Image Preparation for 3D-LCD

Cees van Berkel[§]
Philips Research Laboratories, Redhill, UK

ABSTRACT

The simplicity and inherent robustness of the Philips 3D-LCD, both in manufacturing and usage, make it highly suitable for a cost effective, mass-market autostereoscopic display For successful adoption in a wide range of applications, efficient 3D image preparation is very important. A generic expression for the relation between LCD pixels and the multiple perspective views is derived that can be used in the image preparation for different 3D-LCD systems. This paper then describes two approaches to 3D image preparation. One is an intuitive graphical user interface and the second is at source code programming level as an extension to the existing OpenGL 3D graphics API. Using the latter we examine the computer overhead of the 3D image preparation process.

Keywords: 3D, multiview, autostereoscopic, lenticular, display, LCD, OpenGL

1. INTRODUCTION

The Philips multiview 3D-LCD provides a truly autostereoscopic display. It requires no artificial devices, allows freedom of movement in front of the display and can be seen by any number of people at the same time.

In the 3D-LCD, a sheet of cylindrical lenses (lenticulars) is placed on top of an LCD in such a way that the LCD image plane is located at the focal plane of the lenses. The effect of this arrangement is that different LCD pixels located at different positions underneath the lenticulars fill the lenses when viewed from different directions. Provided these pixels are loaded with suitable stereo information, a 3D stereo effect is obtained in which left and right eyes see different but matching information.

Traditionally the lenticular approach to 3D-LCD has two important drawbacks. Firstly it suffers from a Moiré-like effect in which the user sees dark bands on the screen which result from the lenticular imaging of the black space between the LCD pixels [1]. Secondly, it makes uneven use of the horizontal and vertical pixel resolution in the LCD panel, typically the resolution in each eye is obtained by dividing the horizontal LCD pixel count by the number of views that is offered, without affecting at all the vertical resolution[2].

Our approach to lenticular 3D-LCD solves both these problems by not placing the lenticular cylinder lenses vertical and parallel to the LCD column direction, but by slanting them at a small angle. This simple device has the effect of dissolving the Moiré-like black bands and also means that both the horizontal and vertical pixel resolution are used to populate the individual views that reach the user's eyes. For example, in a 7 view system[3] the horizontal resolution of each view is reduced by a factor 2.5 from the original LCD and the vertical resolution by a factor 3. Hence the overall

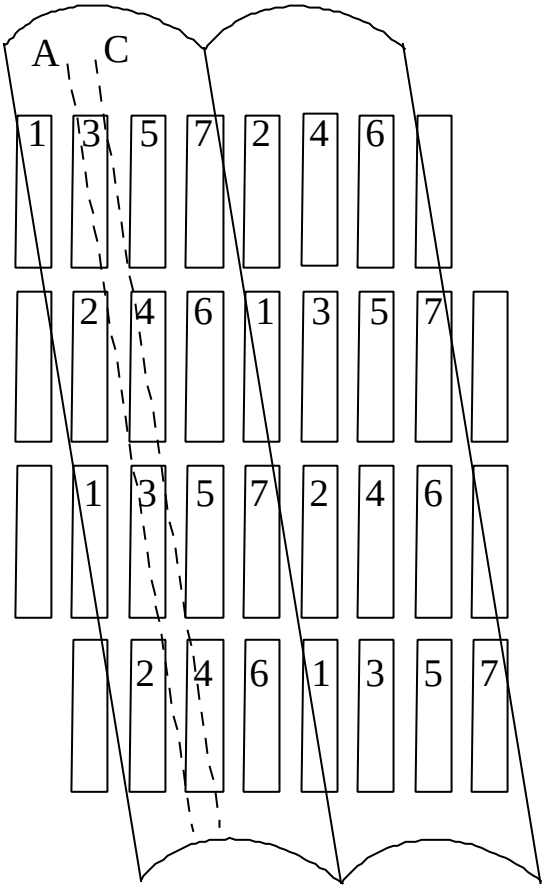

Figure 1 Slanted Lenticular 3D-LCD

[§] berkel@prl.research.philips.com http://www.research.philips.com/generalinfo/special/3dlcd

resolution reduction of a factor 7 is achieved by both a horizontal and vertical resolution reduction by factors nearing √7. Figure 1 illustrates the slanted lenticular 3D-LCD for this 7 view system. The vertical rectangles represent the LCD R-G-B colour triplets and the numbers indicate the view number that the individual sub-pixel belongs to. Line A represents the location of points on the LCD imaged by the lenticular into a specific direction; the direction of view 3. Similarly, line C represents the points along view 4. As can be seen, because of the lenticular slant, the pixel view numbers appear in a interlocking 2D pattern and it is this 2D topography that means that both the horizontal and vertical LCD pixel resolution is used to constitute the different views.

In this paper we will discuss the image preparation process for the 3D-LCD. We will do this by first deriving a generic expression for the pixel mapping relation between LCD pixels and the view number the pixels belong to in a multiview system. We will then discuss two ways in which this expression can be used. Firstly through a graphical user interface in which image manipulation takes place on the computer desktop through intuitive point and click actions. Secondly at the application programmer's source coding level in which the 3D-LCD functionality can be achieved through a few simple extensions to existing 3D graphics software libraries. All these methods of 3D-LCD image preparation make use of existing monoscopic 3D computer graphics capabilities. These software architectures were designed and optimized for the monoscopic rendering process and there are obvious drawbacks in using these for multiview 3D-LCD. However, as we will show, the critical measure in multiview rendering process is not the number of views, but the total number of pixels rendered.

2. MULTIVIEW PIXEL MAPPING

To determine the view number of a given point *x,y* in the plane of the LCD, we need to know the horizontal offset of that point with respect to the edge of the lenticular under which it is positioned. Using the micro lens magnification *m* and other definitions apparent from figure 2, this offset is given by

$$x_{offset} = (x - y\tan(a)) \bmod (\frac{m+1}{m}\frac{p_m}{\cos a}) \quad (1)$$

in which p_m is the pitch of the micro lenses measured perpendicular to its long axis. $p_m/\cos\alpha$ the pitch measured along x-axis, and

$$(\frac{m+1}{m}\frac{p_m}{\cos a}) \quad (2)$$

the projection of that pitch onto the LCD plane using the viewing position as origin. The magnification *m* can be expressed in terms of the viewing distance *D* and the lens focal length *f*, as *m+1 = f D*. To simplify things, we divide the projected horizontal lens pitch by the pixel pitch of the LCD p_h and call this the number of *views per lens X*.

$$X = \frac{m+1}{m}\frac{p_m}{p_h \cos a} \quad (3)$$

Note that *X* is the number of views per lens measured along a single row of the LCD and is different from the total number of views in the multiview system. For instance, in the 7 view example of figure 1, *X*=3.5.

For a data graphic LCD in which pixels are arranged as an orthogonal array of RGB colour triplets, the coordinates *x,y* can be expressed in terms of the pixel indices *k,l* and the

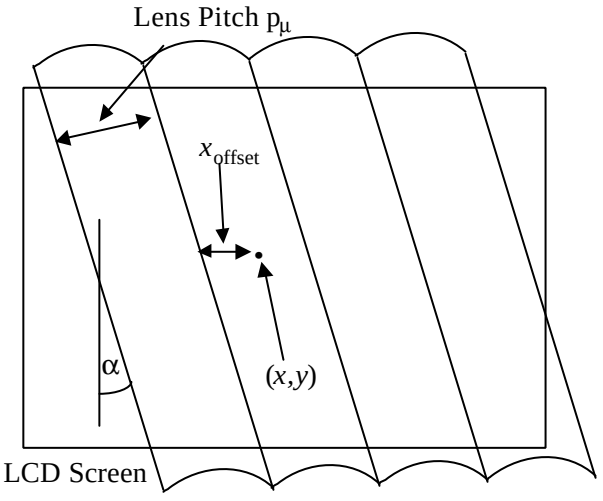

Figure 2 Multiview Pixel Mapping

horizontal pixel pitch p_h as follows:

$$x = kp_h$$
$$y = 3lp_h \qquad (4)$$

Note that the indices k,l point to individual red, green or blue (sub) pixels and not to colour triplets. Other relationships between pixel indices and x,y can be written down for displays with different pixel layouts such as video and projection displays.
Dividing the expression for x_{offset} above by the projected horizontal lens pitch, inserting the definitions for X,k and l, and introducing N_{tot}, the total number of views, we find for the view number N of each sub pixel k,l

$$N = \frac{(k + k_{offset} - 3l \tan a) \bmod X}{X} N_{tot} \qquad (5)$$

This equation can be used to calculate the view number N for each pixel k,l which can then be used to assign the appropriate image data to the pixel. The parameter k_{offset} is introduced into the formula to accommodate an arbitrary horizontal shift of the lenticular lens array with respect to the LCD.

For specific sets of parameters, for instance the 7 view example above (k_{offset}=0, a=9.4623°, X=3.5 and N_{tot}=7), the view number N will always be an integer as there are only 7 different positions in which the LCD sub pixels can be located relative to the lenticular lens. For arbitrary values of a and X and finite values of N_{tot}, N will not be an integer. In that case, in the actual mapping, we simply take the nearest integer of N to decide from which input image to take the information for a given pixel.

Expression (5) is generic and can be used to describe almost any lenticular/LCD combination. For example, $a=0$ describes traditional non-slanted lenticular 3D displays[4]. For α=9.4623°, X=1 and N_{tot}=2 it provides a pattern of horizontally interlaced views used in some holographic displays[5,6]. It is also worthwhile to note that using this formula, four parameters describe the mapping between pixels in the perspective views and the 3D-LCD. These are X, α, N_{tot} and k_{offset}. Using an interactive tool these can be set for any lenticular/LCD combination and then used to describe the view interlacing or 'weaving' process.

An important issue in the slanted lenticular arrangement is the appearance of the colour stripes in each view. In a data graphic LCD the colour filters are arranged in vertical Red-Green-Blue stripes. However, because of the slant, the colours present themselves in the individual views in stripes at varying angles to the vertical. Indeed, for 3,6 and 12 view systems the colours appear in wide horizontal stripes. We can measure the pitch of these colour stripes for 3D-LCD systems with differing number of views. Figure 3 shows the result by showing the stripe pitch measured in LCD pixel units as function of the view number. The slant angle α is fixed at atan(1/6)=9.4623°. Also indicated in figure 3 are the optimum square root line and the worst case unity line. The unity line represents the case for a conventional, non-slanted lenticular arrangement in which the horizontal size of the pixels grows linearly with the number of views. As we can see in figure 3, the colour stripe pitch of the slanted lenticular 3D-LCD oscillates between these two extremes as function of number of views, with minima at 4 and 9 views and maxima at 3 and 6 views. This feature provides an interesting design consideration; although a 7 view system has a higher pixel count per view, the smaller colour stripe pitch of an 8 or 9 view system will give those systems a better visual appearance.

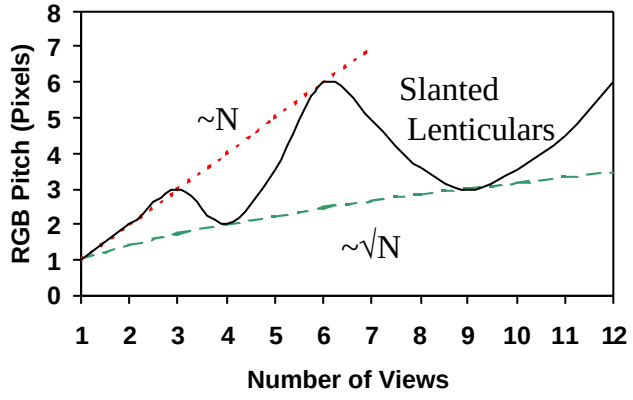

Figure 3 Slanted Lenticular Colour Stripe Pitch

The versatility of the 3D-LCD design makes it suitable for a large number of applications in which a number 3D image sources offer material for display. These range from complex multiview video camera systems [7,8,9], post processed stereo or monoscopic video footage, 3D vector model information in for instance VRML format [10] and scientific point cloud data [11] or slice information such as CT scans.

All these image sources can be used to provide, extract or compute different monoscopic 2D images that are used by the 3D-LCD multiview display to provide a 3D image. In that sense the Philips 3D-LCD is principally a stereoscopic display in the sense that it creates a 3D illusion by providing different 2D images to the user's eyes. In the remaining sections of this paper we will discuss some of the issues involved in this image preparation process.

3. IMAGE PREPARATION - GRAPHICAL USER INTERFACE

There are a number of different environments in which image preparation for the 3D display can take place. In this paper we will consider two. One environment is the computer desktop in which the graphical user interface of a custom computer application is used to manipulate images and image sources to create 3D images. Another environment is the programming environment in which the application programmer writes lines of code to cause a computer program that otherwise would create only monoscopic 3D computer graphics, to create multiview 3D images directly. Identification of these two environments is not a comprehensive classification. In particular multiview image preparation in video production and broadcast TV fall well outside it. However, while the principles that will be discussed below apply most directly to computer graphics, some of the conclusions are of sufficient generality to be of interest in other fields also.

A computer program called 'Octopus Multiview Editor' has been developed to allow the user to manipulate sets of 2D pictures to create multiview 3D images for the 3D-LCD. It aims to provide both intuitive and instructive access to the 3D-LCD functionality by showing how *individual* images can be used or mapped in the 3D-LCD displays and by suggesting how *sets* of images can be used to create complete multiview 3D pictures.

Figure 4 illustrates the overall data flow from the 2D image domain to the 3D image domain. At the top are the different image sources; bitmaps or 3D vector models on the computer hard drive or remote server, (multiple) video cameras and image data from 3rd party applications running on external hosts. All these sources provide different monoscopic perspective images, which are assigned to the 3D display using the generic multiview pixel mapping expression, discussed above.

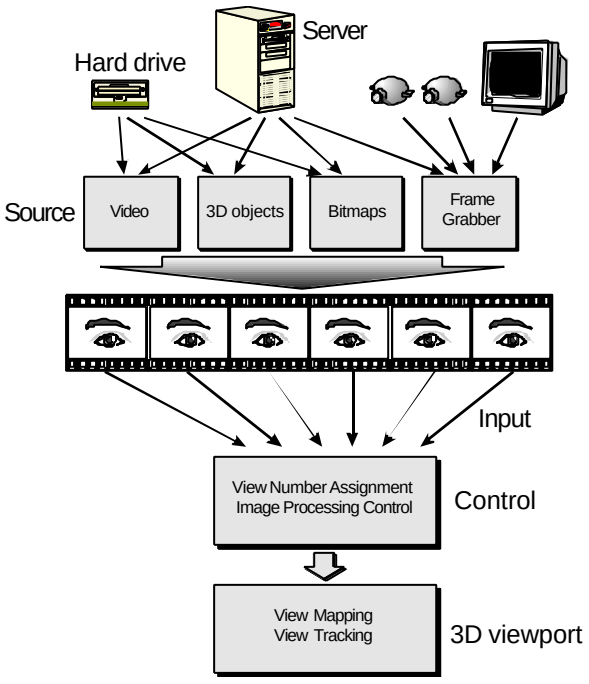

Figure 4 Image Preparation Flow

The program creates not just one, but two windows on the desktop. One is the 2D Control Port in which the multiple (2D) image sources are opened just like any other application that handles bitmaps, video movies, cameras or Internet connections. These individual images can then be assigned to different view numbers by simply pressing appropriate buttons on a toolbar. The result is visible in the second window that the program creates on the desktop; the 3D View Port, which is intended to be viewed through the lenticular sheet. Hence the graphical design of the program directly supports the concept that image preparation for the 3D-LCD is mainly about combining multiple 2D images into a single 3D image. It also underscores that the image quality of the 3D image is controlled by the image quality of the individual 2D views that constitute it.

Whilst the basic units of the program are the individual 2D images, these images can often be grouped in 'sets' because they share important characteristics. The Octopus Multiview Editor offers different tools to create and manage

such sets of images. One example is different camera shots of a single VRML scene. The program will create these images in different sub windows in the 2D Control Port, each with a progressive horizontal camera angle and position offset, and assign them to a staircase of view numbers. This will result in an overall 3D view of the VRML scene in the 3D View Port, all through the execution of a single command. To control the quality of the 3D image, the program user can alter the characteristics of the individual views, or control global 'set' parameters such as camera convergence distance. Also VRML interactivity and camera position control is possible through the master 'Examiner' window, which the other image windows will 'follow'. Thus the program allows the user to conceptually shift gear from working with individual images, to sets of images, or just to the 3D scene itself through the Examiner window.

Another example of 'set' generation is that of frame delays into video movies. Akin to the Pulfrich effect, for suitable source material, the frame delay can be used to provide the parallax required for 3D display. By creating multiple images of the movie sequence, each with progressive frame delay, a 'set' of windows is created that can be assigned to a staircase of view numbers, resulting in a single 3D image. As before, the user can choose to alter the image characteristics of the individual to change the 3D image or to control the overall animation properties of the movie. Again allowing different levels at which to approach the 3D image preparation process.

4. IMAGE PREPARATION - PROGRAMMING INTERFACE

The merits of a graphical user interface for 3D-LCD image preparation are considerable in terms of its versatility and instructive value. However, once, within a given application context, the characteristics of 3D image preparation have been explored and the requirements defined, it is often desirable to incorporate the 3D-LCD capability into the end user application package. Such applications may be Internet browser software (plug-ins), games, presentation software and others. They often exist already in a monoscopic (2D) version and there is a need to adapt them for the 3D-LCD. There are different levels at which such adaptation might be made. Figure 5 illustrates the different software layers of toolkits, graphics API (application programming interface) and operating systems that a program for 3D graphics may or may not make use off. From this we can identify different levels at which an adaptation for 3D display can be made.

1) Within the Operating system. As additional graphics drivers or indeed designed into the operating system itself. This may appear to be the most desirable level because it implies that existing applications can benefit from the 3D display functionality without originally having been written for it. However it is also the most challenging because the information required to create a good stereoscopic image (i.e. depth) is often not available. There have been some attempts to overcome this problem related to the general 2D/3D conversion issues. In the future this issue may be alleviated as the 2D desktop metaphor makes way for 3D-world representation on the computer.

2) Within an API such as OpenGL[12]. This is the ideal level because all the information required for stereoscopy is available. Whereas it would be ideal to do this within the API itself, copyright and license issues prevent this in the short term and instead the 3D-LCD functionality can be offered as an extension or utility to the API..

3) Within a 3D toolkit such as Open Inventor[13] or VTK[14] (Visualization Toolkit). The advantage of this is that the extension can be made much more sophisticated, making it easier to transform the versatility of the Graphical User Interface into an equally versatile programming interface.
From the application programmers point of view 3D-LCD adaptation at level 3 is very easy and we have successfully done this for both Open Inventor and VTK. However, the constituency for these is much smaller than that for the 3D APIs, particularly OpenGL. It is therefore for the latter that we see the most widespread use of a programming interface to 3D-LCD.

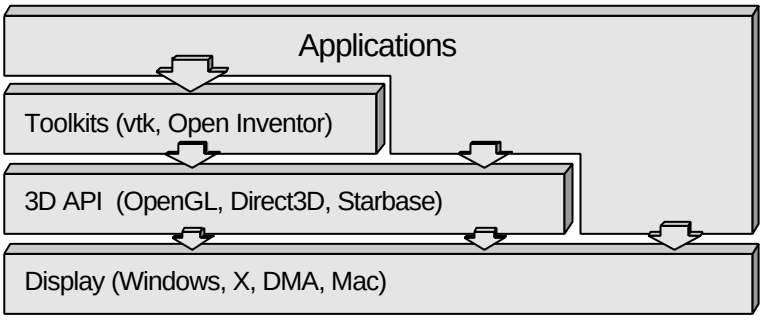

Figure 5 3D Adaptation levels

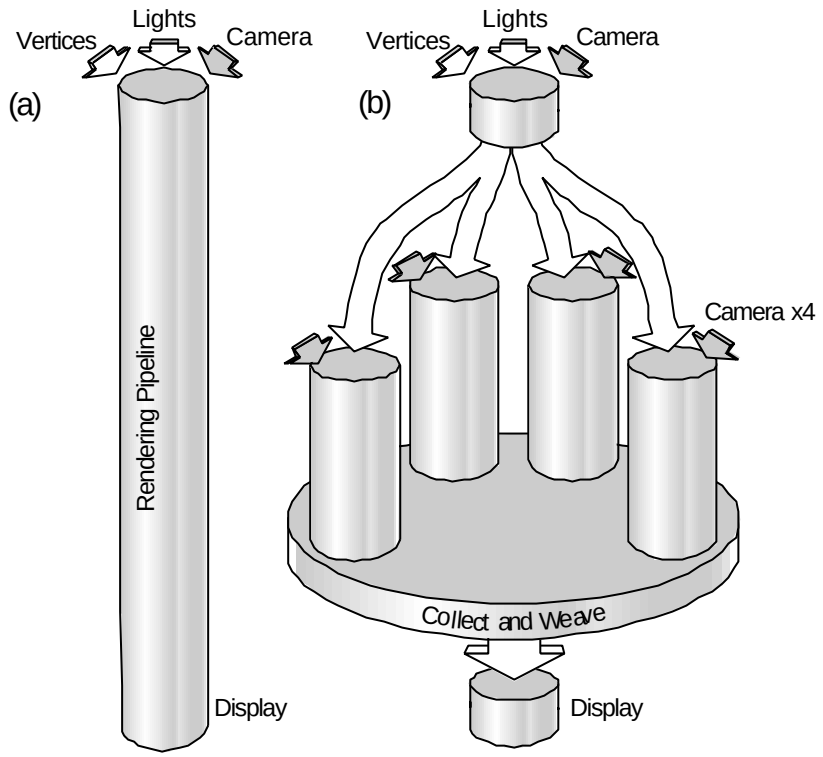

Figure 6 Repeated Rendering Pipelines

5. REPEATED RENDERING PIPELINES

At a general level any 3D rendering program will contain the repeated execution of a few basic steps. These are 1) Handling user input and events inside the 3D world, 2) Pointing the camera at the scene and 3) Rendering the scene. In addition to this, most programs use double buffering in which at step 3 the scene is rendered in a hidden back buffer and a further step is necessary 4) swapping front and back screen buffer. The execution of step 3 is often referred to as the rendering pipe line; the process whereby the graphics engine takes the vertices, light and camera position as indicated in figure 6a, and performs transformations, clipping, culling and filling in a rendering pipline.

One way in which to approach the use of a 3D-LCD, is to force the graphics engine to go through the pipeline not once, but several times. As illustrated in figure 6b, each time with the same vertices and lights, but with a different camera position. Each offset slightly differently from the main camera direction. The results of these different scene renders are collected and combined into the 3D-LCD to provide a 3D picture. Although this process may appear complicated and inefficient, it is actually neither. We will leave the latter point for a section below. The former issue is dealt with now.

A simplified but nevertheless typical piece of a 3D graphics program is shown in figure 7a. The MyDisplayFunc is called each time a new screen needs to be created. It does that by executing or simulating the steps outlined above. Moving the camera (steps 1 and 2), drawing the scene using OpenGL (step 3) and swapping the font and back buffer (step 4).

To change this example into a 3D-LCD capable program the function MyDrawScene() needs to be called several times. Each time with a slightly different camera position and the results stored, managed and placed in the back screen buffer as illustrated in figure 6b. This is achieved by using a custom written function glP3D_MultiViewCamera() which handles all these tasks by simply wrapping the MyDrawScene function in a while loop. The function will manipulate the camera position and cause multiple execution of the MyDrawScene function by returning TRUE several times. After the last required call to MyDrawScene the function performs the pixel mapping task, writes the result into the back buffer and returns FALSE to allow escape from the while loop and a call to MySwapBuffer to make the result visible from the screen.

```
void  MyDisplayFunc(void)
{
 MyMoveCamera();
 MyDrawScene();
 MySwapBuffers();
}
```

Figure 7a - Standard 3D graphics code

```
void  MyDisplayFunc(void)
{
  MyMoveCamera();
  while(glP3D_MultiViewCamera ()){
        MyDrawScene();
  }
  MySwapBuffers();
}
```

Figure 7b 3D-LCD Adapted code

A detailed description of the glP3D_MultiViewCamera() falls outside the scope of this paper, but the function and other helper functions to control the number of images generated, the pixel mapping (number of views) etc, are written in standard C and have been used on both Unix and PC platforms. Note that no modifications are necessary to OpenGL code in MyDrawScene() or any other function in the program application code.

6. MULTIVIEW RENDERING PERFORMANCE

Having discussed the relative simplicity with which 3D-LCD functionality can be added to an application program, we will now address the computational overhead of this. To obtain a quantitative measure for this, we have measured the frame rate of an application that uses the example code in figure 7b as function of the number of views. The result is shown in figure 8. The test program renders a medium complex scene that includes a texture-mapped background. An XGA display window was used of 1024x768 pixels. The unit on the y-axis in figure 8 is the render time per pixel. The experiment was performed on a 400MHz PC with Windows95/OSR2. The PC platform was chosen because it is the most common platform that the 3D-LCD is used on. Two different conditions were used in the experiment. In one, the size of each rendered view was kept constant as the number of views was increased (the dashed line in figure 8). In the other the number of pixels rendered in each view was reduced in proportion to the number of views. This keeps the total number of pixels rendered constant. The full line in figure 8 indicates this result. The number of views on the x-axis ranges from 0 to 12. '0 views' corresponds to no multiview functionality, i.e. program execution with code as in figure 7a or the pipeline in figure 6a. '1 view' corresponds to the use of the repeated pipeline mode with just one view image. For 0 and 1 view there is no difference between the two experimental conditions and the pixel rendering times are identical. For higher view numbers, however, there is a heavy multiview rendering performance penalty of 1.9 µs/pixel/view for constant rendered view size. Under those conditions it takes 9 times longer to create an 8 view 3D-LCD image than a monoscopic picture. Under the 2nd condition however, when we render only as many pixels as we need in the final image, the multiview overhead is much smaller at 0.21 µs/pixel/view. Indeed, in that case rendering an 8 view image is only 40% times slower than the 0 view monoscopic rendering.

Consider the aspects of the multiview rendering process that will cause a performance penalty in comparison with the standard monoscopic rendering process. These are: Pure rendering overhead, Buffer copying and writing, Weaving or pixel mapping overhead. The first issue simply refers to the computational time spend in the OpenGL graphics library functions that draw the different views i.e. the time spent in the different repeated rendering pipelines. The experiment above demonstrates the importance of the total number of pixels rendered which indicates that the pixel filling can provide a significant bottleneck in the pipelines. The 2nd issue relates to the fact that we need to transfer the data out of the back screen buffer after each render and write the final result back into it. The final point refers to the work using the generic pixel mapping expression above to assign the pixels from the individual views to the final 3D picture. Table 1 illustrates how the computational overhead is divided up between these tasks. These percentages were measured for 8 views, constant total rendered pixels. As we can see the bulk of the work is still in the OpenGL graphics library. The specific multiview tasks; pixel copying, writing and mapping, take up a relatively smaller part of the time.

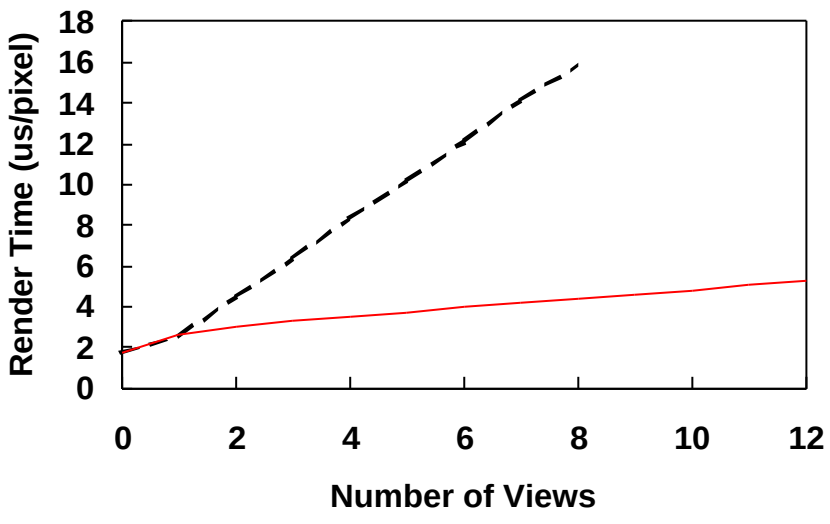

Figure 8 Multiview Rendering Performance

We have also performed these experiments on a Silicon Graphics Inc (SGI) Octane confirming the same general trends on this platform. The high degree of hardware acceleration on the SGI means however that the rendering is

performed very efficiently and the total number of rendered pixels is much less important that it was on the PC. The second column in table 1 illustrates the task percentages for this case. The relative load of the pixel mapping or weaving tasks is now much larger but the rendering pipelines still take up a significant amount of time. This probably means that in this case the pipeline bottleneck lies else where and we look forward to investigate this issue in the future.

Table 1 Multiview Tasks

Task	PC	SGI
Rendering	56 %	24 %
Pixel Copy	14 %	22 %
Pixel Write	15 %	1 %
Pixel Mapping	9 %	50 %

7. CONCLUSIONS

In this paper we have discussed a 3D image preparation process that is based on a generic expression for multiview pixel mapping. This multiview pixel mapping was developed for the Philips 3D-LCD but can also be used to describe many other 3D display systems. The generality and versatility of the expression means that it can be used for many different applications, taking a wide range of image sources as input to it. The mechanics of the pixel mapping, and the image material that is required as input to it, can be explored firstly through a graphical user interface and secondly as an extension of well known 3D programming APIs, making it easy to adapt existing graphics applications to the 3D-LCD.

Although there is a significant performance overhead for the multiview image generation, this overhead is not as severe as might be expected. In particular if the total number of pixels rendered by the graphics engine is limited only to the required minimum, a frame rate reduction of only 40% can be achieved. Specific multiview tasks such a pixel mapping and buffer management contribute a relatively small factor to this reduction. We are hopeful that when in the future we will see the integration of the multiview functionality into 3D graphics engines, the resultant optimizations will lead to improved multiview rendering capabilities.

References:

[1] , D J McCartney, G R Chamberlin, D E Sheat and P Gentry , “Telecommunications Opportunities for 3-D Imaging Systems”, Proc 4th European Workshop on 3-D Television, Rome 1993 pp65-69

[2] C van Berkel, D W Parker and A R Franklin, "Multiview 3D-LCD", Proc SPIE vol 2653 pp32-39 (1996).

[3] C van Berkel and J A Clarke, "Characterization and Optimization of 3D-LCD module design", Proc SPIE vol 3012 pp179-187 (1997)

[4] R Borner, “Autostereoscopic 3D-imaging by front and rear projection and on flat panel displays”, Dispays vol 14 pp39-45 (1993)

[5] P St-Hilaire, S A Benton, M Lucente, J D Sutter, and W J Plesniak, "Advances in Holographic Video," SPIE Proc Vol. 1914 pp188-196 (1993).

[6] D Trayner and E Orr, “Autostereoscopic display using holographic optical elements”, SPIE proc 2653 pp65-74 (1996)

[7] N A Dodgson, J R Moore and S R Lang, “Time-multiplexed autostereoscopic camera system”, SPIE Proc 3012 pp72-82 (1997)

[8] “An Autostereoscopic Real-Time 3D Display System” G Bader, E Lueder and J Fuhrmann, Proc SID Euro-Display96 (Birmingham) pp101-104

[9] "Virtual Camera Movement; The Way of the Future?", Dayton Taylor, American Cinematographer, Sept 1996 pp93-100

[10] “VRML 2.0 Source Book”, 2nd Ed. A L Ames, D R Nadeau, J L Moreland. John Wiley & Sons, Inc, New York 1997 ISBN 0-471-16507-7

[11] Eskut et al, “The CHORUS experiment to search for ν_μ to ν_τ oscillation”, Nucl. Instr. and Meth. A401 pp7-44 (1997)

[12] “OpenGL Programming Guide”, J Neider, T Davis and M Woo, Addison-Wesley, Reading 1993 ISBN 0-201-63274-8

[13] “The Inventor Mentor”, Josie Wernecke, Addison-Wesley, Reading 1994 ISBN 0-201-62495-8

[14] “The Visualization Toolkit”, Will Schroeder, Ken Martin and Bill Lorensen, Prentice Hall Inc Upper Saddle River, 1998 ISBN 0-13-954694-4