If the players happen to be standing still, such change can be induced by simply waving their hands in the air – which would create a localised change of pixel values in the approximate placement of the player within the playground. Under idealized conditions (under the assumption that the white gloves represent the most contrast with the background on the captured image, and that the captured image is an ideal top projection of the playground

scene), such a situation is presented in Figure 15, representing a playground scene with two players, and the expected difference-based output. Of course, if the players are moving throughout the playground, the waving movement is not necessary to cause output – although it was observed during prototype demonstration that most players end up continuing with their waving actions when moving as well. Thus, the displacement of the players hands within the playground scene (either by waving while standing still, or by translation through the playground), emphasized through high contrast gloves, is the chief method of causing local output in the processed image, in an area that indicates the position of the player within the playground – and as such is the chief method of interaction of a player with the rendered 3D world.

The top part of Figure 15 represents the mapping between the situation on a physical playground scene, and an idealized difference-based processed image obtained from the camera, and thus represents the task performed by EyesWeb. The bottom part represents the mapping between the processed output image, and its interpretation as a 3D surface, and thus represents the task performed by Virtools. It is actually the 3D world, composed of such a 3D surface, which is presented back to the players. However, difference-based processing in not the only image processing algorithm performed by EyesWeb. For a detailed screenshot of the EyesWeb patch, please see Appendix K: Screenshot of EyesWeb patch with plugins. Two other major input image-processing issues, in addition to the difference-based algorithm that performed motion detection, were providing a persistence effect, and correction of spherical distortions. Those two aspects, described further on in this section, helped in achieving the idealised situation represented on Figure 15 more closely. Finally, the entire input processing done by the EyesWeb patch, along with the 3D interpretation, allowed for the game interaction implemented for the prototype demonstration, which is described in section 7: Prototype gameplay.

6.1 Persistence effect

Since the difference based algorithm is quite quick – it works frame by frame – as soon as a person becomes still, the visual indication of position is gone as well. That is why steps had to be taken to ensure longer persistence of a users action. EyesWeb was again used to implement this functionality, and it is here where the input processing chain started growing beyond the mere difference-based processing. By introducing a certain amount of blur, as well as a slight amount of positive feedback in the processed image stream, an effect of a

wave-like persistence of the users action was achieved. Basically, whenever the user made a motion that induced appearance of bright pixels in the processed stream, and then stopped the motion - the blur/feedback mechanism ensured that this change was continuous, so the pixels fade from their initial brightness into total black in about 0.5 seconds.

This persistence effect, of course translated into the 3D surface interpretation as well, and displayed itself as an effect of a wave-like rise and fall of hills from the surface. So, the interaction of the players with the 3D world, composed of a 3D surface, can be finally summed up as the possibility to induce rise and fall of local “hills” (or rather, waves) at the surface – whose location is correspondent to the location of the player within the playground limits – by executing motion, either translatory or a hand-waving one (if standing still). The only condition is that this motion is detectable as motion of the gloves (as the highest contrast local points) in the scene captured by the camera. The players also have the possibility to move the wave (since one will be rendered whenever the player makes a translatory movement anyways). So, the creation and movement of wave-like hills on a surface in a 3D world, is the scope of the virtual world interaction through image processing researched in this project. Gameplay was further developed by conceptualising interactions between the 3D surface and independent virtual objects within the virtual world.

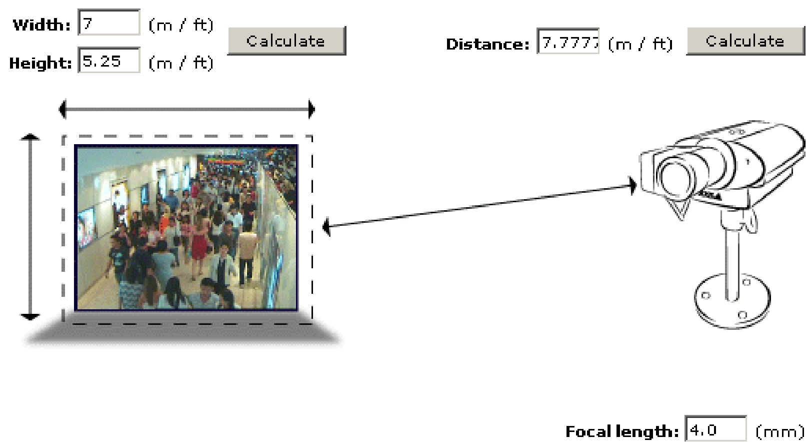
The implementation of the persistence algorithm in EyesWeb is noted in Appendix J. However, the persistence effect was not the only extension to the difference-based algorithm within the input processing chain. Due to the physical restrictions of the available space for the demonstration prototype, the team needed to implement some changes to the basic scene setup that was noted up to this point in the report.

6.2 Catadioptric extension and radial distortion algorithm

The system proposed on Figure 2 assumes that a camera is placed far enough from the floor, so that the entire allocated playground area can be captured from a top perspective. How far it needs to be placed depends on several factors – the aperture of the lens of the camera, zoom, focal distance etc. The game was initially envisioned to support up to 50 audience members as active players, and by allowing each member some 1 m^2 elbow space, the scene that the camera should capture, would measure some 50 m^2 on the floor (say, a 7m x 7m playground). As it was mentioned, orthogonal top view video was to be used as source for a difference based algorithm, which would provide indication of local

motion within the image. A 320 x 240 = 76800 pixels image, and top capture of 50 people, yields approximately 1536 pixels per person, or an appoximate square patch of 40X40 pixels – which the team deemed enough of an area so a waving hand movement would be identified from each member of the audience – for instance, if they were seated in rows and columns equally distributed across the playground area. The limitations of the available space and the available camera soon put this expectation a bit lower, and required a change in the scene setup.

The Axis camera (Axis 206) that was used for the project has a focal length of 4 mm for the lens. The Axis website features an online calculator which provides the following results:



Example: If the distance to the scene is 8 meters, the calculated width and height of the scene will be 7.20 and 5.40 meters, respectively.

Figure 16.Online lens calculator for the Axis 206 camera (From Ref. [43])

That means, in order to capture a 7m x 5m scene on the floor, this camera needs to be placed 7.7 meters away, however – the available height in the space allocated for the prototype demonstration was only around 3 meters.

This issue was resolved by applying a technique used in what is known as one shot or panoramic imaging[5] (also 360° imaging) – which is to use a convex mirror, whether spherical or conic, to obtain a 360° of the entire space, and then use a camera to capture the mirror image. A lot of times, an algorithm such as rectangular to polar conversion is used to "unfold" the image (see [5] for the effect). The usage of spherical mirrors is also known as catadioptric image formation (see [42] for more on catadioptric sensors) – "We refer to the approach of using mirrors in combination with conventional imaging systems as catadioptric image formation. Dioptrics is the science of refracting elements (lenses) whereas catoptrics is

the science of reflecting surfaces (mirrors) [Hecht and Zajac, 1974]. The combination of refracting and reflecting elements is therefore referred to as catadioptrics [6]” In that context, the basic scene setup presented in Figure 2 was extended through an addition of an industrial spherical mirror, 45 cm in diameter as a catadioptric tool:

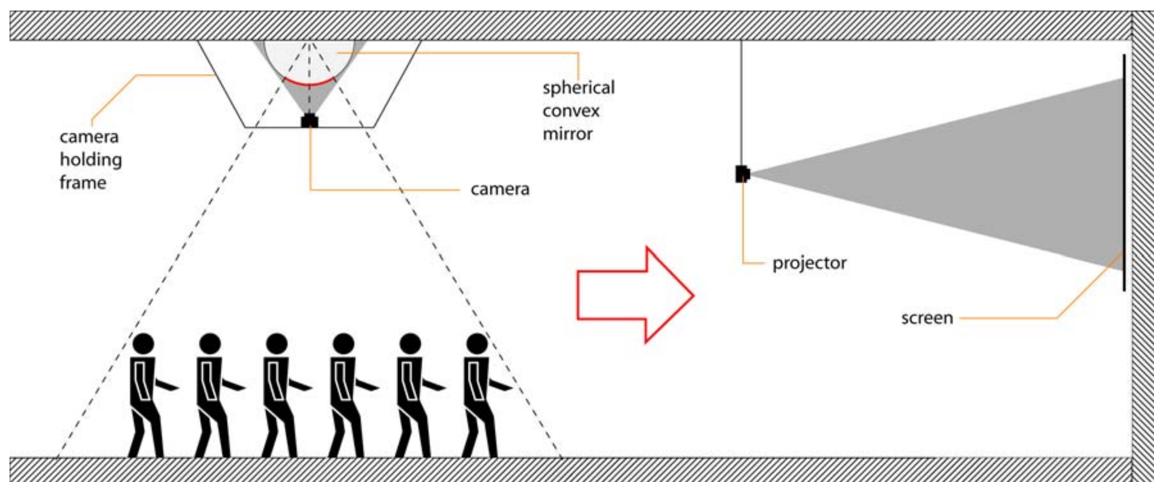


Figure 17. A side view of the demonstration space, with catadioptric enhancement to the basic camera setup

As the figure demonstrates, the limitation of low ceiling height was overcome by mounting a convex spherical mirror on the ceiling. A metal frame was mounted on the ceiling as well, so that a camera could be placed on it, focused on the center of the mirror, which would capture the mirror image. The red highlight of the mirror surface shown on the figure demonstrates the extent of the playground scene on the mirror surface, and indicates that only a fraction of the mirrors surface holds the playground image scene. The rest of the image on the mirror encompasses most of the room space, including the vertical walls – consider an actual camera image capture from the convex spherical mirror:

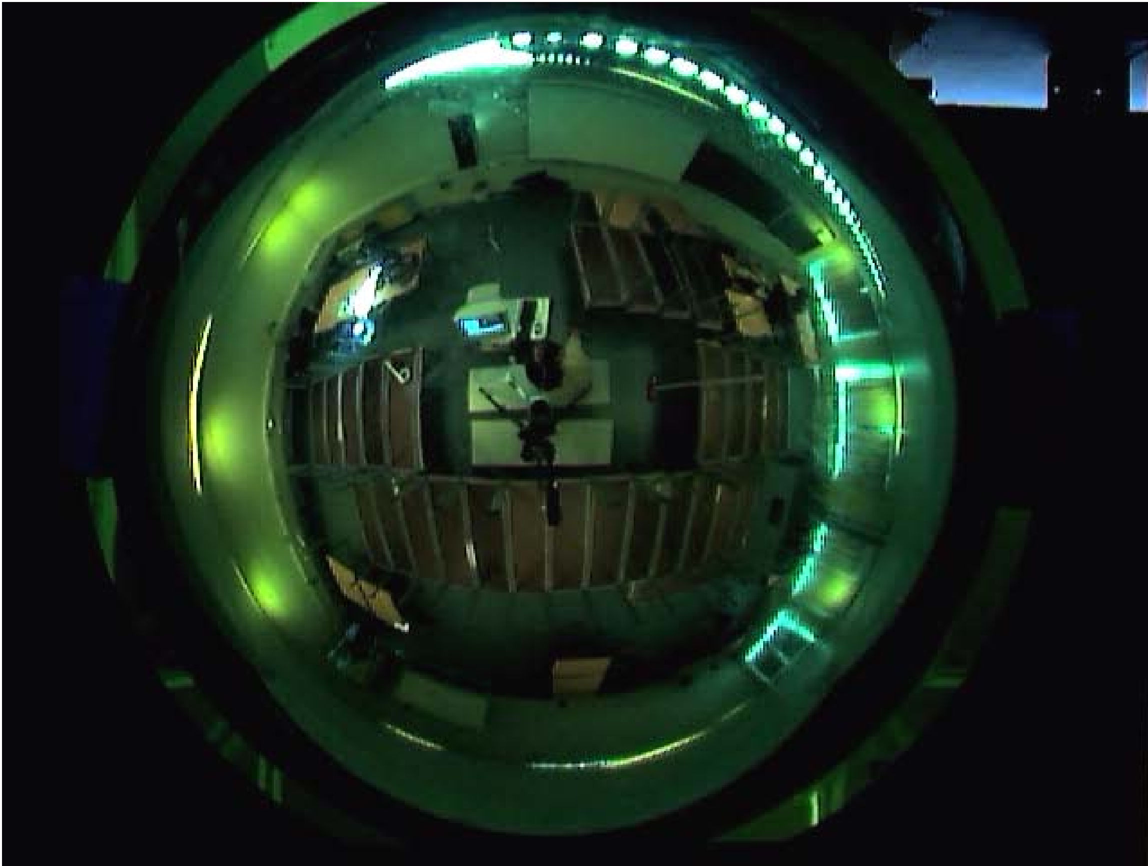


Figure 18. Example camera image from a spherical mirror mounted on the ceiling

Obviously, an image from a spherical mirror does not even come close to the idealized orthogonal projection as suggested in Figure 2. The acquired image suffers from spherical distortion – what was originally a straight line on the floor, becomes a curved one on the spherical mirror image. Furthermore, an ideal orthogonal projection exists only for the line directly below the center of the mirror (where the camera is placed) – for all objects positioned elsewhere some additional perspective will be included in the image as well. For instance, Figure 18 shows an image of desks placed upside down, with their legs upwards. An ideal orthogonal projection would make the length of the legs "collapse" in the image as being oriented only upwards (on the 3D dimension) – in the spherical image, we can observe that not only the length of the legs is displayed in the image, but they are also curved, and the amount of the projected length of the legs on the image depends on the overall position of the desk on the floor.

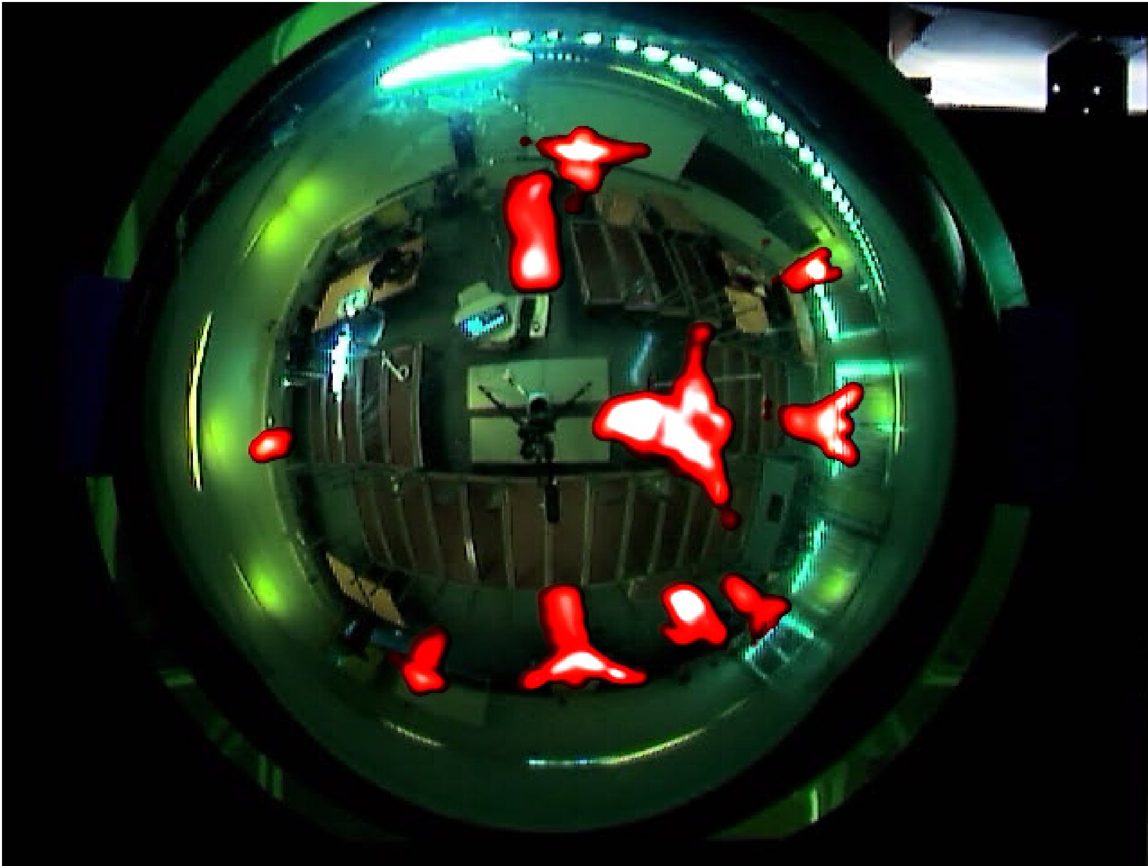


Figure 19. The red areas represent the ammount of pixels a person occupies in the spherical mirror image, depending on the persons position on the floor

This is a problem in relation to the interaction in the game, as we are aware that any change of pixels within the image contributes to the difference based output, and thus the image of the entire body of an audience member influences the algorithm. And as we can see from the image above, how many pixels the image of the participants body will occupy on the spherical mirror image, depends on the position of the member in relation to the sphere center. If positioned directly below the mirror, the image of a participant collapses simply to an orthogonal image of the participants head – for any other position, other parts of the body will be visible on the image as well – and so the area that a participant occupies on the image will increase. Obviously, this will influence the amount of bright pixels obtained through difference-based motion analysis.

Taking into account that the gloves should occupy lesser area than the rest of the participant's body in the spherical image, and that due to background contrast, they will cause the brightest pixels in the difference-based output, we can work with the assumption that this spherical perspective problem can be ignored. However, straight lines on the floor are still

mapped to curved ones on the spherical image, and it is here where an additional image processing algorithm was applied in the input processing chain.

The applied image-processing algorithm is known from applications like Adobe Photoshop, where it resides under the name of Pinch or Spherize. This is a known problem from optics in lens, where it is simply known as distortion, one of the Seidel aberrations (see [41]); radial distortion[40]; or, "since straight lines in the object space are rendered as curved lines on the film[39]" also as curvilinear distortion – whereas the terms *barrel distortion* and *pincushion distortion* are used instead of Pinch and Spheric. Based on the description and the Matlab code for this algorithm that is described in [7], the SDK extensibility of EyesWeb was used and a block was programmed for EyesWeb, which provided this functionality – it is described in more detail in Appendix I. The compiled client plugin took the form of an EyesWeb block, which can be seen on the screenshot in Appendix K as the block called "EywPinchSpheric".

The effect of the algorithm on a grid can be seen on the following figure:

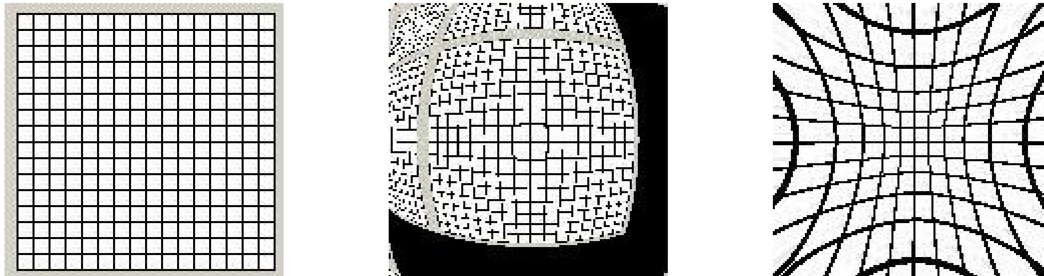


Figure 20. Pinch/Spheric effect with a distortion parameter gamma: a) gamma = 0 b) gamma negative – barrel distortion (Spheric) c) gamma positive – pincushion distortion (Pinch)

As we can see from the figure, the algorithm produces a spherical (or rather, radial) distortion, quite like the one from the mirror – for a negative distortion parameter, where the inverse is achieved for a positive distortion parameter. In that sense, the algorithm was used in attempt to correct curved lines due to the spherical distortion effect of the mirror. Note, however, that the authors of the algorithm in [7], which we used as a base, used it to actually distort an input image – in their document, they describe other means of reconstructing the original input image from the radially distorted one (polynomials and neural networks). We assumed that by applying the inverse distortion of the observed spherical effect from the convex mirror, we would restore the distorted curved lines into straight ones – only to that degree, in which the audience members would have the perception that their linear

translatory motions across the playground is accurately reflected in the 3D world waves. As the next figure shows, the reconstruction of the curved lines back into straight is far from ideal – however, there were no complaints from the audience; straight motion in the playground was perceived as accurate enough – as the blurring, persistence and general low resolution representation of the tracked input video may have masked eventual distortion residues.

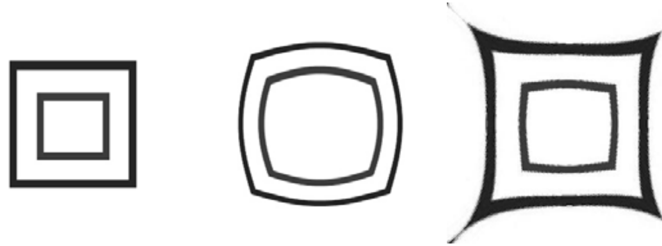


Figure 21. a) Original straight line pattern,b) the pattern seen on a spherical mirror, c) reconstructed spherical image

Basically, a straight path on the floor (like the pattern on Figure 21 a) captured by the spherical mirror system, would be represented as a curved path (like the pattern on Figure 21 b). In that sense, an audience member walking in a straight line through the playground, would experience a “wave” moving through the 3D surface in a curved line. By using the Pinch/Spheric effect, the distorted image can be reconstructed, although not ideally - Figure 21 c shows that the outer rim is “straightened” more than the inner one. Still, even without that precision, it was possible to tune this filter for the demonstration, in such a way that translatory movements in a straight line within the playground were experienced as straight movements in the 3D world.

Finally, there is one more problem with the proposed catadioptric extension – due to the camera placement, player movements can not be tracked right below the mirror center – as we can see from Figure 18 or Figure 19, the center of the image is basically the reflection of the camera lens, and as such, the camera blocks imaging of the floor situation right below. Eventualy, that means that a player can not cause a “wave” right in the middle of the 3D surface – but that can be compensated for in the gameplay programming (for instance, the center can be programmed to be a level transition space, so to speak – so once a given object reaches it, the level has been passed and a new level is displayed to the player).

7 Prototype gameplay

In review, the input processing chain performed by EyesWeb, with all the enhancements described above, can be conceptually modelled as on the following figure (see Appendix K for the implementation):

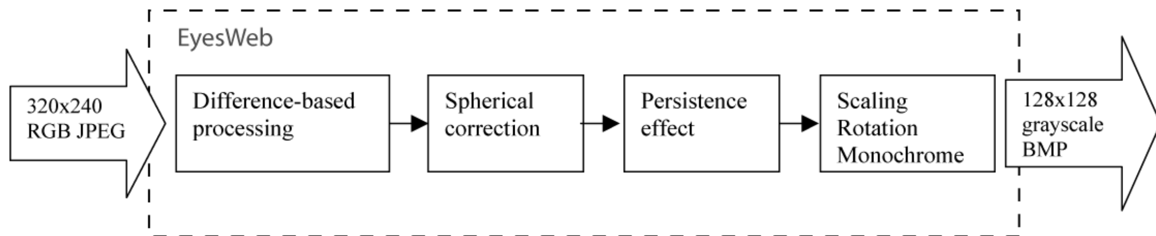


Figure 22. The input processing chain performed by EyesWeb

The definition of the interaction mentioned earlier, involves creating bright spots in the processed image, obtained through the input processing chain, which would translate into elevations of the surface that composes the virtual 3D world. That is demonstrated in Figure 15, which shows the following mapping between a 2D image and a 3D surface performed in Matlab

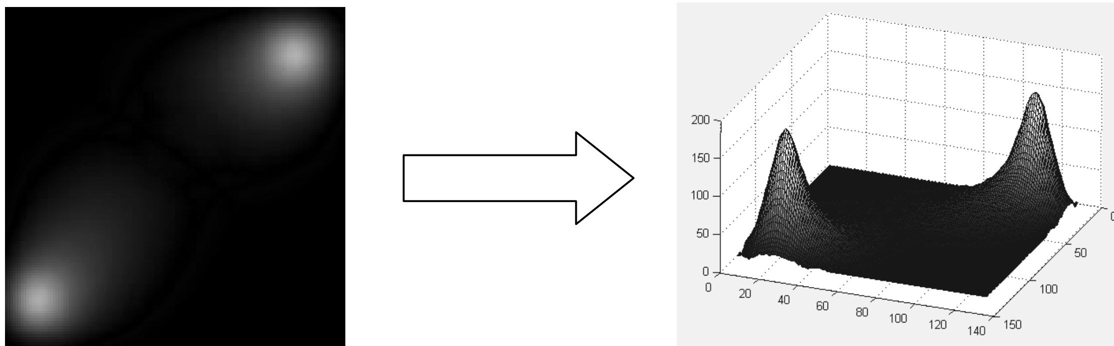


Figure 23. 2D - 3D mapping of grayscale image in Matlab

As mentioned before, Virtools was used as the end 3D rendering application. Similarly to the Matlab example, the Virtools program also rendered a surface. This surface, however was composed of a certain number of tiles, which could be elevated depending on the pixel brightness at the corresponding location in the 2D image:

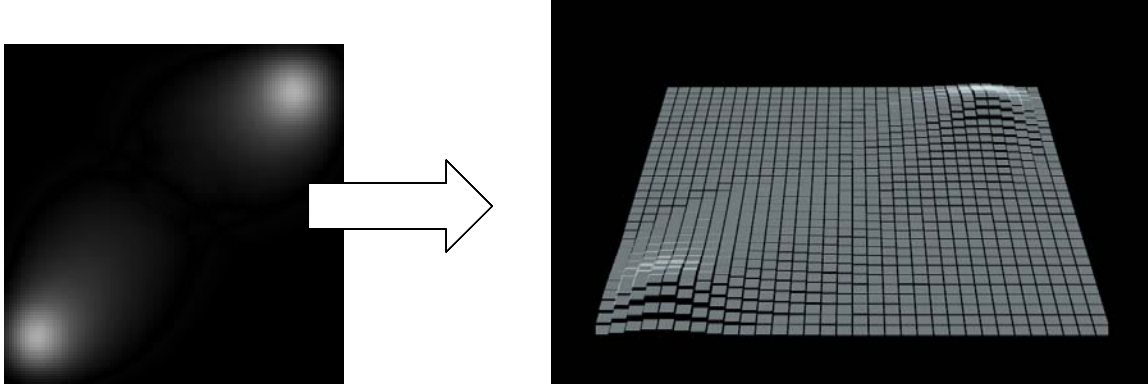


Figure 24. Example mapping between a 2D image and a 3D surface in Virtools game

Eventually, it was this tiled surface, that represented the 3D world, and which was presented back to the players. The Virtools program allowed for a custom choice of number of tiles – the image above shows a 30 x 30 tile matrix. So, eventually the game setup looked like this (the spherical mirror system is not displayed):

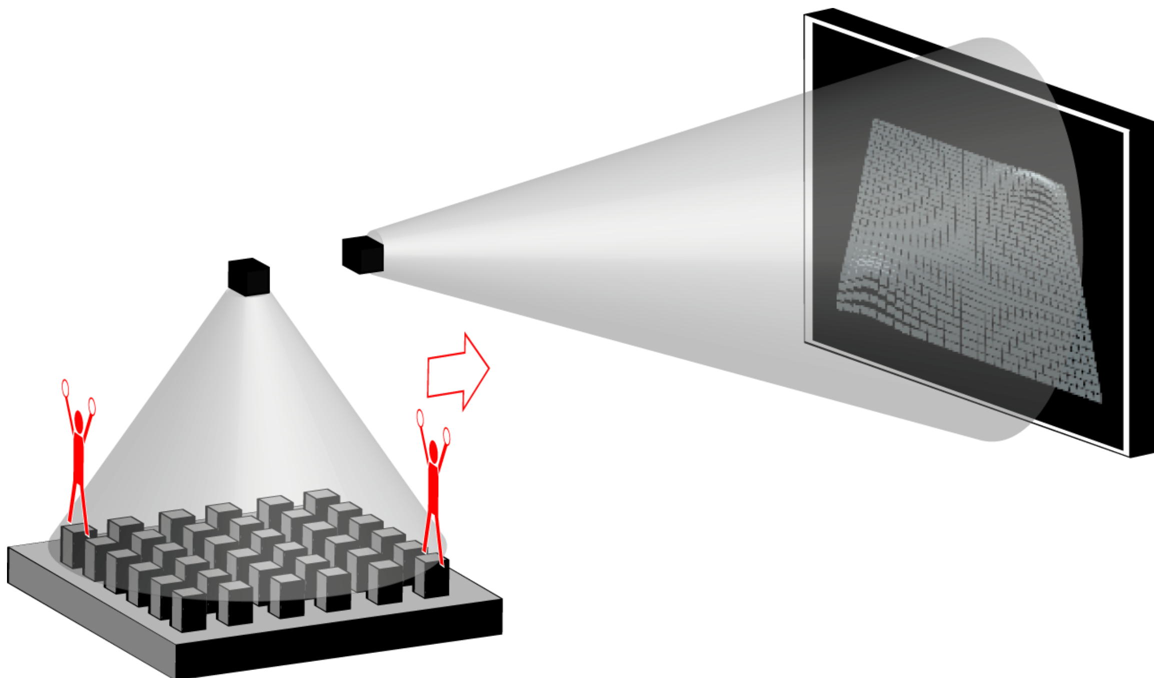


Figure 25. The game setup with the Virtools interpreted 3D world

The definition of the interaction eventually shaped the gameplay in the demonstration prototype. With the player having the possibility to induce rise and fall of waves in the 3D world, and move them as well, a model of an inert ball was placed in the 3D world, which could be displaced by an incoming “wave”. A number of collectable objects were placed on

the surface as well, and the goal for the players was to guide the ball towards collecting all the collectable items, by causing and guiding waves on the surface. A timer was set so that the objects were to be collected in a given amount of time. Eventually, the gameplay achieved in the demonstration prototype was that of a classic arcade game, such as Pacman. The choice for a tiled surface, as well as the usage of classic arcade sounds as a music background for the game, added to the arcade-style perception of the game.

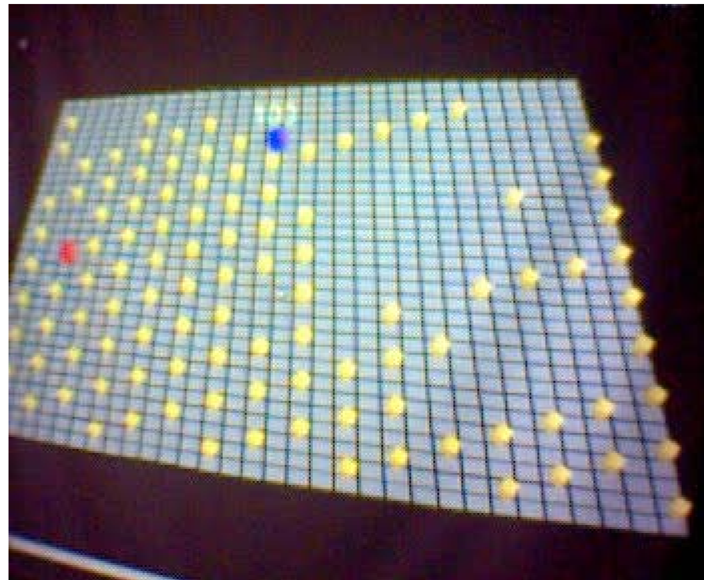


Figure 26. Low-res photograph from the actual Virtools game demonstration prototype

This idea, although simple, allowed for a demonstration of gameplay both in multiplayer mode (with an audience of about 10-20 people), where the players cooperate in guiding the ball, as well as in single player mode, where a single person must traverse the playground in the attempt to guide the ball.

The images below show the actual physical setup of the demonstration prototype of the game.

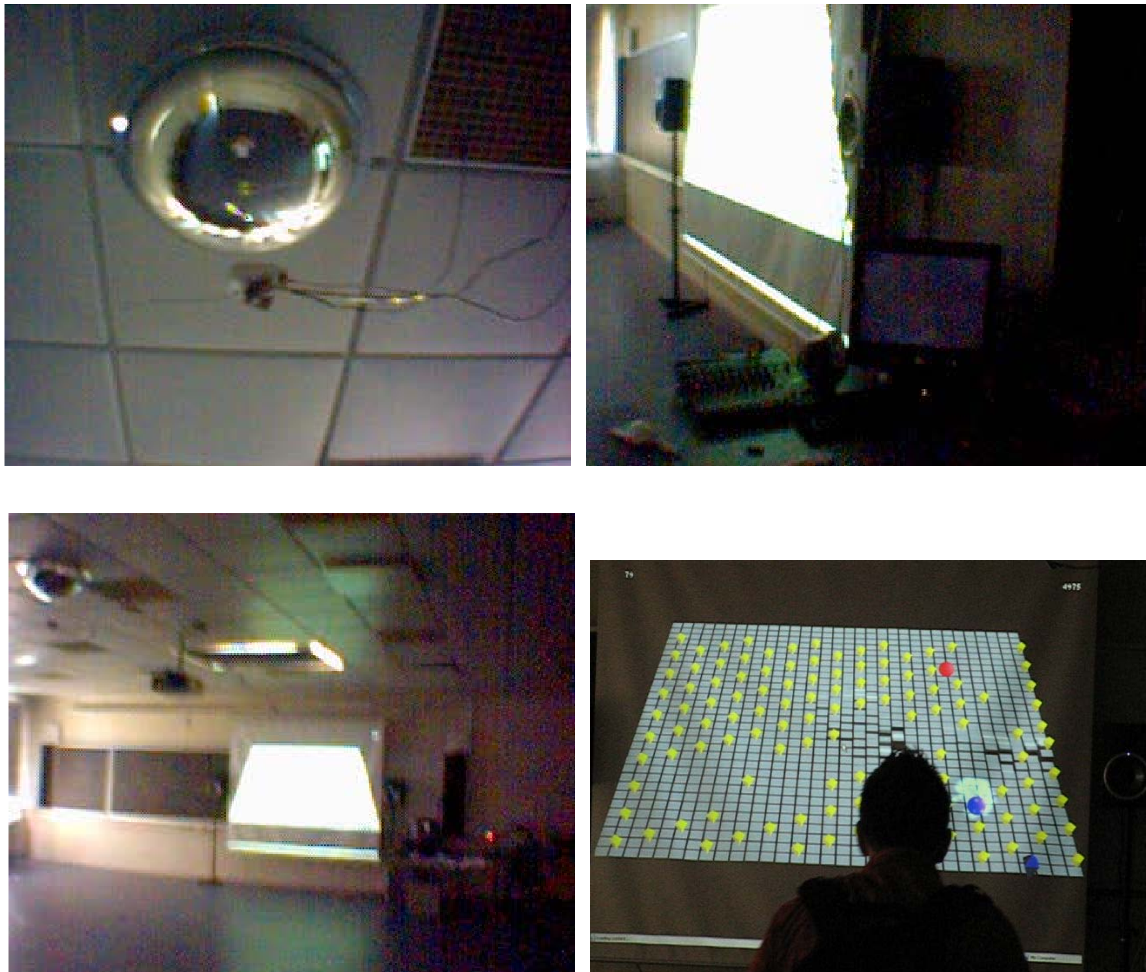


Figure 27. Actual demonstration setup: a) spherical mirror system, b) a view from the control terminal c) overview of the game setup d) scene of actual gameplay

8 Comparison to other methods of virtual world interaction

Essentially, the concept of a virtual world raises the question of offering a computer generated environment which is believable to the user, and two primary parameters to be discussed in that relations are immersion and interactivity. Audience interaction as a problem, always has the physical space that encompasses the audience as a limiting parameter. Also, the chief modal channel for representation of the generated virtual world so far relies on a singular visual output to the audience via a projection screen – meaning that the individuals in the audience need to be oriented in the same direction in order to perceive the visual feedback. Yet another limiting factor is the social one – on one hand, effort is made to track and display the interactive actions of the individuals within the audience, in order to provide instant gratification to an individual player, and motivate further interest in a game – on the other hand, the entire audience needs to be involved, so cooperation between the individuals within the audience becomes just as important.

Given these limitations, as a starting point one almost intuitively attempts to map the limits of the playground (the physical space where the audience is situated) into the limits of the computer-generated world itself – attempting to preserve the already existing bond of the audience with its physical environment. However, then the computer-generated world becomes but a map of the physical space, and it is displayed as such as visual feedback to help the audience achieve the sense of orientation and presence; the gameplay – the interactive storytelling – is created through individual actions within the audience, relating to a rendering of a given event in the computer-generated map.

Thus the gameplay happens on the playground, and it is essentially recorded, reconstructed and broadcasted back to the players. Both this project and the aforementioned Squidball game, can be seen as exemplifying this – and it can be especially seen in the similar choice of rendering angle of the 3D world – obviously meant to assist the audience members in using the image of the virtual world as a map of the physical space where they themselves are. Of course, in both cases the governing idea is that the audience members are expected to physically move, or rather translate, across the playground. We might call this a reconstructed playground approach, and it deals with the issues of mapping between the physical world and virtual world limits, and the actions within – thus, it demands tracking of translatory movement within the playground. The audience has a 3rd person view of itself, and

enhancing the interpretation of this view in the virtual representation creates the game. The events in the game have a geometric relationship to events in the audience playground. The benefit of this approach is that it can render a simple, yet meaningful 3D world representation based on the audience situation.

In this sense, the term “augmented reality” is truly more applicable to this kind of approach than “virtual reality”, especially in the sense of its current understanding. Leaving CAVEs and 3D goggles and other expensive / custom equipment, the increases in domestic PC computing power has led to 3D rendering of a virtual world as a standard in computer games in the recent years – and it is here where virtual reality enters the common knowledge. Almost all recent computer games offer some sort of an individual immersive experience, that deals with the interactive limitations of a home PC system – keyboard and mouse for input, and monitor, possibly speakers as well for output. In this sense, as the 3D world is represented on a 2D screen, the reconstructed playground approach can be compared to the contemporary concept of a PC game.

And the virtual world experience in PC 3D games is quite different than in reconstructed playground approach. An individual player controls an avatar (virtual character) through the keyboard and the mouse; the avatar is placed in the virtual world, whose bounds are not related to anything physical – the experience of the size of the world is through the relative experience of the size of the avatar within it. That means that events in the virtual world do not have a geometric relationship with the physical one – the individual player simply clicks to cause a translatory movement of the avatar within the world. Most of the games allow switching between 3rd and 1st person view of the avatar, and the camera that provides the window into the virtual world is usually made to follow the avatar. In that sense, the experience provided aims to simulate everyday perception stimuli to the best of the limitations of the 2D monitor screen as a medium. That being said, the sense of immersion stems from the sheer size of the worlds, the quality of the details being rendered, and the possibility of actions that can be undertaken with the world. The increases in available Internet bandwidth bring about MMORPGs (massive multiplayer online role-playing games), where these games truly live up to the term virtual worlds – as they become virtually inhabited by other player’s avatars (see for instance PlaneShift[10] or Anarchy Online[11]). The possibility for chatting and communication between players allows for cooperation within the virtual domain, where such is non-existant, and even impossible, in the physical domain. In some cases, the actions undertaken by the players become milestones in the

history of the virtual world – and as such influence the general story that guides the respective gameplay.

As said before, the reconstructed playground approach aims to assist physical cooperation between players, relying on the mapping between the physical playground and the virtual world, and aiming to represent the 3D virtual world both as an augmented version of the events in the playground; and as a map to assist the players – since the players' awareness of the immediate physical space, where the actual gameplay happens, is already automatically disturbed (as they will need to turn their eyes and attention to the visual feedback anyways, if they are to participate in the game). As such, immersion in the sense of current PC home gaming is impossible in the reconstructed playground approach – and for that matter, in any audience interaction system that relies on a 2D projection for visual feedback. In the PC game context, the camera follow of the avatar (as the window into the virtual world), allows for individual rendering of the 3D world which relates to out individual everyday perception, which helps in minimizing the conscious effort of the player to get immersed in its avatar and thus in the virtual world. The camera, and the view obtained, is basically driven by the individual actions within the world. This is almost irreconcilable with the reconstructed playground approach, as a single camera view is shared between all the audience participants, and as such it must provide a view that all can relate to while playing – so an individual action should not change the camera view, although it should cause an event in the game world.

If this demand for a reconstructed playground is relaxed, as in the template-matching[1] or Cinematrix[9] cases (where there is no geometric correspondence of the audience space with the visual feedback, and no individual action tracking) immersion similar to the home PC game context can be obtained – as the visual feedback is no longer an orientation map of the physical space. In a reconstructed playground, the players share one visual feedback, but their attention may be on different parts of the image – mostly on those where they have the influence at the moment. When the individual actions are integrated into a singular response of the audience, the visual feedback indicates only a singular action – so the audience members focus on the same action, and thus the same part of the image. As such systems have been used for say, maze navigation, it is possible that they can be used for a 3D world navigation, quite similar to the home PC gaming context – except that here, all of the audience members would control (and identify with) only one avatar.

In any case, these are more less differences due to technology available to the audience participation context. Imagine each member of the audience equipped with a tracking system

which can track body part positions (hands, feet etc – such that they can be mapped to an avatar skeleton) and goggles that are both see through and can render a 3D computer image (such as maybe LCD shutter goggles), and audience interaction from an first person perspective for each individual member of the audience can be envisioned – thus bridging the gap between the home PC game and audience game experience, and deserving a description of a mixed reality system. Maybe systems such as LiveActor[12] might live up to the challenge – but it would still be hugely expensive to implement such a solution – not to mention the hassle of dressing even a small crowd of 150 people into motion capture suits. The possibilities that would open are though huge – gesture recognition could be applied for game interaction – and even in a context of two people, Tekken-style combats could be envisioned, where "combo hits" or "spells" could be initiated by a given gesture, and rendered as an effect in the virtually augmented world (augmented since we would still rely on the actual physical space to set the limits of the virtually generated one). Similar systems already have appeared – consider for example Human PacMan[13] from Mixed Reality Lab in Singapore, where the playground is extended through an entire city. Obviously, such approach goes far beyond mere image processing as input interaction of a group of people.

9 Networking approach

9.1 Networking overview

As displayed on Figure 6, all our hardware – an IP camera, and two Windows PCs – was connected in a local area network via Ethernet, where each hardware component performed a specific task in the processing chain, and communicated the results to the next machine in the chain. A more detailed diagram is displayed on the following image:

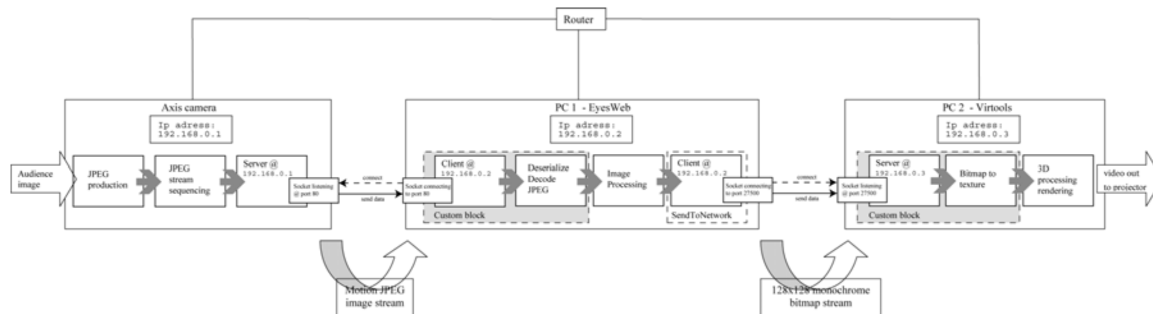


Figure 28. A detailed diagram overview of the network setup

We have implemented a distributed computing approach (for a short overview, see [14]) to the problem of audience interaction with a 3D world, utilizing the client/server model[15]. The client/server communication was implemented in the standard way using sockets, "endpoints of a two-way communication link between two programs running on the network [16]". As the diagram suggests, this communication is applied so that in effect, we get a series processing chain.

9.2 Initial development and server/client structure

At start of development, the Axis camera had built in network functionality, as it represents an IP network camera – and as such, features a built in web server. In addition, the EyesWeb application in that particular version (3.3.0) offered the possibility for network connection between two EyesWeb applications, via its native blocks SendToNetwork and ReceiveFromNetwork. As both EyesWeb and Virtools can be extended via their respective software development kits, we developed extension plugins for both of them to implement the necessary interconnectivity, using C++ and the standard Windows sockets library[19]. As

both EyesWeb and Virtools represent graphical programming environments, the functionality is organized into visual blocks. As such, our extension plugins, eventually became graphical "blocks" or "modules" in the EyesWeb terminology, and "Building Blocks" in Virtools terminology.

The starting point of the networking development was the native server functionality of the Axis camera, and the native client functionality of the SendToNetwork block. The server/client functionality is implemented using sockets and in both cases, the sockets can read and write data – the difference is mostly in the roles in the interaction in the initiating phase for establishing communication. A very general model makes the following distinctions between client and server (from [15]):

Properties of a server:	Properties of a client:
Passive (Slave)	Active (Master)
Waiting for requests	Sending requests
On requests serves them and send a reply	Waits until reply arrives

This means that the role of the server socket in the initiating phase is to wait for requests, and the client socket role is to send a request – upon which a communication link is established and data is exchanged. However, after communaction is established, both sockets can "speak both ways" or (both send and receive data), although the model implies that the server sends data, and the client receives it (as is the case in most client/server communication).

With this in mind, we had to specifically develop a client for EyesWeb, which would match the server on the Axis camera, and a server for EyesWeb, which would match the client functionality of the EyesWeb SendToNetwork block. This set the situtation where the EyesWeb application actually has two clients (and no servers), which can connect to the respective servers of the Axis camera and the Virtools application – which automatically sets a connection sequence as well – for the clients on EyesWeb to connect successfully to the two servers, those servers have to be running beforehand, as they are the ones that wait for requests: so the Axis camera and the Virtools application (or the respective servers) have to be started before the EyesWeb application is started. In addition, the link between EyesWeb and Virtools is such that the client socket (the one initiating communication) is the one that sends data, instead of receiving it, as it should according to the general client/server model –

however that is the way the native functionality of the SendToNetwork block is implemented; so a matching server had to be developed for Virtools.

All of the machines were connected to the router using normal RJ-45 networking cable, forming an Ethernet local area network. The router allowed 1 GB/s throughput, however only the Virtools computer had a matching network card – the other PC and the Axis camera sported 100 MB/s connections, which effectively limited the speed on the LAN. On the network, the computers are assigned local IP network addresses (192.168.*.*) – the sockets themselves are bound to a port number. Note that only the port number of the server socket which waits for connections is important – once connection is established, both server and client negotiate new sockets on other available port numbers through which they will exchange data (see [16]). The Axis camera is meant to be viewed with a web browser, so it represents a HTTP web server. HTTP communication is defined to occur on port 80, so the corresponding client socket in EyesWeb connected to this port, whereas we chose 27500 as the port number for the Virtools server listening socket.

9.3 The camera from a network perspective

In order to describe the communication setup in more detail, let us now reconsider parts of Figure 28.

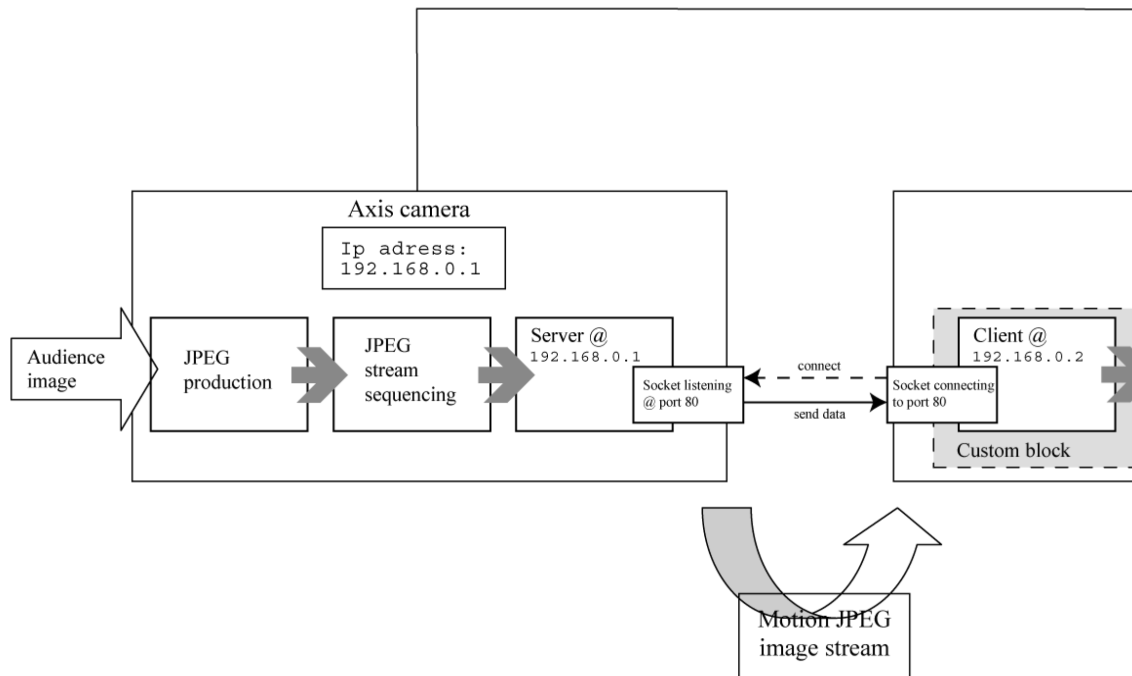


Figure 29. Functionality of Axis camera and the link between Axis and EyesWeb

It can be useful to conceptualize the functionality of the Axis camera, although most of these aspects are completely transparent to us as developers. Basically, we need to be aware that the camera has to perform some sort of processing to acquire a digital version of the image from the CCD chip. After the image has been acquired, it is compressed into the JPEG format, and made available to the internal camera web server. Upon request, the last available JPEG image is served through the network, as a series, or stream, of 8 bit characters. Only the JPEG format is available to the end user of the Axis camera, and it is available either as a single (most recent) image, or as a delimited sequence of JPEG images. The JPEG compression is the one that encompasses the two dimensions of a given image – on the other hand, video compression algorithms utilize differences between image frames as a factor in the compression too. As such, a sequence of JPEG images is not "officially" a video compression routine, and it is commonly known as motion JPEG or m-JPEG. As such, the Axis camera needs to take care of the sequencing of images and creation of the stream, and making it available to the users.

It is here where a network camera differs from standard cameras used so far in PC applications running EyesWeb – they usually either provide an analog output, which requires a dedicated frame-grabbing driver installed in the OS, or a digital output, carrying a digital video (DV) signal, which implements video compression routines, and again requires a dedicated driver being installed in the OS. EyesWeb, as an image processing application, provides access to live input of these kinds of cameras, but not to network cameras. Therefore, the task in developing the EyesWed plugin was not only to establish the connection to the camera and retrieve the image data, but also to make it available to EyesWeb in the image format it can understand.

9.4 EyesWeb client and networking to the camera

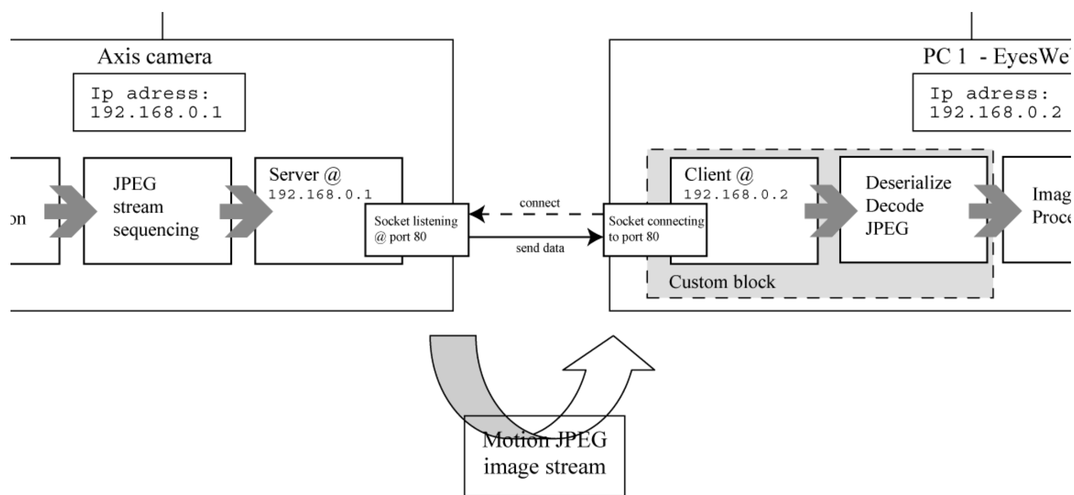


Figure 30. The task of the EyesWeb client

Going back to the two flavors of availability of the camera image: either as the latest available JPEG image, or as an m-JPEG sequence of them – we can actually use both to acquire animated video. In the first case, one can demand an image from the Axis server, and receive it – however, the socket communication rules demand that the socket is destroyed as soon as the transfer has completed; that is, a socket cannot be reused: "Unlike a telephone, sockets are disposable, and cannot be reused once they've been connected to something. To make another call, we have to create a new socket[21]". That means, in order to receive a consecutive sequence of JPEG images, one must reinitiate connection to the server each time a transfer of a single JPEG image has completed. On the other hand, when receiving an m-

JPEG stream, connection is established only once, and the entire length of the transfer is not known; the Axis server begins to stream a consecutive sequence of JPEG images immediately after connection establishment. In that sense, we spare the resources on re-initiating connections when using m-JPEG, so we opted for this version in the development.

Thus the client socket in the EyesWeb patch initiates connection to the Axis camera only once, and it follows the regular process of a client connection to a web server (see [18]). The client connection code we used is given in Appendix C, and the only difference with the standard implementation, is that we wanted to allow the user the possibility to enter the host address of the server from the EyesWeb GUI, either as a Internet DNS name, like "www.yahoo.com" or as a local address, like "192.168.0.1" – since these two procedures differ slightly, we had to implement a check which one is actually used. After establishment of a connection, a standard GET request is issued to the server, to ask for the m-JPEG stream (code not provided).

9.4.1 Stream reception and extraction of frames

The usage of m-JPEG, means however that the EyesWeb plugin should feature an independent receiving thread whose only job will be to continuously receive the Axis stream data, and extract individual images from it – and to do that, the images have to be somehow delimited.

If we look at a TCP communication transcript from an Axis camera streaming m-JPEG, we can observe how the images are delimited within the stream:

```
0060   33 2E 19 83 0E 98 3C D7  9D F1 9E 32 0F D6 A5 8E   3.....<....2....
0070   E6 E2 2E 52 77 1F 8D 2B  0B D9 9F FF FF FF FF D9   ...Rw..+........
0080   0D 0A 2D 2D 6D 79 62 6F  75 6E 64 61 72 79 0D 0A   ..--myboundary..
0090   43 6F 6E 74 65 6E 74 2D  54 79 70 65 3A 20 69 6D   Content-Type: im
00A0   61 67 65 2F 6A 70 65 67  0D 0A 43 6F 6E 74 65 6E   age/jpeg..Conten
00B0   74 2D 4C 65 6E 67 74 68  3A 20 34 30 38 36 37 0D   t-Length: 40867.
00C0   0A 0D 0A FF D8 FF FE 00  0E 54 00 00 08 CC 25 00   .........T....%.
00D0   00 00 00 00 00 FF FE 00  0E 49 01 36 01 A5 01 00   .........I.6....
```

As it can be seen, each image is delimited with a header, containing the string "--myboundary" and indicating the size of the next JPEG image in bytes. That is enough information to deserialize the stream and extract individual images – however it is not a trivial matter. In order to receive continuously, the thread would have a form of a

conditionally infinite loop – ending only when the entire EyesWeb patch is terminated. Within every step of the loop, a receive command is performed, which picks up all the bytes that have arrived from the camera up to that point and stores them into an array. The amount of received bytes per each receive call is different and depends basically on the situation on the network (speed, congestion, etc.), but also on the fact that the processor will in reality take different amount of time to process everything in between two receive calls. In addition, the JPEG images do not have a fixed size either – since the end file size of a JPEG image depends on the image content as well as the compression level – so less details, when the capture scene is say darker, mean a smaller JPEG file size, for the same amount of compression. Depending on the situation, we have observed numbers of bytes being picked up by a receive call being from around 2000 to around 8000 – whereas we can see that for a 640 x 480 JPEG, used in the above example, we get a size of around 40-50 Kb. That means that the receiving loop has to go through several iterations before having a full image – but it also means that we generally do not know neither the size of the next JPEG, nor the size of the next received data chunk. Thus we basically do not know in advance when the images start and end, so some algorithm has to be implemented that will correctly extract the images.

This was achieved through a double buffer strategy. When the data packet is received each call, it is searched for the delimiter string and if found, storage in the first buffer begins. The consecutive packets follow the same procedure, being added to the previously received ones in the first buffer. When a new delimiter string is found, the current packet is finally assembled, and it is copied, without the header (only the raw JPEG data), into the second buffer. The first buffer is thereupon erased and the whole process starts all over again – while the second buffer holds the last received raw JPEG data long enough so that it can be decoded and used in the EyesWeb patch. This was implemented in a single thread function in the EyesWeb plugin, which we called ThreadProcess, given in Appendix D. This algorithm takes into account that we do not know the packet size between two receive calls, as well as the JPEG size. This means that we cannot on beforehand predict where the boundary string and the content length string will appear. For instance, a "--myboundary" delimiter string may appear on the end of a receive packet, whereas the content length may be in the next one – and the algorithm takes account for that. However, it does not take into account that either the "--myboundary" string or the "Content-Length" string may themselves be cut into two consecutive receive call packets, in which case they are not registered. The code simply registers those as dropped frames and reinitializes to prepare for the next JPEG in line. In spite of this, the dropped frame rate is still quite low, maybe staying at a dropped frame each couple of minutes.

9.4.2 Decoding of the received frames

The reception and correct extraction of a single JPEG raw data is still not the end of the plugin work. The raw data acquired represent a JPEG data structure, as if it would be written on a file – and since it is a compressed format, this data does not represent picture pixel data. Thus, the JPEG image needs to be decompressed and made available in the image format that EyesWeb can work with – and that is the other task of the EyesWeb plugin. That however, comes out to be slightly problematic. In essence, EyesWeb uses the Intel's Image Processing Library (IPL) (this once free library has been discontinued and replaced with the commercially licensed Intel Performance Primitives (IPP) library[22]) or rather, a small part of it made freely available version to the public, aimed at computer vision and called OpenCV[23].

Through the EyesWeb SDK, the developer has access to common OpenCV methods, and those include reading a JPEG image from file. However, they work strictly with the hard disk file system (require a filename as input), and cannot decode a JPEG that we have obtained in memory. On the other hand, the foundation Windows classes have the same approach to reading both files and memory, and decoding a JPEG is pretty much a straightforward procedure – however, the final image pixel data is obtained as a Windows Bitmap class, and there is no legal method from OpenCV that can convert a Windows Bitmap into the internal Intel image format, which is called IplImage. Intel did offer a free JPEG library (IJL [24]) – but it has also been superceded by IPP, and the free JPEG library from the Independent JPEG Group[44] seemed a rather specialized library, conversion is not straightforward, and there was simply not enough time in the project duration to go through its learning curve. Luckily, a hack was discovered, where the internal data representations of a pixel bitmap both in the Windows and Intel formats is the same, so a simple casting will do. Assume `intel_pic` is a variable of OpenCV class IplImage*, and `ms_pic` is a variable of Windows class BITMAP – then the following can be used to obtain a Intel format from Windows:

```
intel_pic->imageData = (char *) ms_pic.bmBits;
```

So, we simply used a standard Windows method to read a memory stream, and obtain a decoded image from the JPEG data using OleLoadPicture. As soon as the decoded image was obtained, it was copied into Intel's IplImage format, making it available for further EyesWeb

processing. This functionality of the EyesWeb plugin was implemented in the TransferBitmap function, listed in Appendix E.

In retrospect, we have implemented the following functionality in our EyesWeb client plugin:

- A client socket establishes connection to the server on the Axis camera, requesting a m-JPEG stream
- A client thread process runs in a loop, and uses a double buffer strategy to assemble the incoming packets received at each call. The first buffer is populated with incoming data, as soon as a new delimiter arrives, the raw JPEG data is extracted from it, and copied in a second buffer.
- A function is called that transfers the data in the second buffer into a bitmap pixel data format which is usable in EyesWeb; meanwhile
- The first buffer resumes with receiving the new image.

The source code for this plugin, as well as some others developed in the course of this project, has been made publicly available on the Internet, in a discussion forum for EyesWeb users[25]. The compiled client plugin took the form of an EyesWeb block, which can be seen on the screenshot in Appendix K as the block called “EywIPMJpg”.

9.4.3 Further processing

The EyesWeb patch, being in possession of usable image data, resumes with the processing, described elsewhere in this report:

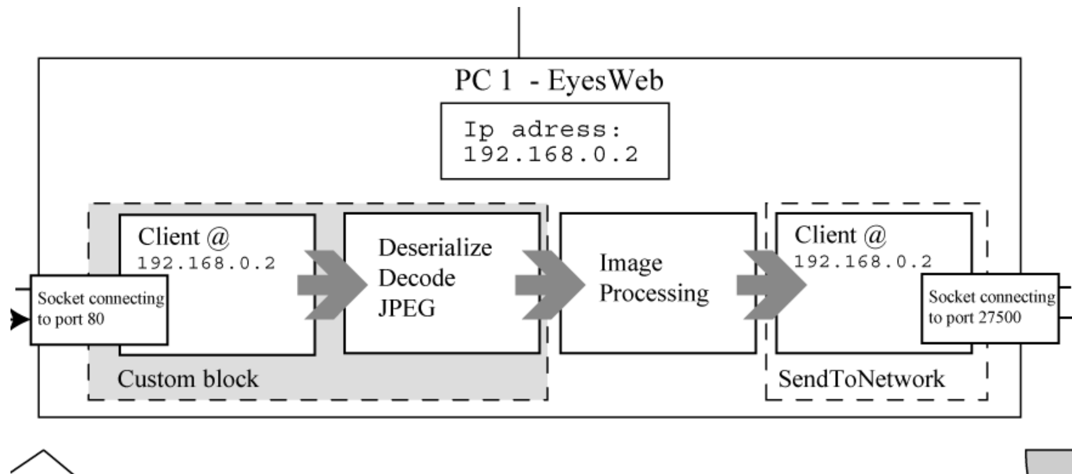


Figure 31. Functional overview of the EyesWeb patch

The image processing was triggered each time a new image was decoded, and in total, we obtained a framerate of about 17 fps, retrieving a m-JPEG stream of JPEG images 320x240 pixels in size over a 100 Mb/s connection. Obviously, this framerate dictated the framerate of processing, and sending to Virtools as well – and it means that it was the limiting framerate of the world surface update in the virtual 3D world. As soon as the image processing of a frame finished, it was sent to Virtools, using a native EyesWeb block called SendToNetwork – this part of the networking is described in the next section.

However, Virtools achieved framerates of about 60 fps while generating the surface, and since the generation of surface distortions (based on the received image) was extrapolated between frames, a smooth and unified experience was obtained which covered both the surface animation and the animation of other objects in the game (the ball) – in spite of the fact that the base images arrive at a 17 fps rate, lower than the standard TV framerates of 25-30 fps.

9.5 Virtools server and networking to EyesWeb

Let us now consider the connection between EyesWeb and Virtools. It was mentioned elsewhere that the end result of the EyesWeb processing chain was a monochrome bitmap, 128x128 pixels in size. If using an 8-bit grayscale bitmap, each pixel of the image can be described with one byte – or one unsigned `char` in C/C++ terminology. So, in this case, since the network transfer we work with operates on a byte level – each character of the stream in this part of the connection actually does describe pixel data, as opposed to the connection between EyesWeb and the camera. Thus, we know that each frame of the processed video will be exactly 16384 bytes = 16 KB in size – which made the coding considerations for the image reception on the Virtools side much easier that the ones already described for EyesWeb.

Again, we used the native EyesWeb block called SendToNetwork, to send this 128x128 bitmap to Virtools, and as it is somewhat described in the EyesWeb documentation, it shall not be discussed further. It does behave like a network client, and sends the raw bitmap data pixel by pixel, thus we needed to code a corresponding server for the Virtools plugin, developed in this project. The plugin is based on the VnetSync plugin by Sphaero.org[26], and it reuses some of its architecture.

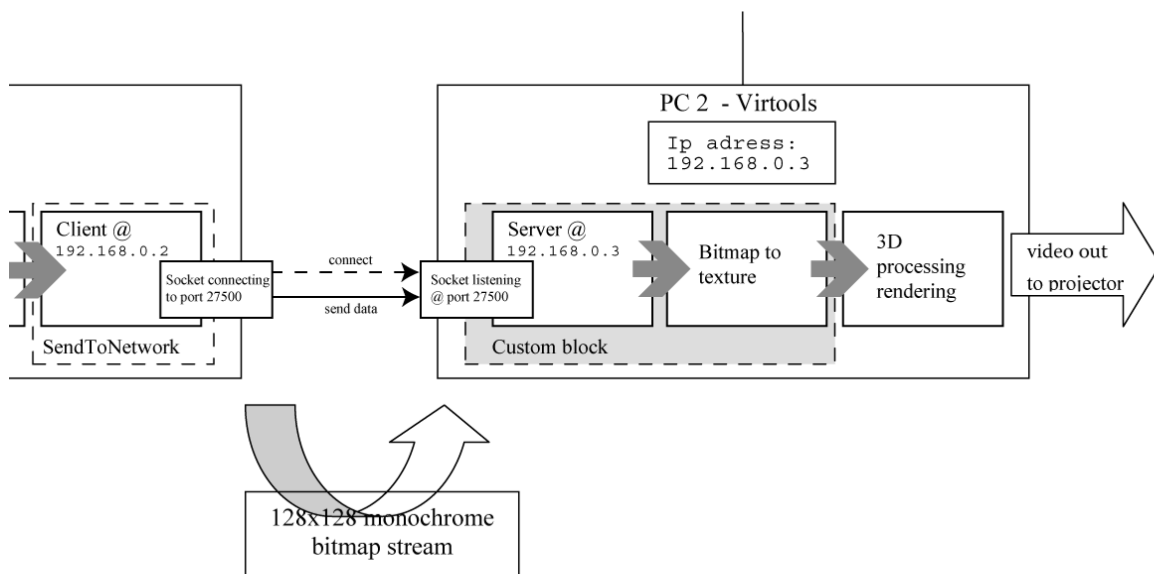


Figure 32. The task of the Virtools server

The original Sphaero architecture of the plugin, features a so-called network manager – a process that runs in the background and takes care of all the networking details, namely the

transfer of data. Then, separate buildingblocks can be developed, that use the same network manager, based on their functionality. The original Sphaero plugin was meant to enable master/slave synchronization between two Virtools applications – our extension allowed for a server and client building blocks, that use the same network manager, and can be used to connect to two different application. The server was of course intended to connect to EyesWeb as described in this context – and we experimented with a simultaneous connection to a Max/MSP application from Virtools as a possible use for the client.

9.5.1 Stream reception and extraction of frames

In any case, when the Virtools patch was started, the network manager initiates a listening server socket, using the code described in Appendix F, and starts listening for incoming connections. Here the SDK plugin architecture of Virtools comes into play – since this event happens before the entire Virtools patch actually starts playing, this event does not occur in its own independent thread, thus the while loop blocks the patch during its entire duration. Using a timeout setting of 10, we would limit this period to 10 seconds, where we would have to quickly turn on the EyesWeb client on the other computer in order to initiate a connection to the Virtools listening socket.

Once the connection was established, the blocking stops and the network manager would run in the background normally, receiving the data that the EyesWeb SendToNetwork block automatically starts sending upon successful connection. Especially, the Virtools SDK allows for a time slot before a frame is rendered, entitled PreProcess Frame – and it is within this method that the network manager uses the server socket to execute a receive call. The listing for this part of the data transfer is noted in Appendix G.

Technically speaking, PreProcess is not an independent thread of its own, so an endless loop for receiving, like the one used in the EyesWeb client, would simply block the Virtools patch entirely (however, PreProcess does repeat in tact with the Virtools rendering refresh rate). That problem, as well as the fact that we in advance know how large each image is, made us set the socket options of the server socket as a non-blocking one. That means, that if the expected length of a packet that should be received is made known to the algorithm, then each receive call would fail if the received packet length is not yet 16388 bytes – this exits the while loop, and the program continues, until next time PreProcess is run and the state of the receiving buffer is checked. Here, even if we have several receive calls before we have

obtained the full bitmap image data, still the call will be successful only when we have reached the expected length – which means that the packet assembly process is done for us in the background by the OS, instead of us patching the parts manually using double buffers as in the EyesWeb case. Eventually, the 16388 byte packet is received in a char array buffer.

9.5.2 Decoding of the received frames

As soon as PreProcess is finished, the main procedure of all the building blocks is called during frame rendering – the main routine of the EyewWeb server building block as well. It is so here where the received char array buffer is transferred to a data structure that represents a texture within Virtools. The routine used with this intent is listed in Appendix H. Once the texture has been populated with the received contents, it is ready to be utilized within Virtools in the 3D interpretation algorithm, which creates the virtual world surface based on the image – as well as the rest of the game engine.

The source code for this plugin, has been made publicly available on the Internet, in a discussion forum for Virtools users[27]. The compiled plugin took the form of two Virtools building block – the described server for EyesWeb and an experimental client, which can be seen on the screenshot in Appendix L as the blocks called “NS_EywServer” and “NS_Client”.

9.6 Relationship to OSI layers

As this project relied in part on customized networking, it might be interesting to see how does it relate to the OSI model. OSI (Open Systems Interconnection) is a "layered abstract description for communications and computer network protocol design … a hierarchical structure of seven layers that defines the requirements for communications between two computers[28]", and it helps in a hierarchical understanding of the network processes, similar to those in a postal service (see [29]).

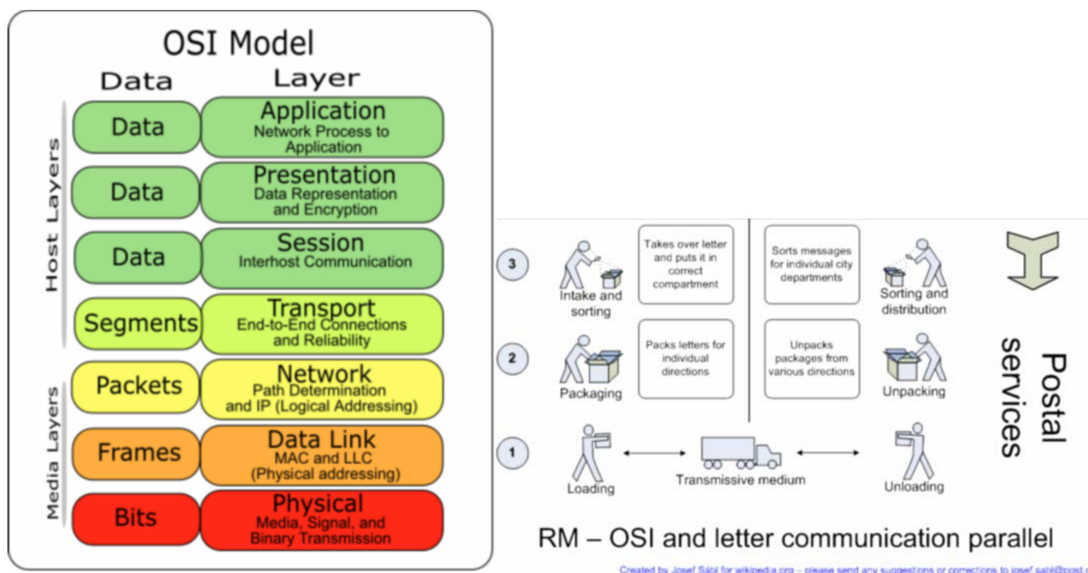


Figure 33. The OSI model and comparison to postal communication (From Ref. [28])

Although the OSI model is a complicated standard, which never got fully implemented, it helps understanding the reasoning in computer communication through logical abstraction into seven hierarchical layers, ordered so that each layer uses only the layer below, and provides functionality to the layer above – which makes it easier to percieve the functionality of several layers together, which are referred to as a stack. In any case, as we have used TCP/IP, a very common method in network communication which is already described within the OSI framework, this should make the description of our project much easier.

To begin with, the first (lowest) defined layer is the **physical** one, which "defines all electrical and physical specifications for devices[28]". This metaphor especially in the modern OS-es, allows for the separation of the networking code and functionality from the actual medium through which the network is established – cable, or as it becomes more available

with time, wireless (radio). That means in an ideal case, that although in this particular project, we applied networking cable as a medium, we might just as easily reuse the same game engine and corresponding plugins for a wireless network – which would definitely make setting up of the hardware much more transparent, and would extend the applicability of the system.

Above the physical layer, there is a **data link** layer, which also operates on a physical level – within the networking card – so it uses hardware networking adresses (MAC or Media Access Control adresses, which are hard coded into network cards). The data link layer is meant to take care of the data transfer, and detect errors that happen on the physical layer. Ethernet is a layer two or data link protocol, and it is this particular protocol implemented in all the hardware in our project – as with most off-the shelf network hardware that uses RJ-45 network cable. So, we can pretty much perceive these two layers as one, since the network card interface (layer two) is pretty much determined by what sort of a medium (layer one) it is made to operate on, although Ethernet as a data-link may be applied both to wired and wireless network media. Another important thing to mention about the data-link layer is that "This is the layer at which bridges and switches operate. Connectivity is provided only among locally attached network nodes. [28]". As our project was implemented in a local area network, using only a switch, technically speaking using adressing schemes for an Internet network is not necessary – however, since the coding is nonetheless the same, using the standard libraries offered on modern OS-es, our project would just as well work over the Internet – say with camera placed in one own and computers that process in another. Of course, the performance will be limited by the traffic conditions, and it is questionable what sort of special application might arise from that kind of a setup.

The possibility for an extended network such as the Internet comes already on the next level, the **network** layer. The protocol that operates on this level is called, not surprisingly, Internet Protocol, or IP, and it "provides the functional and procedural means of transferring variable length data sequences from a source to a destination via one or more networks … performs network routing, flow control, segmentation/desegmentation, and error control functions. The router operates at this layer -- sending data throughout the extended network and making the Internet possible[28]"

Right above this layer, we have layer four or **transport** layer. This layer "provides transparent transfer of data between end users … controls the reliability of a given link … [and] can keep track of the packets and retransmit those that fail[28]". This is actually the

level of definition of sockets. Sockets can be connection-oriented (stream sockets) or connectionless (datagram sockets), and each has their own protocol (see [17] [30] [31]). In essence, in a connectionless protocol (such as UDP), all that we have to send is one finished message with a known size, so there is no need for a connection as such – we think of the message as simply a letter. Hence, there is no guarantee that two messages sent one after another will arrive in the same order. However, we may as well have a message that has not finished yet, so it does not yet have a known size – as is the case with any broadcast. Hence the need to sequence the message in smaller packets as it is generated, and the need for a virtual connection between the two hosts, through which these packets will be sent – and finally, a need for a guarantee that the packets will arrive in order so that the message can be reconstructed as they arrive.

A connection-oriented protocol allows for establishment of a connection from a client to a server, in the same analogy of a phone call – once the connection is established, we have a virtual circuit established, so data can be streamed to either direction – and the protocol ensures that the data is delivered reliably. Reliable delivery in this case means that the packet reception is acknowledged – so if a packet is not delivered, the sender is notified to retransmit; and there is a guarantee that the packets will arrive one after another. TCP, or Transmission Control Protocol is the layer four protocol that governs connection-oriented sockets, and this is the first level where we had influence of choice as developers – as we implemented the sockets in a client/server architecture so they will allow for streaming of data.

Actually, TCP/IP appeared as a working technology before the definition of the OSI model. The image on the left shows a comparison of the TCP/IP stack with OSI layers.

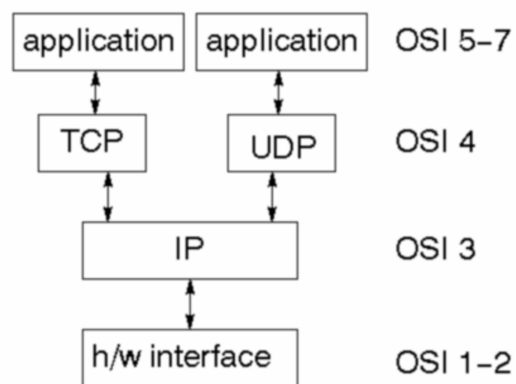


Figure 34. Comparison of TCP/IP stack and OSI layers

We should note that Winsock, as a library that we used to program the socket functionality, is an interface standard “between applications (layers 5 and above) and the transport (layer 4) [28]”.

The next, fifth, layer in the OSI model is the **session** layer. This layer "provides the mechanism for managing the dialogue between end-user application processes … establishes checkpointing, adjournment, termination, and restart procedures … is responsible for setting up and tearing down TCP/IP sessions. [28]" This functionality is established by the networking plugins that were developed for both EyesWeb and Virtools, since it was there where the connection was established using the sockets, as described earlier – and of course, where the connection terminates. Especially it is the EyesWeb plugin that manages more low-level aspects of the session, where we implemented a double buffer strategy to correctly recreate the stream, and provided for a basic mechanism to detect and discard dropped frames.

The sixth layer in the OSI model is called **presentation**, and deals with data representation and encryption. This is another area of operation of the EyesWeb and Virtools plugins that we developed. In both cases, we receive some stream of data, and that data needs to be interpreted within the given context. For instance, in the EyesWeb case, we receive an m-JPEG delimited stream – once we have extracted a single JPEG, we also need to make it available in a pixel bitmap format that is usable within EyesWeb, as discussed before. This is a task performed by the plugin, and it represents functionality of the OSI presentation layer. The same thing is valid for the Virtools plugin, although the conversion process is much more straightforward there – there is no special decoding of the received frames from the stream, as they do represent pixel values, however, they need to be passed into the internal bitmap representation that Virtools can work with, which in this case is the notion of a texture.

The final, top layer in the OSI model is the **application** layer. This layer "interfaces directly to and performs common application services for the application processes[28]" and HTTP (Hyper Text Transfer Protocol) represents a layer seven protocol. Obviously, this layer logically encompasses all the processes that are performed within the application once the arrived data is in a format understandable to the given application (or for that matter, all application-native processes that lead to a generation of a message). In the EyesWeb case, this encompasses the entire image processing chain, whereas in Virtools, it is the entire game engine – including the 3D conversion algorithm, which creates the virtual world surface.

So we might look at both cases of Virtools and EyesWeb plugins, which both represent receivers and decoders of data (although from a socket perspective, one is a server and the other is a client, respectively), and attempt to fit the into the OSI model framework:

OSI Layer	*EyesWeb + custom plugin functionality*	*Virtools + custom plugin functionality*
Layer 7 **Application**	Image processing of received image	3D surface generation from received image and game engine
Layer 6 **Presentation**	m-JPEG stream – a received frame is a JPEG image, which is decoded and – provided into EyesWeb internal image format (as Intel IplImage*)	Char pixel data stream – a received frame is a grayscale bitmap image, which is – provided into Virtools internal image format (as texture CKTexture*)
Layer 5 **Session**	TCP/IP Session establishment – client initiates connection to server, – data is received in buffer, – extracted frame by frame, – check for dropped frames	TCP/IP Session establishment – server listens for incoming connection, – data is received in buffer, – extracted frame by frame (via TCP)
Layer 4 **Transport**	TCP – dictated by choice of stream, connection-oriented sockets	TCP – dictated by choice of stream, connection-oriented sockets
Layer 3 **Network**	IP	IP
Layer 2 **Data link**	Ethernet	Ethernet
Layer 1 **Physical**	100-BaseT (RJ-45)	100-BaseT (RJ-45)

Winsock - Windows Sockets interface

- Provided by the OS

- Custom plugin

Figure 35. Comparison table of Virtools and EyesWeb with the custom plugins, fitted in the OSI model

10 Conclusion and perspectives

This audience interaction game project was a great learning experience. As almost all other projects in this area, it uses a live video capture of the audience as an input interaction signal – so it raises issues in image processing, networking, 3D coding as well as considerations in interactivity and immersion. Although video imaging seems already a standard in this kind of projects, the approaches used in its treatment are different, and we have proposed an approach where a catadioptric system (mirror + camera) helps overcome the limitations of available space, when the aim is to acquire a top orthogonal image of the audience. Such filming position allows for a specific interaction, where an audience member should perform a motion local to the boundaries of the physical playground (auditorium), which is detected by the system and displayed correspondingly within the virtual 3D world boundaries.

An audience member can interact either by sitting/standing still, and performing a waving motion of the hand – which would be indicated as a static effect in the 3D world; or by performing the same motion while walking through the playground – in which case the rendered effect correspondingly translates throughout the 3D world boundary. As such, individual input from two distinct audience members is detectable, but not uniquely identifiable – that is, two audience members interacting with the system will cause two distinct effects, but the same situation will be obtained if the audience members switch places, which is a concern in audience interaction (some approaches average out the participation of the audience, making individual input undetectable).

This is due to the simplicity of the image-processing algorithm used to detect local motion within the bounds of the scene (a difference based analysis) – which simply indicates areas where pixels change over time within the image (where motion occurs in the scene) as brighter pixel areas. The usage of a catadioptric extension to the input system, requires a closer look into image processing algorithms – here we have applied a geometric pincushion distortion filter, in aim to correct the curvilinear distortion that the convex mirror introduces in the system. The end of the image processing chain is a monochrome video stream, which is used as an information signal to drive the update of the virtual world. The algorithm that interprets this signal as 3D information is again simple – the processed 2D monochrome image is simply interpreted as a 3D surface plot – a technique basic to mathematical function visualisation. Thus, the processed image – which is a scene of the auditorium, with areas of

local motion, caused by audience members, indicated as bright spots – becomes a 3D surface, which spatially bounds the virtual world where the game occurs, where the bright spots are interpreted as elevations (hills/waves) of the surface. Of course, we had to take networking considerations into account in order to implement the system, and provide a link between the image processing segment and the 3D game engine.

Hence, the interaction of an audience member with the world, in this phase of the project, was limited to the possibility of “rising” a wave in the virtual world surface by waving the hand, and moving it throughout the world by walking throughout the playground. The game engine then is based on calculating the interaction between the current state of the surface and the virtual gameplay objects – in this case, a ball as a main actor, moved throughout the world by the waves caused by audience members, and static point objects that should be collected by the ball in the course of its movement. In the end, the whole system is bound by the physical context of audiences – the auditorium is their playground, and all participants face the stage, so the most applicable way to provide feedback is to project the game output video signal to a screen placed on the stage. Due to the fact that the audience participants are both expected to constantly be oriented towards the screen to receive feedback, as well as to have a perception of their physical environment – the auditorium – as a playground and interact with other participants (to increase the collaboration factor of the gameplay) – their perception attention is bound to be divided between the video feedback and the physical environment. In spite of the fact that 3D home gaming suffers from the same limitations of the user having to face the screen, receive a 3D rendering on a 2D screen and using a keyboard/mouse to interact – the fact that the user does not have to move while playing, means that the user does not have to interact with the physical environment – so more perception focus can be allocated for the game feedback, and thus greater immersion can be experienced in this context of a 3D world.

The system can thus not be considered a virtual reality system, since it neither aims to nor can provide means to augment the sensory input of the participants to that level that they would be fully immersed in the computer generated world. The computer generated world in this context is used merely as a map of physical reality, aimed to enhance the link between the physical situation in the playground and the one in the game – so maybe augmented reality would be more applicable as description of these kinds of audience interaction game systems. Nonetheless, the experience we had with the audience shows that total immersion in a computer generated world is not a demand from the audience on beforehand – the audience members that tested the system seemed to have fun and get immersed in the interplay

between their actions in the physical surroundings and the corresponding rendered effects in the 3D world.

In retrospect, we have provided a system that introduces a simple, though novel kind of interaction. The interaction is motion-based and does not require additional tools – though more obvious effects are rendered when there is large contrast between the moving pixels and the background; so the audience was equipped with white gloves as interaction tools. The catadioptric extension to the input video system makes the system scalable, so that it can be implemented in smaller spaces – and one individual can easily interact with the system, as well as a group. We envision that the same concept can be scaled to bigger spaces as well, where the only limit to individual action tracking is the size of the mirror and the camera resolution with which it is filmed. As such there is greater focus on image processing algorithms, in order to obtain a usable input signal for the 3D game engine.

In addition to being scalable, the system is relatively easy and inexpensive to implement – it consists of a network of off-the-shelf hardware: two PCs, a network camera and a network switch (with the spherical mirror being the only item which is a bit more exotic). The system is not difficult to calibrate, as the resolution demanded from it is not that great – for instance, the camera need not be ideally positioned under the mirror center etc; such errors are averaged out in the final feedback, and ca be tolerated. It also uses publicly available software for implementation – although EyesWeb is still free, Virtools is a commercial application. However, as the input interaction processing and 3D world generation is rooted in some very basic mathematical concepts, there is no reason why the front end couldn't be developed in a public domain 3D environment, such as OpenSG.

The network communication goes on through a standard TCP/IP local area network over Ethernet, using the standard client/server architecture implementation via sockets. It provides the distributed computing aspect where each high level task (input video signal acquisition, input video signal processing and 3D game engine) is allocated resources on a separate machines – the network communication then establishes a series processing chain (cascade) between these machines, where each machine propagates its results to the next in the chain via a TCP data stream. The standard networking approach allows separation of application from OS, and allows that the system can be implemented across different OS-es, if such need arises. In addition, the physical level of networking can be easily replaced from wired Ethernet to wireless – which would make the system much easier to physically set up, without much change in the existing code, as we are using the standard OS libraries for

socket connectivity, which then utilizes the OS to interface a socket with a desired network adapter.

This approach to computing, where the high-level tasks are distributed across platforms and machines, allows for easier conceptualisation of further development of the system within each area. For instance, a different image processing algorithm can be utilised in respect to input interaction – for instance, some form of motion tracking. Using high contrast gloves with unique colors as interaction tools, it will be possible to uniquely identify them – thus allowing the system to uniquely identify input from a given audience member. This can be an interesting approach for interaction in duel type games, where two people battle one another. On the other hand, a similar approach could be pursued as in the Cinematrix system – where the audience composes two teams with gloves of different colors – the difference being that here the audience would be encouraged to move across the playground, and forming groups that would give a given constellation of two types of "hills" in the 3D world. Applying some recording and processing of the tracked data, gesture recognition could be developed in a context of a duel game, where for instance a player could initiate a game interaction with the other player (like "casting a spell") with a given uniquely recognizable gesture, which would be an interesting build up to the tradition of using "combos" in duel games through a unique combination of interactive actions – basically, a string or a word composed of interaction letters. (Consider also that in a lot of PC games, the animation that results from performing a "combo" is a render of the avatar performing some sort of a unique gesture).

It is interesting to ponder whether the catadioptric extension might offer something new related to use of cameras as in the Squidball game. There, a system of 27 cameras, uniquely positioned, film a given part of space, and extract the 3D position coordinates of objects (balloons made of special reflective material) that can be uniquely tracked on the image. The standard cameras have a limited field of view, which is probably the reason why such an amount of the cameras is necessary. Catadioptric sensors aim to answer precisely that problem, and it can be easily imagined that a system of two or three catadioptric sensors, positioned on a given constellation on the ceiling as in our prototype, might be able to uniquely identify the 3D space coordinates of a given object. There is some research that points in that direction – consider [47][48][49]. That would of course, allow that interaction gestures could be tracked in three dimensions, and so an eventual gesture recognition would work hopefully for a range of natural movements of the players.

Finally, the demonstration of the prototype that this report described, demonstrated use by a small group of people, which may or may not be considered audience in the sense of the original demands from this project; in addition, it basically deals with a generation of a virtual world, but does not conform to the understanding of immersion in virtual reality, which may or may not be considered as conforming with the theme of the semester, "Virtual Worlds", that this report attempts to address. Still, in perspective, we may say that the future development might answer these demands with a twist – it might provide a basis for a usable game system for small groups of individual players, in a context of augmented reality. Eventually, if the system can be easily scaled to bigger spaces, it might be possible to extend it in to a real audiences context. In any case, this project was a great learning experience – since it relied on quite basic mathematical mappings for the acquisition of input interaction and its effectuation in the 3D world, however it demanded a deeper look into image processing and networking, both from theoretical and practical software perspective for the implementation of those basic mapping; and as such, it represents a fine educational example of cross-modal processing and interpretation.

11 Appendix

This section contains the appendices for this report:

Appendix A. Matlab example codes....73
Appendix B. Difference-based analysis example from EyesWeb....75
Appendix C. Code listing for client connection used in EyesWeb plugin....75
Appendix D. Code listing for client receiving thread used in EyesWeb plugin....77
Appendix E. Code listing for JPEG reading and decoding used in EyesWeb plugin....80
Appendix F. Code listing for server connection used in Virtools plugin....82
Appendix G. Code listing for server receiving used in Virtools plugin....84
Appendix H. Code listing for bitmap transfer used in Virtools plugin....85
Appendix I. Pinch/Spheric Distortion algorithm....87
Appendix J. Persistence effect algorithm....95
Appendix K. Screenshot of EyesWeb patch with plugins....98
Appendix L. Screenshot of Virtools patch with plugins 99

Appendix A. Matlab example codes

Code for circle image generation

```
for x = 1:256
    for y = 1:256
 r = sqrt((x-128)*(x-128) + (y-128)*(y-128));
 if (round(r) == 81)
         G(x, y) = 1;
 else
 G(x, y) = 0;
 end;
    end
end

figure
colormap(gray)
imagesc(G)
```

Code for image generation of $x^2 + y^2$

```
for x = 1:256
    for y = 1:256
        G(x, y) = (x-128)*(x-128) + (y-128)*(y-128);
    end
end
```

```
figure
colormap(gray)
imagesc(G)
```

Code for image generation of a plane in 3D

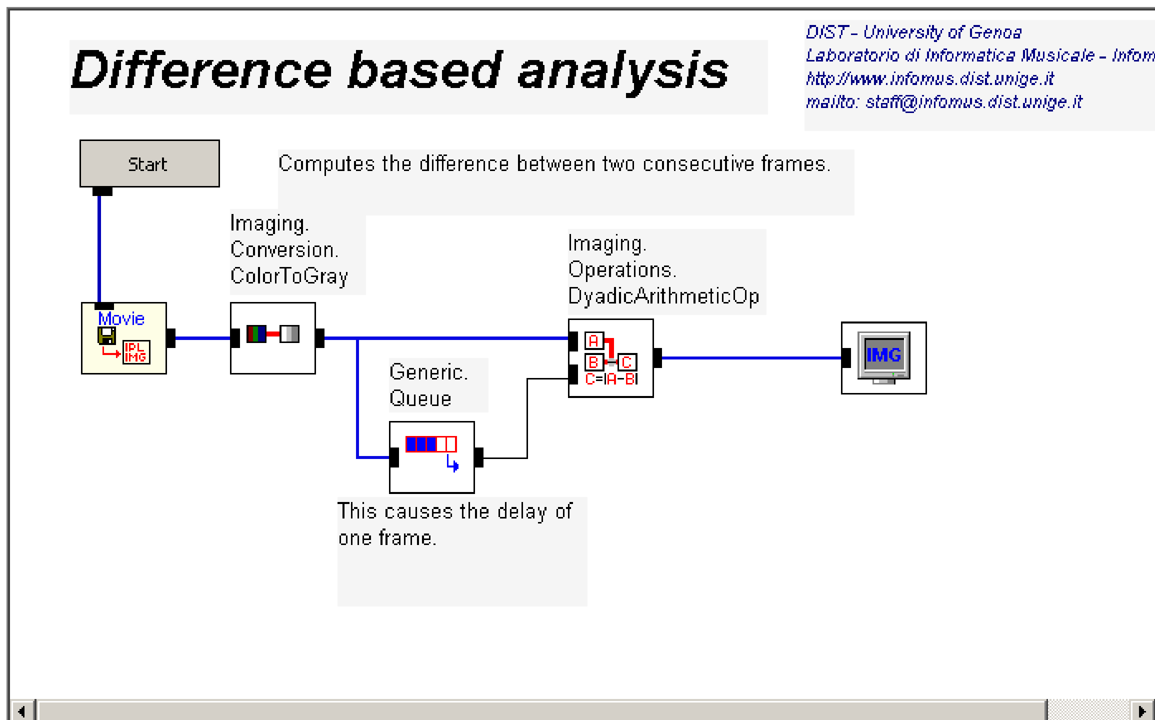
```
% for all input x,y assign the output calue 1.5
% and represent it as a surface

fh = @(x,y) 1.5;
ezmesh(fh,80)
colormap([0 0 1])
```

Code for difference based example

```
A=[0 0 0; 0 25 50; 0 0 0]
B=[0 0 0; 25 50 0; 0 0 0]
C = abs(A-B)
D = A - A

figure
colormap(gray)

subplot(2,5,1);image(A)
subplot(2,5,2);image(A)
subplot(2,5,3);image(B)
subplot(2,5,4);image(B)
subplot(2,5,5);image(B)
subplot(2,5,7);image(D)
subplot(2,5,8);image(C)
subplot(2,5,9);image(D)
subplot(2,5,10);image(D)
```

Appendix B. Difference-based analysis example from EyesWeb

The basic set-up for difference-base analysis is provided as an example file in the EyesWeb 3.3.0 installation:



The libraries and blocks used are noted in the image. The Generic.Queue object causes a delay of one frame. Its output, which represents the previous frame, and the current frame, get subtracted in the Imaging.Operations.DyadicArithmeticOp block. The output of this block represents the difference based output.

Appendix C. Code listing for client connection used in EyesWeb plugin

```
int CTestImgBlock::SockClientStart(char *host,int in_port)
{
    //EyesWeb demands this
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));

    //the input arguments are: (entered by user in the EyesWeb patch GUI)
    // host - a string having the IP adress of the server,
```

```
//      either as a DNS entry "www.yahoo.com"
//      or as a local adress "192.168.x.x"
// port - integer specifying the server port number, which for
//      HTTP is 80

//This will tell windows to access the WinSock subsystem
//and sets the version to use - performed elsewhere
//WSADATA wsa_data;
//WSAStartup(MAKEWORD(1,1), &wsa_data);

int r; //connection result
int nodelay=1; //var for socket options

//variable to determine whether we are using DNS names "www.yahoo.com"
//or local adresses "192.168.x.x"
bool isUsingName;

//create client socket
client_socket = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

//check if client socket is created properly
if(client_socket == INVALID_SOCKET)
{
    //something is wrong - return error
    return 0;
}

//port number
port = in_port;

//initialize structure that keeps the server adress information
memset((void *)&server,0,sizeof(server));

//try resolving adress once, to see if we are using
//local or DNS adress
server.sin_addr.s_addr=inet_addr(host);

if(server.sin_addr.s_addr==INADDR_NONE)
{
    //we have used a DNS name
    isUsingName = true;
    h = gethostbyname(host);
    if(h==NULL)
    {
        //something is wrong - return error
        Message("SOCKET : gethostbyname failed: error#:%ld\n", WSAGetLastError());
        return 0;
    }
}
else
{
    //we have used a local adress
    isUsingName = false;
}

//knowing whether the name is local or DNS, resolve the adress for real
if(isUsingName)
{
    server.sin_family=AF_INET;
    server.sin_addr=*((in_addr*)*h->h_addr_list);
    server.sin_port=htons(port);
}
else
{
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr( host );
    server.sin_port = htons( port );
}

//connect the socket
r = connect(client_socket,(sockaddr *)&server,sizeof(struct sockaddr));
```

```
    //check if connection is OK
    if(r == SOCKET_ERROR)
    {
        //Message("SOCKET : error connecting socket#:%ld\n", WSAGetLastError());
        closesocket(client_socket);
        return 0;
    }

    //connection was OK, set socket options
    setsockopt(client_socket, IPPROTO_TCP, TCP_NODELAY, (char*)&nodelay, sizeof(nodelay));

    //connection succesful, exit
    return 1;
}
```

Appendix D. Code listing for client receiving thread used in EyesWeb plugin

```
UINT CTestImgBlock::ThreadProcess (LPVOID param)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));

    //cbuffer is the array where the data packet from each receive call is stored
    //fullbuffer is the first buffer
    //transferbuffer_raw is the second one
    //declared as class properties, so accessed through the help variable

    //INITIALISATION PRE LOOP

    //help var to access class variables
    THREADSTRUCT*    ts = (THREADSTRUCT*)param;

    ts->_this->fullbuffer = (char*) calloc(1, sizeof(char) );

    int len = 1;
    int startoffset = 0;
    bool isFrameOK = true;

    //pointers to calculate slices
    char *name_ptr;
    char *val_ptr;
    char *ret_ptr;

    //START LOOP
    while (ts->_this->isThreadRunning)
    {
        //wait
        Sleep(50);

        // *** execute a receive call **
        len = recv( ts->_this->client_socket, ts->_this->cbuffer, 8192, 0 );

        if (len > 0)
        {
            //since it is going to loop forever and stream,
            // the end of jpg frame is not the end of packet (len = 0)
            // must work here through tags

            //check for boundary= first
            if (ts->_this->c_boundary == NULL)
            {
            name_ptr = strstr(ts->_this->cbuffer, "boundary=");
```

```
            if ( name_ptr == NULL )
                {
                ts->_this->Message_Error("boundary= not found in %s \n", (unsigned char*)
ts->_this->cbuffer);
                break;
                }
            else
                {
                //boundary found
                //search for closing enter and extract value
                int sublength;
                val_ptr = name_ptr + strlen("boundary=");

                ret_ptr = strstr(val_ptr, "\r\n");
                sublength = abs(ret_ptr - val_ptr);
                ts->_this->c_boundary = (char *)calloc( sublength+1, sizeof( char ) );
                strncpy(ts->_this->c_boundary, val_ptr, sublength);

                startoffset = abs(ret_ptr - &ts->_this->cbuffer[0]);
                }
            } //end if cboundary empty

            //expecting that content boundary will be found in the first packet, so no
checks
            //look for boundary in cbuffer with - if found it is start of a new
            //fullbuffer, if not, probably it is filling the previous fullbuffer
            bool firstTime = false;

            int trylen = len - startoffset;
            name_ptr = (char *) memmem(ts->_this->cbuffer + startoffset, trylen, ts->_this-
>c_boundary, strlen(ts->_this->c_boundary));

            if ( name_ptr != NULL )
            {
                //we got ourselves a new fullbuffer to fill or start the initial one
                //copy the previous (if any) fullbuffer to transferbuffer_raw

                //erase and repopulate transfer, unless it is initial
                //(content_length not yet init)
                //check if initial
                int cl_old = ts->_this->content_length;
                if (ts->_this->content_length == NULL)
                {
                    firstTime = true;
                }

                //if we found boundary, then new ContentLength is here as well
                //look for content length and copy
                //only jpeg data (no header) into transfer
                char *ContLength_ptr;
                char *NextSlice_ptr;

                ContLength_ptr = strstr(name_ptr, "Content-Length:");
                if ( ContLength_ptr == NULL )
                {
                    //content length not found
                    //broken frame - can be content length is in next frame?
                    isFrameOK = false;

                    ts->_this->Message("BAD FRAME: Content-Length not found in %s \n",
name_ptr);

                    ts->_this->content_length = 0; // should reinit at the next frame
                    startoffset = 0;
                    NextSlice_ptr = name_ptr;
                }
                else
                {
                    //content length found
                    //search for closing enter and extract value
                    char *num;
```

```
                    int sublength;
                    val_ptr = ContLength_ptr + strlen("Content-Length: ");

                    ret_ptr = strstr(val_ptr, "\r\n");
                    sublength = abs(ret_ptr - val_ptr);
                    num = (char *)calloc( sublength+1, sizeof( char ) );
                    strncpy(num, val_ptr, sublength);

                    ts->_this->content_length = atoi(num);

                    //must reinit here, it aint firstTime anymore
                    startoffset = 0;
                    free(num);
                    NextSlice_ptr = ret_ptr + 4;
                    isFrameOK = true;
                } //end if (else) ContLength_ptr == NULL

                    if (!(firstTime))
                    {
                    //here is the full packet - the jpg requested is received
                    // see the content length and copy only
                    //jpeg data (no header) into transfer

                    //not firstTime, meaning it may be sliced.
                    //slice 1 - close fullbuffer and toss the part
                    //before myboundary into fullbuffer

                    int sliceprev_length = abs(name_ptr - &ts->_this->cbuffer[0])-2;
                    size_t size = _msize( ts->_this->fullbuffer );

                    //correct negative sliceprev
                    if (sliceprev_length < 0)
                    {
                        //note that if sliceprev_length was -3 (thus now 0)
                        //we should truncate abs(sliceprev_length) chars
                        sliceprev_length = abs(sliceprev_length);
                        ts->_this->fullbuffer = (char*) realloc( ts->_this->fullbuffer,
size - (sliceprev_length * sizeof( char )) );

                    }
                    else
                    {
                        ts->_this->fullbuffer = (char*) realloc( ts->_this->fullbuffer,
size + (sliceprev_length * sizeof( char )) );
                        memcpy( ts->_this->fullbuffer + size - 1, ts->_this->cbuffer,
sliceprev_length * sizeof( char ) );
                    }

                    size = _msize( ts->_this->fullbuffer );

                    free( ts->_this->transferbuffer_raw );
                    ts->_this->transferbuffer_raw  = (char*) calloc( 1, sizeof(char) );
                    ts->_this->transferbuffer_raw = (char*) realloc( ts->_this-
>transferbuffer_raw, (size * sizeof( char )) );

                    //transfer the fullbuffer (first) into transbuffer_raw (second)
                    memcpy( ts->_this->transferbuffer_raw, ts->_this->fullbuffer,  size *
sizeof( char ));
                    free( ts->_this->fullbuffer );

                    //this is the point where we got a new image in transferbuffer
                    //done with frame transfer. Now call Execute, which should
                    // in turn call TransferBitmap, which should
                    // copy the data to a usable bitmap pixel data
                    if(isFrameOK)
                        {
                        ts->_this->haveTransferImage = 1;
                        //setting the event calls Execute
                        SetEvent(ts->_this->hEvent);
                        }
```

```
                    } //end if (not) firstTime

                //reinit - erase fullbuffer. Data in fullbuffer
                //starts with first content byte
                ts->_this->fullbuffer  = (char*) calloc( 1, sizeof(char) );

                //slice 2 - after fullbuffer is reinited.
                int slicenext_offset = NextSlice_ptr - &ts->_this->cbuffer[0];
                int slicenext_length = len - slicenext_offset;
                size_t size = _msize( ts->_this->fullbuffer );
                ts->_this->fullbuffer = (char*) realloc( ts->_this->fullbuffer, size +
(slicenext_length * sizeof( char )) );
                memcpy( ts->_this->fullbuffer + size - 1, NextSlice_ptr, slicenext_length *
sizeof( char ) );

                //OK Done.
                free(ContLength_ptr);
                free(NextSlice_ptr);

            } //end if ( name_ptr != NULL ) where name_ptr = mybondary
            else
            {
                //receiving a regular slice
                //simply add all content to fullbuffer
                size_t size = _msize( ts->_this->fullbuffer );

                ts->_this->fullbuffer = (char*) realloc( ts->_this->fullbuffer, size + (len
* sizeof( char )) );
                memcpy( ts->_this->fullbuffer + size - 1, ts->_this->cbuffer, len * sizeof(
char ) );

            } //end if (else) ( name_ptr != NULL ) where name_ptr = mybondary

        } //end if len != 0 - meaning a packet frame received

    }   //END LOOP

    AfxEndThread(0);
    return 0;
}
```

Appendix E. Code listing for JPEG reading and decoding used in EyesWeb plugin

```
void CTestImgBlock::TransferBitmap( int inWidth, int inHeight )
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));

    //variables for reading the memory stream
    HGLOBAL hMem = NULL;
    LPVOID lpData = NULL;
    IStream *pstm = NULL;

    //var that will hold the Windows image
    LPPICTURE pPicture;

    size_t content_length_loc = _msize( transferbuffer_raw );

    //alocate memory on global
    if ( ( hMem = GlobalAlloc( GMEM_MOVEABLE, content_length_loc) ) != NULL)
    {
        //read file content in global allocated memory
        if ( ( lpData = GlobalLock( hMem)) != NULL)
```

```
        {
            memcpy (lpData, transferbuffer_raw, content_length_loc);
            GlobalUnlock( hMem );
        }
    }

    //create stream from allocated memory
    HRESULT hr = CreateStreamOnHGlobal( hMem, TRUE, &pstm);

    if(SUCCEEDED(hr) )
    {
        //create picture from stream
        int ctl = (int) content_length_loc;

        //read and decode JPEG from memory, put into pPicture
        hr = OleLoadPicture(pstm, ctl, FALSE, IID_IPicture, (void**)&pPicture);

        if ( FAILED( hr))
        {
            Message("OleLoadPicture failed \n");
            pPicture = NULL;

        }
        pstm->Release();

        if (pPicture != NULL)
        {
            isPicOK = true;
            short sType;
            hr = pPicture->get_Type(&sType);

            if (SUCCEEDED(hr))
            {

                if (sType == PICTYPE_BITMAP)
                {
                    //transfer to windows bitmap

                    HBITMAP hBitmap;
                    hr = pPicture->get_Handle((OLE_HANDLE *) &hBitmap);

                    if (SUCCEEDED(hr) && hBitmap)
                    {

                        BITMAP bm;
                        GetObject( hBitmap, sizeof(BITMAP), &bm );
                        //copy to a char* buffer called laBuff
                        //when done, laBuff will be assigned to the
                        //plugin output image, called out, of type IplImage*
                        //out->imageData = out->imageDataOrigin = laBuff;
                        memcpy(laBuff, (char *)bm.bmBits, inWidth*inHeight*3);
                    }

                free(hBitmap);

                }
            }

        }
        else
        {
        Message("TROUBLE WITH PIC\n");
        isPicOK = false;
        } // end else pPicture NULL
    }


    //free global alocated memory
    if ( hMem) GlobalFree( hMem);
    if (pPicture) pPicture->Release();
```

```
    haveTransferImage = 0;
}
```

Appendix F. Code listing for server connection used in Virtools plugin

```
sock_server_struct * sock_server_start(int port, int timeout, CKContext *m_Context)
{

    //we have two sockets here, put in one structure srv
    //srv->s is the socket which listens to incoming connections
    //srv->t is the socket for data exhange, once a connection from
    //      client has been made

    int i;
    int nodelay=1;
    unsigned long t=0;
    sock_server_struct *srv=0;
    char initstr[]="KnP\0";

    // Create a socket structure
    srv = (sock_server_struct *) calloc(1,sizeof(sock_server_struct));
    if(srv==0) return 0;

    srv->s = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
    if(srv->s==INVALID_SOCKET)
    {
        free(srv);
        return 0;
    }

    //set socket options for s
    setsockopt(srv->s, SOL_SOCKET, SO_REUSEADDR, (char*)&nodelay, sizeof(nodelay));

    // Bind the socket.
    // port is entered through the Virtools GUI, passed as parameter
    srv->port=port;
    srv->slen = sizeof(srv->r_addr);

    memset((void*)&srv->addr,0,sizeof(srv->addr));
    srv->addr.sin_family=AF_INET;
    //set the server adress to own computer - 127.0.0.1 works too
    srv->addr.sin_addr.s_addr = inet_addr( "0.0.0.0" );
    srv->addr.sin_port=htons(srv->port);

    //bind the socket
    i=bind(srv->s,(sockaddr *)&srv->addr,sizeof(srv->addr));
    if(i==SOCKET_ERROR)
    {
        closesocket(srv->s);
        free(srv);
        return 0;
    }

   // Listen on the socket for incoming connections
    i=listen(srv->s,SOMAXCONN);
    if(i==SOCKET_ERROR)
    {
        closesocket(srv->s);
        free(srv);
        return 0;
    }
```

```
    //ioctlsocket disables the blocking mode
    //when FIONBIO is set to 1
    //so it wont loop when buffer empty
    t=1;
    ioctlsocket(srv->s,FIONBIO,&t);

    //Loop and accept connections.
    //if non blocking, the loop will exit
    //must have another socket here - the t, which
    //will accept the connection that has come on s.
    //timeout is usually 10 seconds wait time for client

    do
    {
        //try to accept a connection
        srv->t=accept(srv->s,(sockaddr *)&srv->r_addr,&srv->slen);
        //wait
        timeout--;
        Sleep(1000); //50 is too short, must be 1000 ms
    } while (srv->t==INVALID_SOCKET && timeout >0);

     //loop has ended, check if we have a connection
    if (srv->t==INVALID_SOCKET)
    {
        //no connection, exit with error
        closesocket(srv->s);
        free(srv);
        return 0;
    }

    // passed the if, have a connection
    //set t to non blocking as well
    t=1;
    ioctlsocket(srv->t,FIONBIO,&t);

    //close srv.s - that was only to accept the connection, then
    //communication goes on through srv.t
    closesocket(srv->s);

    //set socket options for t
    setsockopt(srv->t, IPPROTO_TCP, TCP_NODELAY, (char*)&nodelay, sizeof(nodelay));

    //set a buffer size for the expected 16384 bytes, plus
    //4 bytes header that SendToNetwork from EyesWeb adds
    int buflen = sizeof(char[16388]);
    srv->buflen=buflen;

    srv->buffer=calloc(1,srv->buflen);

    //receive the EyesWeb signature sent upon connection
    recv(srv->t,(char *)&srv->buffer,20,0);
    recv(srv->t,(char *)&srv->buffer,36,0);

    //connection initialised, end and return the sockets
    return srv;
}
```

Appendix G. Code listing for server receiving used in Virtools plugin

The server receiving is initiated in the PreProcess method of the network manager. All received bytes are stored in the `inCharP_serv` buffer (char array).

```
//Will be run before the frame is executed and rendered and forms the main execution loop
CKERROR VNetworkMan::PreProcess()
{
    //problem with netmode when %s must use %d
    //m_Context->OutputToConsoleEx("PreProcess: %d, client: %d, server %d ", netmode,
client, server );

    switch (netmode)
    {
    ...

        case 2:
        {
            //socket mode - client or server(list.)
            //client and server can be concurrent = do not switch
            if (server)
                {
                if (srv_on)
                    {
                    //run a receive call
                    inCharP_serv = (char *)sock_server_getdata(srv, timeout, m_Context);
                    }
                }

    ...
            framecount++;
            break;
        }

    }

    return CK_OK;
}
```

Which in turn calls the following function:

```
char *sock_server_getdata(sock_server_struct *srv, int o, CKContext *m_Context)
{

    int len;
    len = 1;

    //try to receive 16388 bytes
    len = recv( srv->t, srv->cbuffer, sizeof(srv->cbuffer), 0 );
    srv->buflen = len;

    if ( ((len == -1)) || len == 0 || len == WSAECONNRESET || len == WSAEWOULDBLOCK )
    {
        //no packet from EyesWeb client, erase the buffer and return
        strcpy(srv->cbuffer,"");
        return srv->cbuffer;
```

```
    }

    //something was received in the buffer - return it
    return srv->cbuffer;
}
```

Appendix H. Code listing for bitmap transfer used in Virtools plugin

The bytes received in the `inCharP_serv` buffer are applied as grayscale colors of a 128 x 128 RGB texture in Virtools.

```
int NS_EYWServer(const CKBehaviorContext& BehContext)
{
    CKBehavior *beh = BehContext.Behavior;

    beh->ActivateInput(0, FALSE);
    beh->ActivateOutput(0);


    VNetworkMan *man = (VNetworkMan*)BehContext.Context->GetManagerByGuid( VNETWORK_GUID );
    if (!man)
    {
        BehContext.Context->OutputToConsoleEx("Can't get the Network Manager");
        return CKBR_GENERICERROR;
    }

    if (man->VN_IsServSocketSetOn() )
    {

        if (man->srv->buflen == 16388 )
        {

            //tex = the Virtools texture we are changing
            CKTexture* tex = (CKTexture*)beh->GetTarget();
            if (!tex) return CKBR_OWNERERROR;

            VxColor col; //color variable
            VxImageDescEx TexDesc; //texture description var
            tex->GetSystemTextureDesc(TexDesc);
            DWORD* ptr=(DWORD*)tex->LockSurfacePtr();
            DWORD dcol=RGBAFTOCOLOR(&col);

            //set texture pixels according to received packet
            if (ptr && (TexDesc.BitsPerPixel==32))
            {
                //internal counter for reading index
                int j = 0;

                //arow, acol - row and column counters
                for (int arow=0;arow<128;arow++)
                {
                    for (int acol=0;acol<128;acol++)
                    {

                        //read a pixel value
                        unsigned char i = (unsigned char) *(man->inCharP_serv+j);

                        // color needs floats from 0 to 1 = normalize
                        // i must be float to begin with
                        float ficol = ((float)i ) / 256;
```

```
                        //the received pixel value is grayscale - ficol
                        //however texture pixels are RGB
                        //copy the grayscale value to r,g and b
                        col.r = ficol;
                        col.g = ficol;
                        col.b = ficol;
                        //col.a = 0.5f; // not necessarry to set alpha

                        //get color as dword
                        dcol=RGBAFTOCOLOR(&col);

                        //replace the pixel in the texture with the current
                        // color value
                        SetPixelReplace(&ptr[16384 - 128*arow - (128-acol)],dcol);
                        j++;
                    }
                }
            }
            tex->ReleaseSurfacePtr();
        }
    }

    return CKBR_OK;
}
```

Appendix I. Pinch/Spheric Distortion algorithm

The Pinch/Spheric (barrel/pincushion) distortion algorithm works in the domain of polar coordinates. We have discussed a definition for an image function $f(x, y)$ that assigns brightness to a set of (x, y) points. The spatial coordinates x and y, are of course Cartesian. The same image can be described in a polar coordinate system, which gives a set of (ρ, θ) points. A point (x, y) in a Cartesian coordinate system, will have (ρ, θ) coordinates in the radial coordinate system, where in general

$$\rho = \sqrt{x^2 + y^2}$$
$$\theta = arctg \frac{y}{x} \qquad \textit{Eq 14}$$

The coordinate ρ (sometimes written as r) represents a radial coordinate, which is the distance from the point to the center of the coordinate center (and in this case, the image). All of the points on a circle with radius ρ from the center of the image will have the same ρ coordinate (just as all points on a vertical line in a Cartesian coordinate system share the same x coordinate). They can be uniquely distinguished by the second, angular coordinate θ – which represents the angle that is formed by the chosen point, the coordinate origin and a point on a reference axis (usually by convention the Cartesian x axis)

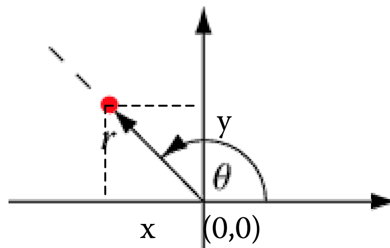


Figure 36. Illustration of polar coordinates, with added Cartesian ones (from Ref.[45])

We have obtained the starting theoretical considerations and Matlab code from Ref. [7]. This code was then translated and ported to C++ and used it as a function in the EyesWeb plugin developed for the purpose of distortion correction. The figures below display the transformation function, which is the base for this algorithm:

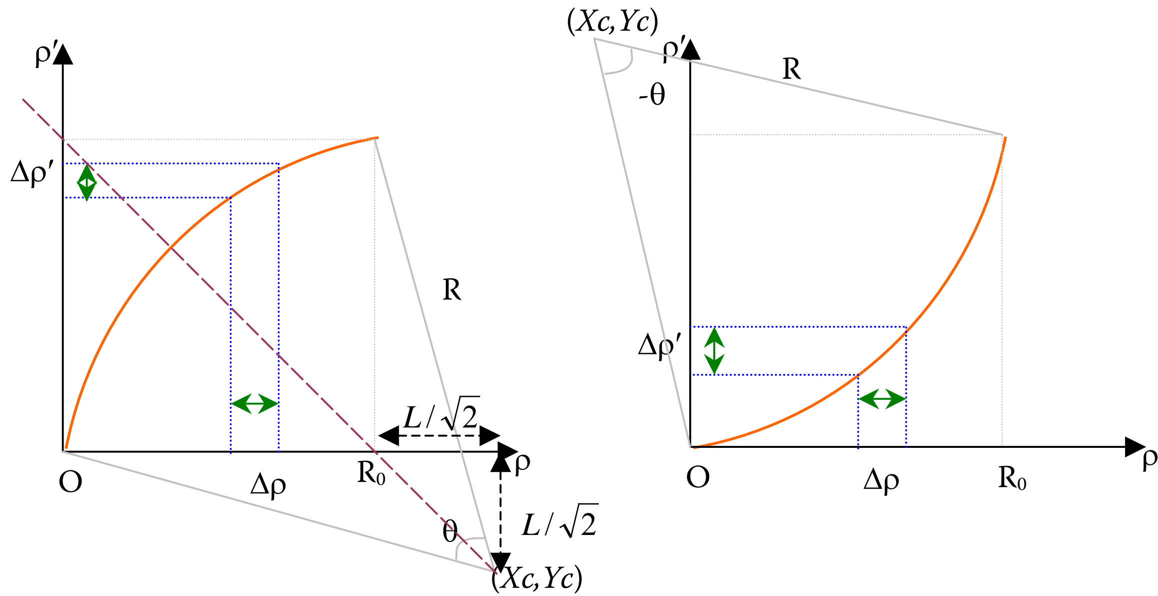


Figure 37. Transformation functions of a) Spherize and b) Pinch filter (From Ref.[7])

In both images, the ρ axis represents the radial coordinate of the input, undistorted image, and the ρ' axis represents the radial coordinate of the distorted output image. Obviously, a point with a coordinate ρ in the input image will be mapped to a (usually different) coordinate ρ' in the output image – thus a circle with radius ρ in the input image will be mapped to a circle with radius ρ' in the output image, either bigger or smaller than the original one. The image size is represented through R_0, which represents the maximum ρ for that image – that is the radius of the largest circle that can be inscribed within the bounds of the image.

The curvature of the transformation function determines the strength of the distortion – and this curve can be seen as an arc of a circle with radius R, with center in (Xc,Yc). The starting point (0,0) and the end point (R_0, R_0) must remain fixed, so it is obvious that the arc can be circular only if the center of the circle remains collinear with (0, R_0) and (R_0, 0), as shown on Figure 37 a). Assuming that the center (Xc,Yc) can be dragged to infinity along this line – in this case the two end radii R of the arc will be parallel, the angle between them θ (not the angular polar coordinate, since the transformation function only converts one radial polar coordinate into another) in this extreme case will be zero. This is the case where the transformation function is a straight line (arc of a circle with infinitely large radius cut with infinitely small angle) and $\rho = \rho'$, hence no transformation. As the center (Xc,Yc) approaches the point (R_0, 0), the angle θ increases and so does the curvature of the function, and thus the

degree of distortion. When (Xc,Yc) reaches the point ($R_0/2$, $R_0/2$), the angle $\theta = \pi$, and further progression in the same direction changes the sign of the distortion – so the barrel version flips into the pincushion version. Obvously, the degree and sign of distortion is determined by the distance of (Xc,Yc) from ($R_0/2$, $R_0/2$); or which is the same, by the angle θ, that we can also use as the parameter for distortion.

Eventually, we need the transformation in algebraic format, such that we can calculate ρ' from ρ and some constant parameters that will describe the distortion. It can be shown that the following mapping is the needed transformation for the barrel (spheric) distortion case:

$$\rho' = Y_C + \sqrt{R^2 - (\rho - X_C)^2} \qquad \textit{Eq 15}$$

where

$$R = R_0 / \left(\sqrt{2}\sin(\theta/2)\right) \qquad \textit{Eq 16}$$

$$Xc = \frac{1}{2}\left[1 + ctg(\frac{\theta}{2})\right]R_0 \qquad Yc = \frac{1}{2}\left[1 - ctg(\frac{\theta}{2})\right]R_0 \qquad \textit{Eq 17}$$

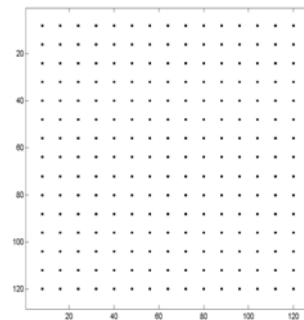
$$\rho = \sqrt{x^2 + y^2} \qquad \rho' = \sqrt{x'^2 + y'^2} \qquad \textit{Eq 18}$$

The transformation finally becomes a transformation matrix in the example Matlab code. Parts of the original code from [7] are included here, with some comments. The usage of variable names is mostly similar to the notation used up to now. Instead of using θ as distortion parameter and entering angles in radians, a normalized parameter gamma is introduced, with values allowable values from –1 to 1, which is then used to calculate θ.

The program first generates and keeps the (x,y) coordinates of 31 x 31 points in two matrices, which originally are ordered in linear fashion. Those are represented on a 128x128 pixel image, so that they represent the original undistorted rectilinear pegboard / grid. Then, the ρ coordinate for the original set of 31 x 31 are calculated and stored in a matrix, which is further used to calculate the transformation matrix T according to Eq 15 (its elements would correspond to ρ'). Finally, the distorted coordinates (x’,y’) of the original set of 31 x 31 points

are obtained in matrices XT and YT – by multiplying the matrices that contain the original (x,y) coordinates of the 31 x 31 points, with the matrix T. The distortion is afterwards though annulled for all points that satisfy $\rho > X_C$. Finally, the distorted coordinates of the 31x31 set of points is used to generate a distorted output image. Here is the Matlab code listing for this program:

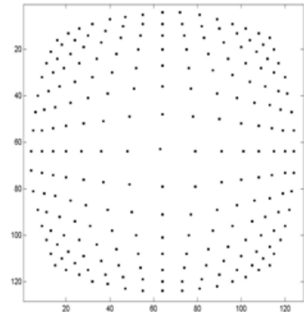
```
% distortion parameter
Gamma=0.5

% size of image matrix N x N
N=128;

% size of sampling interval for dots that represent the image
NInterval=4;

% I is 128X128 matrix of ones - represents light background
I=ones(N,N);

% the image against the background is formed of dark dots
% NI is 31 for N=128 NInterval=4 - the size of the matrix
% which will hold the coordinates of these dots for the image
NI=round(N/NInterval-1);

% X and Y are also 31X31 matrices
% hold the x and y coordinates of the 31 x 31 dark dots
[X,Y]=meshgrid(1:1:NI,1:1:NI);

% with this scaling, the 31 x 31 dark dots
% are evenly distributed across the 128 x 128 background
X=X*NInterval;
Y=Y*NInterval;

% the coordinate of point (15,15) from the 31x31 matrix
% x coordinate is X(15, 15) and y coordinat is Y(15, 15)
% this command sets the brightness of the points in I that have
% the same coordinates as the 31x31 data to zero, obtaining
% the dark dots on the bright background - undistorted input image
I((X-1)*N+Y)=0;


% display image I
figure;
imagesc(I);
colormap('gray(256)');
```



```
% R0 is a radius of a circle inscribed in I
% radius of the distorted area -
% upper limit of the possible rho coordinate for the image I with size NxN
R0=(N-1)/2;

% CtrX and CtrY are coordinates of the center
% of the image I - also the center of visible distortion in image
CtrX=(1+N)/2;
CtrY=(1+N)/2;
```

```
% the angle Theta is related to the distortion argument Gamma
Theta=abs(Gamma*pi);

% help parameter
L=(1/tan(Theta/2)-1)*R0/2^0.5;
% radius of spheric curvature in the transfer function coordinate system
R=R0/2^0.5/sin(Theta/2);

% Xc, Yc - coordinates of origin of angle theta in the transfer function
coordinate system
% (origin of spheric curvature)
Yc=R0+L/2^0.5;
Xc=-L/2^0.5;

% r is 31 x 31 matrix - keeps the distances from the
% center of image from each dark dot
% this is the original rho coordinate
r=sqrt((X-CtrX).^2+(Y-CtrY).^2);

If (Gamma>1|Gamma<-1) error('wrong Gamma, out of boundary [-1 +1]');

 elseif (Gamma>0)  % Convex - Spheric

   % T is 31x31 matrix - transformation matrix
   % it is the formula for rho'
   T=abs(Yc+sqrt(R^2-(r-Xc).^2))./r;

   % XT YT - 31x31 matrices holding the x and y coordinates of
   % the transformed / distorted 31x31 points
   % this is the actual transformation
   XT=round(T.*(X-CtrX)+CtrX);
   YT=round(T.*(Y-CtrY)+CtrY);

   % find which of the 31x31 points have r (rather rho) > Xc and store their
   % coords in ry, cx - for those no transformation
   [ry,cx]=find(r>Xc);

   % reset the transformation for the points with rho > Xc
   % so they are back to their original state - replace the
   % coordinates in the transformed coordinate matrices with the original ones
   XT(ry+(cx-1)*NI)=X(ry+(cx-1)*NI);
   YT(ry+(cx-1)*NI)=Y(ry+(cx-1)*NI);
   T(ry+(cx-1)*NI)=1;

   % display only the transformation matrix as an surface plot
   % figure;
   % surf(T.*r);

elseif (Gamma==0)
   % for Gamma 0 there is no change
   Sign='REMAIN';
   XT=X;
   YT=Y;

  …

end

% IT - the transformed image matrix
% initialise at ones
IT=ones(N,N);

% Generate the distorted image knowing the distorted dot coordinates
% IT at transformed coordinates will have
```

```
% the same values (dark - 0) as the I at original coords
IT(YT+(XT-1)*N)=I(Y+(X-1)*N);

% display the distorted image
figure;
imagesc(IT);
colormap('gray(256)');
```



This code was ported to C++ and ended up in the EyesWeb distortion plugin as the function named cvPinchSpheric. Possibilities were added to change the position and the radius of the visible distortion on the image from within EyesWeb and with usage of GUI elements, such as faders – which greatly helped with the calibration of the plugin, since it was not possible to precisely position the camera lens to be exactly under the spherical mirror center. The basic calculation is the same – in some case, the variables have even the same name as in the Matlab example. However, instead of most of the matrix operations, there a nested for-loop which iterates through all the input pixels, does the calculation (so there is no T matrix, but instead a t variable), and populates the matrices that hold the distorted (x,y) coordinates of points – so called displacement maps. As soon as these are available, an OpenCV function called cvRemap is called, which can generate an output image based on an input image and x and y maps, and thus is the filtered output provided.

```
void CTestImgBlock::cvPinchSpheric( const CvArr* srcarr, CvArr* dstarr, int flags )
{
    //function arguments
    //IplImage* in. IplImage* out from plugin are passed as
    //CvArr* srcarr, CvArr* dstarr
    //flags is remapping flags for OpenCV remap

    //entered through EyesWeb GUI and are class vars:
    //gamma is the distortion parameter
    //offset_x, offset_y - offset for the center of visible distortion
    //offset_radius - offset for the radius of the visible distortion

    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));

    //matrices for mapping coordinates
    //correspondant to Xt and Yt in Matlab example
     CvMat* mapx = 0;
     CvMat* mapy = 0;

     //Center point
     CvPoint2D32f Ctr;

     double R_zero = 0;
     double Theta;
     double L = 0;
     double R = 0;
     double Xc = 0;
     double Yc = 0;

    double sqrt_2 = sqrt((double)2);
```

```
//integer for check whether Gamma is positive
int isGammaPos;

CV_FUNCNAME( "cvPinchSpheric" );

__BEGIN__;

//initialize matrices as OpenCV demands
CvMat srcstub, *src = (CvMat*)srcarr;
CvMat dststub, *dst = (CvMat*)dstarr;
CvSize ssize, dsize;

CV_CALL( src = cvGetMat( srcarr, &srcstub ));
CV_CALL( dst = cvGetMat( dstarr, &dststub ));

if( !CV_ARE_TYPES_EQ( src, dst ))
    CV_ERROR( CV_StsUnmatchedFormats, "" );

if( M <= 0 )
    CV_ERROR( CV_StsOutOfRange, "M should be >0" );

//get the sizes of source and destination
//(input and output) matrices - should be same
ssize = cvGetMatSize(src);
dsize = cvGetMatSize(dst);


//decide whether it is pinch or spheric
if (gamma > 0) isGammaPos = 1;
    else isGammaPos = -1;


//same calculations as in Matlab example

//calculate R0
R_zero = dsize.height/2 + offset_radius; //dsize.width/2;

//calculate center point
Ctr = cvPoint2D32f(dsize.width/2+offset_x,dsize.height/2+offset_y);

//calculate theta
Theta = abs(gamma * CV_PI );

//sqrt(2) trick for cancelling after for Xc Yc - L is help var for them
L = (1/tan(Theta/2)-1)*R_zero/sqrt_2;

//calculate R
R = R_zero/sqrt_2/sin(Theta/2);

//calculate Xc Yc depending on pinch or spheric
if (isGammaPos > 0)
    {
    Xc = R_zero+L/sqrt_2;
    Yc = -L/sqrt_2;
    }
else
    {
    Xc = -L/sqrt_2;
    Yc = R_zero+L/sqrt_2;
    }

//initialize the distorted coordinate matrices mapx = XT, mapy = YT
CV_CALL( mapx = cvCreateMat( dsize.height, dsize.width, CV_32F ));
CV_CALL( mapy = cvCreateMat( dsize.height, dsize.width, CV_32F ));

//iterate through all pixels in the input / output image
int c_y, c_x;
for( c_y = 0; c_y < dsize.height; c_y++ )
{
    //row pointers to the distorted coordinate matrices
```

```
        float* mx_row_pointer = (float*)(mapx->data.ptr + c_y*mapx->step);
        float* my_row_pointer = (float*)(mapy->data.ptr + c_y*mapy->step);

        for( c_x = 0; c_x < dsize.width; c_x++ )
        {
            //calculate rho coordinate for a given point (c_x, c_y)
            // in the image
            double x_loc = c_x - Ctr.x;
            double y_loc = c_y - Ctr.y;

            double rho_res = sqrt((x_loc*x_loc) + (y_loc*y_loc));
            double r_rel = rho_res - Xc;

            // transformation parameter t_res instead of a transform matrix
            // since we can generate the mapping matrices with distorted
            // coordinates here in the loop
            double t_res;

            if (gamma != 0)
            {
                //formula for rho' but divided with rho_res
                t_res = abs( Yc + isGammaPos*sqrt(abs(R*R - r_rel*r_rel)) )/rho_res;
            }
            else
            {
                //here L and R are infinity! so the same image out
                t_res = 1;
            };

            // calculate the new cordinates where the color of
            // this pixel (c_x, c_y) of the input image should appear
            // in the distorted image
            double x_t = abs(t_res*(c_x - Ctr.x) + Ctr.x);
            double y_t = abs(t_res*(c_y - Ctr.y) + Ctr.y);

            // write the "distorted" coordinates in the
            // matrices for mapping coordinates (distorted)
            // cast to int first for lowering of the precision -
            // crash otherwise for rho_res > Xc
            // - that check is not performed here
            mx_row_pointer[c_x] = (float)(int)x_t;
            my_row_pointer[c_x] = (float)(int)y_t;

        }
    }

    // remap the original image src using the mapx and mapy matrices
    // into the distorted output image dst
    // eqiuvalent to IT(YT+(XT-1)*N)=I(Y+(X-1)*N) in matlab ?
    // here it is dst(x,y) <- src(mapx(x,y),mapy(x,y))
    cvRemap( src, dst, mapx, mapy, flags, cvScalarAll(0) );

    __END__;

    cvReleaseMat( &mapx );
    cvReleaseMat( &mapy );

}
```

Appendix J. Persistence effect algorithm

The persistence effect was implemented using native EyesWeb blocks which created a feedback loop. An image stream passed through an OR circuit (essentially, a summer) which sums the current input image with the output from the chain. The rest of the chain is a filter that creates some sort of blur. The following image shows such a feedback loop where an input image stream, consisting of a single moving white pixel on a black background, being processed by a linear blur 5x5 filter:

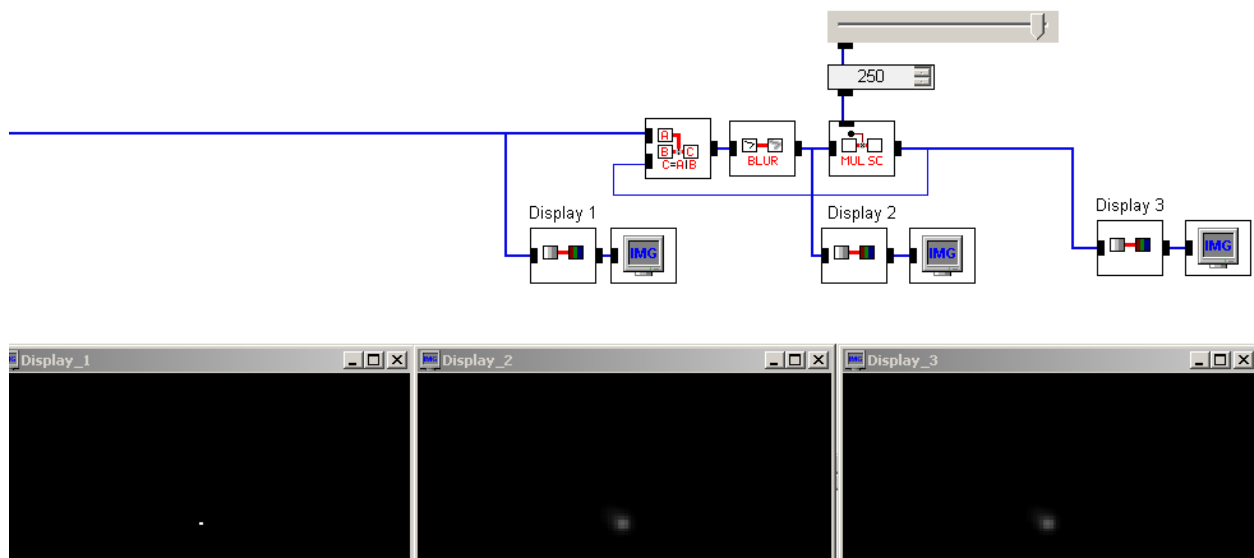


Figure 38. EyesWeb feedback loop with a linear blur filter

Obviously, when the input is a single pixel, the output appears smeared – what we see is basically the pulse response of a filter.

The Gaussian filter is quite similar to the linear blur one – except the brightness of the "halo" around the processed pixel has a different gradient.

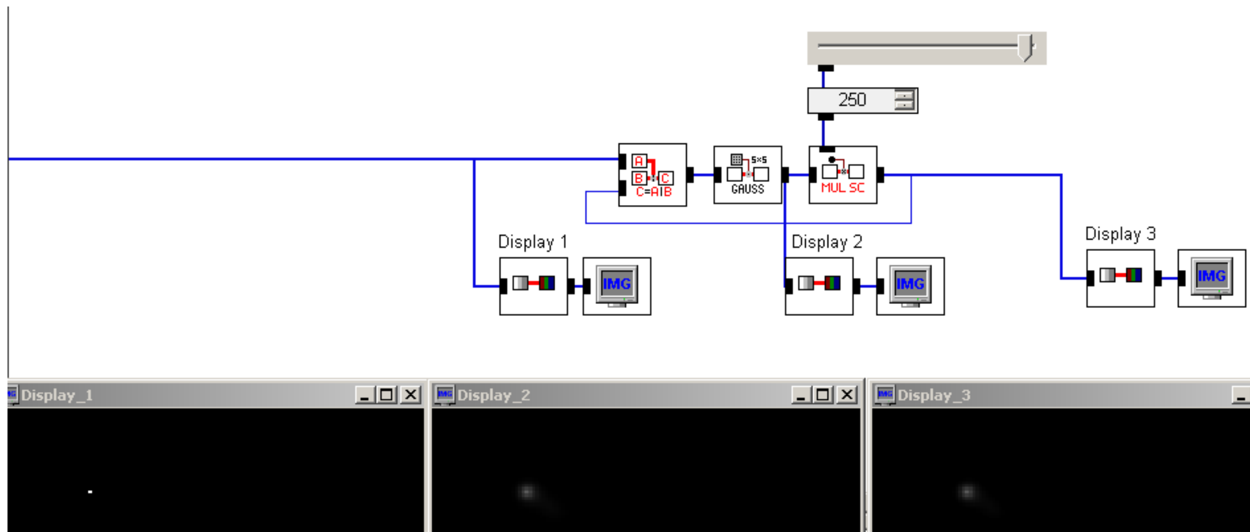


Figure 39. EyesWeb feedback loop with a Gaussian filter

Some may note the slight difference – that the linear blur filter smears out the original input pixel, whereas the Gaussian blur preserves it. The Multiply Scale block with the slider is used to simply control the brightness of the output, and this control the amount of the feedback loop. This is what helps how long in time will the "smudge" persist – considerfor instance:

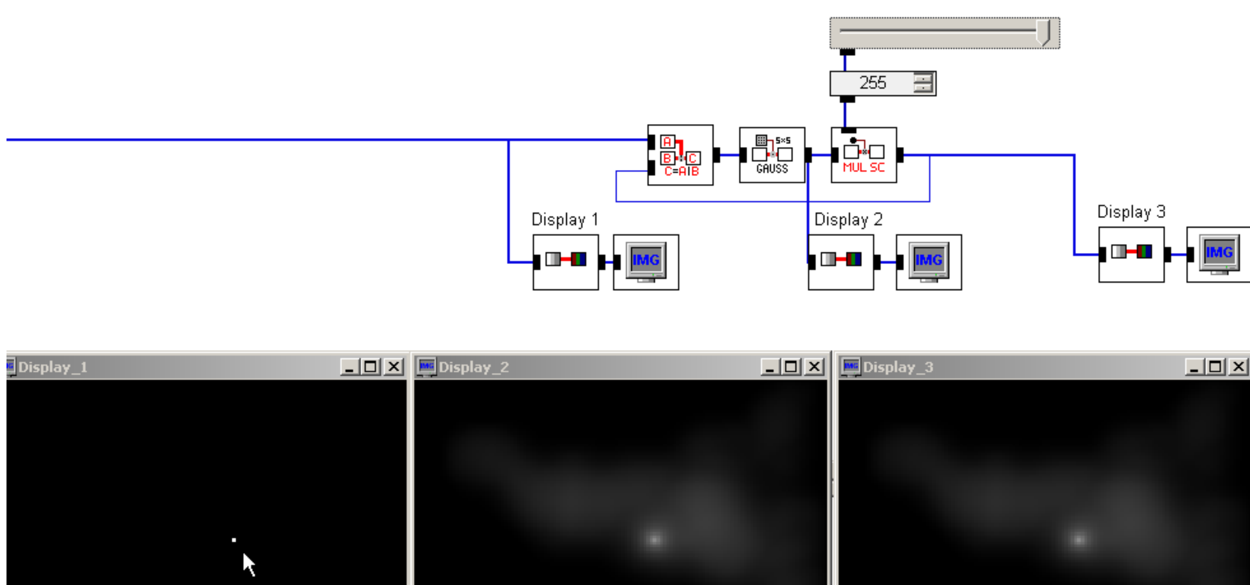


Figure 40. EyesWeb strong feedback loop with Gaussian filter

Notice that the past positions of the point persist over such a long time that they begin to look like a cloud. The final version implemented in EyesWeb utilised both linear blur and Gaussian filter, tuned to different sizes of their filtering response matrices – the multiplication slider was put between them, whereby the following was obtained:

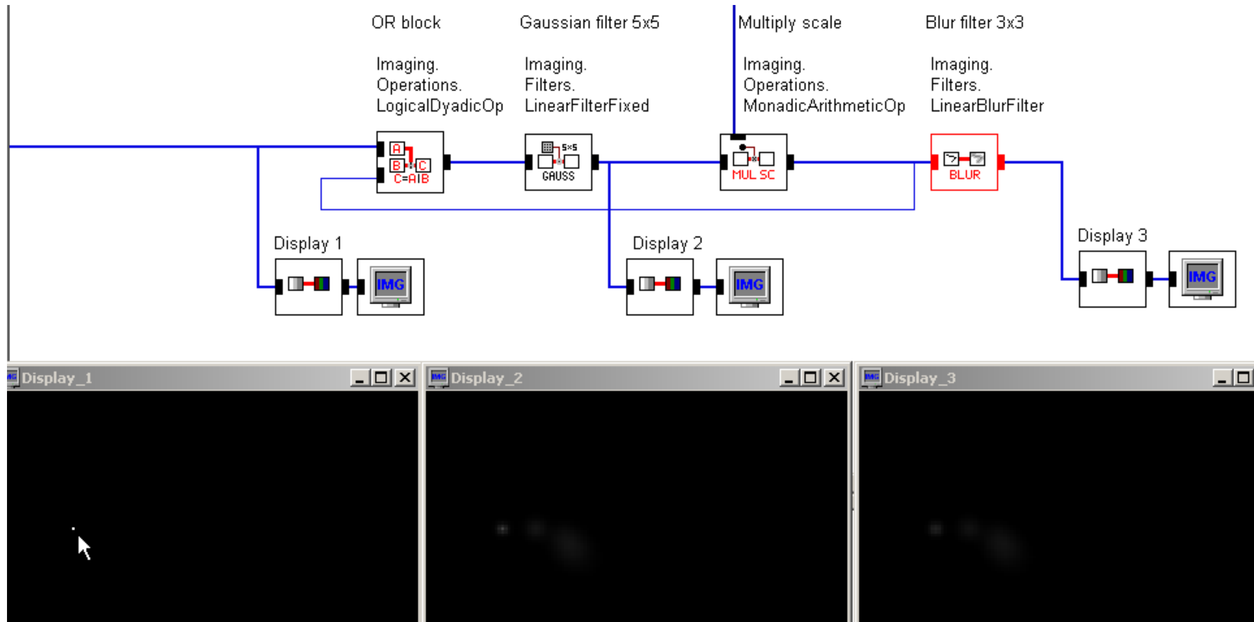
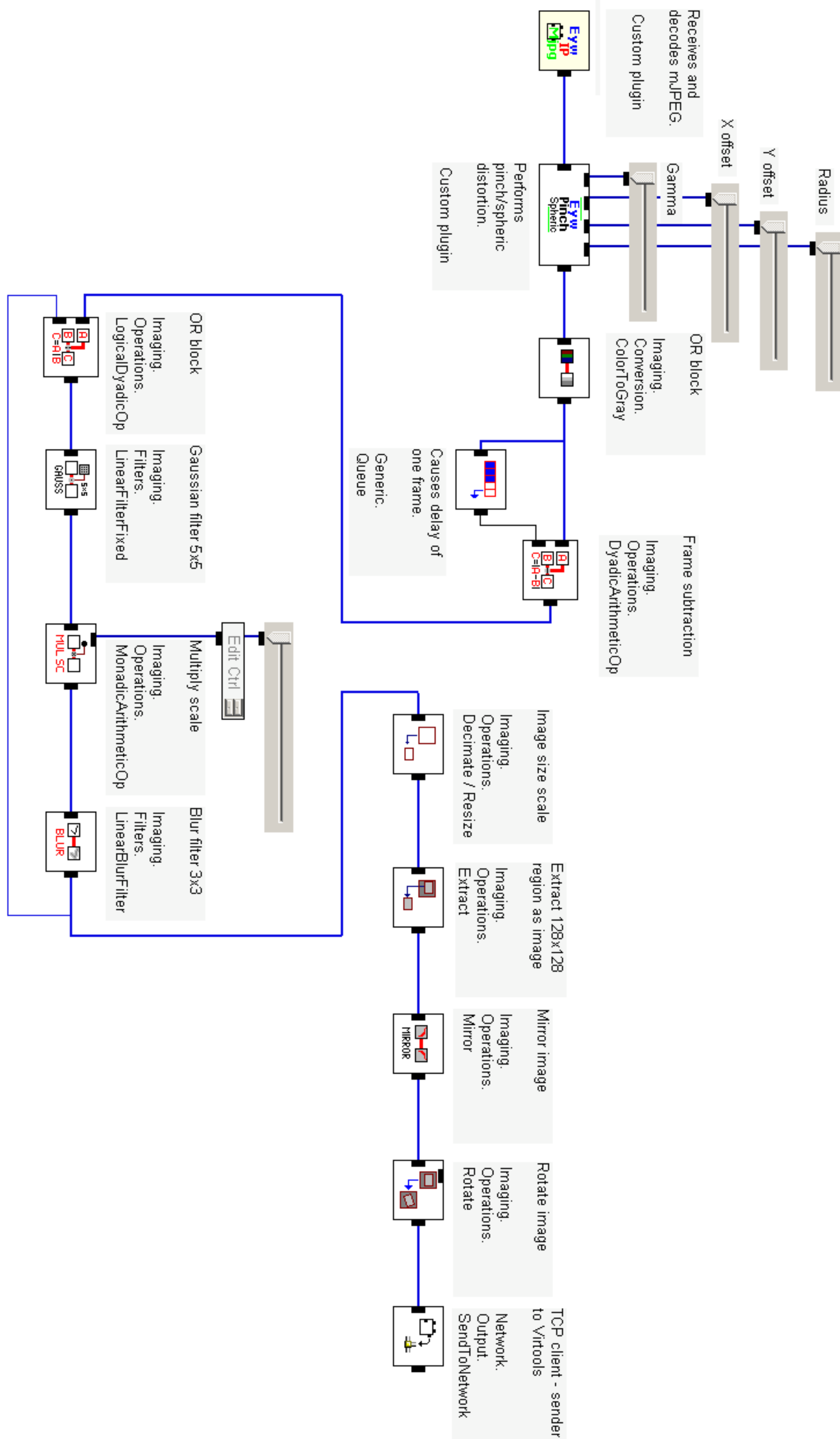


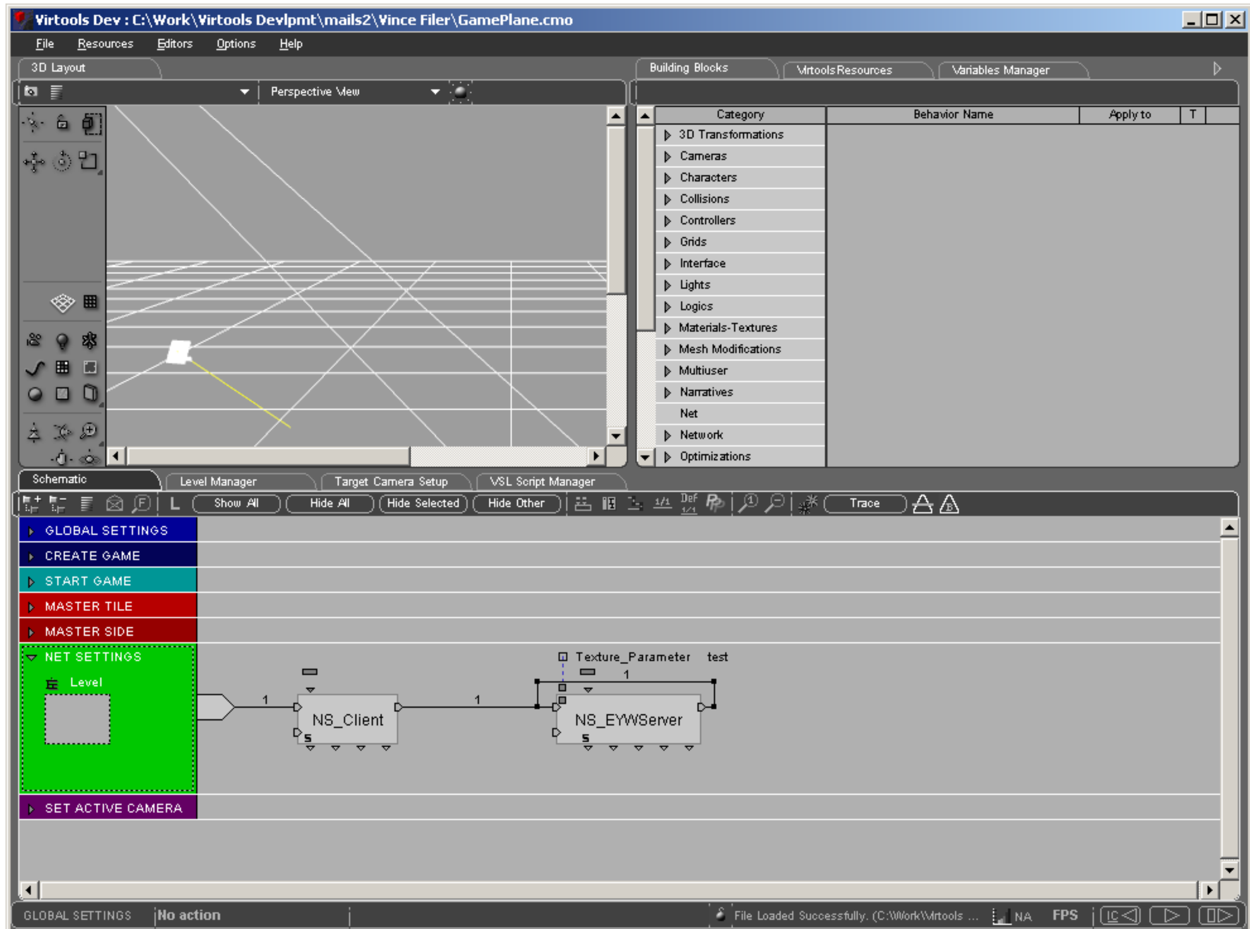
Figure 41.Time persistence feedback loop implemented in EyesWeb input processing

The blocks used from EyesWeb and their corresponding classes are noted on Figure 41. The proper tuning of these elements allow for a comet-like trail – a persistence effect intermediate between the strong feedback loop on Figure 40 and a weak feedback loop such as on Figure 39.

Appendix K. Screenshot of EyesWeb patch with plugins



Appendix L.   Screenshot of Virtools patch with plugins



List of references

[1] Mayenes-Aminzade, D., Pausch, R., and Seitz, S. 2002. Techniques for interactive audience participation. In IEEE Int. Conf. on Multimodal Interfaces, Pittsburgh, Pennsylvania.

[2] EyesWeb homepage, http://www.infomus.dist.unige.it/eywindex.html

[3] Virtools homepage, http://www.virtools.com/

[4] Axis homepage, http://www.axis.com/

[5] Bourke P., Projection personal pages, Conversion of images from OneShot camera, http://astronomy.swin.edu.au/~pbourke/projection/oneshot/

[6] S. Baker and S.K. Nayar, "A Theory of Single-Viewpoint Catadioptric Image Formation," International Journal of Computer Vision, Vol. 35, No. 2, 1999, pp. 1 - 22. http://www.ri.cmu.edu/pubs/pub_2553_text.html

[7] Chen F., Arunachalam K., Geometric Transformation Pinched Hallway and its restoration, project report, original website address http://www.public.iastate.edu/~fengchen/course/ee528/pjt1/getra.htm (not online anymore, use google cache)

[8] Bregler, Christoph, et al. 2005. Squidball: An Experiment in Large-Scale Motion Capture and Game Design. NYU Technical Report TR2005-869, New York. http://squidball.net

[9] Carpenter, L., Cinematrix, Video Imaging Method and Apparatus for Audience Participation. US Patents #5210604 (1993) and #5365266 (1994)

[10] PlaneShift, http://www.planeshift.it

[11] Anarchy Online, http://www.anarchy-online.com/

[12] LiveActor, University of Pennsylvania, http://hms.upenn.edu/LiveActor/

[13] Human PacMan, Mixed Reality Lab Singapore, http://155.69.54.110/RESEARCH/HP/HP_webpage/research-HP-infor.htm

[14] Sawyer, Charlie, CSCIE 160 Java For Distributed Programming, Harvard University, Distributed Computing I Lecture Notes, http://courses.dce.harvard.edu/~cscie160/UnitV-A.htm

[15] Client-server - From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Client-server

[16] The Java Tutorial, What Is a Socket? http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html

[17] Sockets Tutorial, Systems Programming course notes, Rensselaer Polytechnic Institute, http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html

[18] Smith, John B., WWW's Client/Server Architecture, Advanced WWW Programming, University of North Carolina, http://www.cs.unc.edu/Courses/wwwp-s98/docs/lessons/www/arch/client-server-arch/

[19] Windows Sockets home page, Microsoft, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp

[20] MJPEG From Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/MJPEG

[21] Lundh, F., Sockets, http://effbot.org/zone/socket-intro.htm

[22] Intel IPP (Integrated Performance Primitives libraries) page, http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm

[23] OpenCV – Open Source Computer Vision Library, Intel, http://www.intel.com/technology/computing/opencv/index.htm

[24] Intel JPEG Library (IJL) page – http://developer.intel.com/software/products/perflib/ijl/

[25] EyesWeb forum entry, http://eyw.free.fr/forum/viewtopic.php?p=50

[26] Sphaero's VnetSync http://www.sphaero.org/projects/ns/

[27] Virtools forum entry, http://www.theswapmeet-forum.com/viewtopic.php?t=5771

[28] OSI model, Wikipedia, http://en.wikipedia.org/wiki/OSI_model

[29] 7-layer OSI Model, NetworkClue, http://www.networkclue.com/routing/tcpip/osi-model.php

[30] Java Sockets, http://jan.netcomp.monash.edu.au/distjava/socket/lecture.html

[31] Java networking: Sockets http://doc.novsu.ac.ru/oreilly/java/fclass/ch08_01.htm

[32] Image Representation Introduction, Digital Image Processing Documentation, Wolfram Research, http://documents.wolfram.com/applications/digitalimage/UsersGuide/ImageRepresentation/ImageProcessing2.1.html

[33] Monochrome Image Representation, Digital Image Processing Documentation, Wolfram Research, http://documents.wolfram.com/applications/digitalimage/UsersGuide/ImageRepresentation/ImageProcessing2.2.html

[34] Eric W. Weisstein. "Matrix." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/Matrix.html

[35] Eric W. Weisstein. "Function." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/Function.html

[36] Matlab Documentation – Function reference for surf, surfc - http://www.mathworks.com/access/helpdesk/help/techdoc/ref/surf.html

[37] Ron Avitzur, Visualizing Functions of a Complex Variable, http://www.pacifict.com/ComplexFunctions.html

[38] Matlab Documentation, 3D Visualization, http://www.mathworks.com/access/helpdesk/help/techdoc/visualize/visualize.html

[39] Van Walree, P., Distortion, Optics articles, http://www.vanwalree.com/optics/distortion.html

[40] Radial Distortion Correction, RadCor - A Lens Distortion Correction Program, Rheinisches Landesmuseum in Bonn, Germany, http://www.uni-koeln.de/~al001/radcor.html

[41] Drakos, N., Moore, R., Aberrations, Seidel Aberrations, http://astron.berkeley.edu/~jrg/Aberrations/node5.html

[42] Hicks, R. A., The Page of Catadioptric Sensor Design, Department of Mathematics, Drexel University, http://www.cs.drexel.edu/~ahicks/design/design.html

[43] Axis Communications, Online Lens calculator for Axis 206 camera, http://www.axis.com/techsup/cam_servers/lens_calculators/lens_206.htm

[44] Independent JPEG Group homepage, http://www.ijg.org/

[45] Eric W. Weisstein. "Polar Coordinates." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/PolarCoordinates.html

[46] Bourke P., Image warping / distortion, http://astronomy.swin.edu.au/~pbourke/projection/imagewarp/

[47] Cauchois, C., Brassart, E., Delahoche L., Clerentin A. "3D Localization with Conical Vision," cvprw, p. 81, 2003 Conference on Computer Vision and Pattern Recognition Workshop - Volume 7, 2003.

[48] Spacek, Dr L., Omnidirectional Vision page, http://cswww.essex.ac.uk/mv/omni.html

[49] CAVE (Columia Automated Vision Environment) Laboratory webpage, Publications, Catadiotric Vision, http://www1.cs.columbia.edu/CAVE/research/publications/catadioptric.html