Augmented Virtuality:
A method to automatically augment virtual worlds with video images
1.0 Introduction

In this paper we will present research work done with the purpose to create augmented virtual worlds in an automatic way with textures generated from real world images. The virtual world will have some of the appearance of the real world but maintain the flexibility of the virtual world. Objects can be manipulated in a way that the real world does not allow, for example the objects are not dependent on physical laws, which can be changed to the needs of the user. Also irrelevant parts of the real world can be left out to give the user a more easily understood environment. Irrelevant parts will not disturb or confuse the user.

We will look at a method of how to put textures from a video image on objects in an automatic way. From a video camera, images are taken, texture pieces are extracted and put on an object in the virtual world. In this way it is possible to build up a virtual room from a real room by sweeping the camera over the scene and generate textures. Some applications for this technology are in the area of telepresence and tele-exploration. An example of an application can be tele-surveillance. A guard can have a virtual model of the place to guard where the most important objects are highlighted and have an updating texture. Other nonessential parts can be left out. Another example is for firemen equipped with smoke helmets. A head up display in the helmet can show the fireman a virtual model of the scene in front of him and with a heat sensitive camera attached to the fireman each object can have a texture that reflects its temperature. This allows the fireman to see through the smoke and know areas which are dangerous to him.

A number of reasons exist for using video images in a virtual world. For example, if you want to achieve a realistic world, the first reason is the same as for using textures, it gives the virtual world a more "natural" looking appearance. Worlds and objects are built up by polygons that form the objects. To make a complex object without texture a great number of polygons are needed and this gives also the virtual world a sharp look, hard edges and clean areas. This is not the way the real world looks. By using textures the virtual world looks more natural and fewer complex objects are needed since a great deal of complexity is in the texture. A brick wall is a perfect example of this, it can be easily built up with one polygon with a brick texture instead of modelling each brick and the concrete between them. A second reason for using real video images as textures are that the user is very familiar with the real world and can therefore immediately identify the virtual objects as he knows their real counterpart.

In the chapter 2 we will define the problem and the idea how to solve it. In chapters 3 to 9 the basic concepts and some necessary theory will be explained. The results achieved during the work will then be explained and finally the application which has been developed will be explained with a user and systems manual in chapters 13 and 14.
2.0 Problem definition and solution idea

The main purpose of this work is to make an application that takes images from a video camera and generates textures to be placed on objects in a virtual world. The application should be implemented in the DIVE system [1]. The Distributed Interactive Virtual Environment (DIVE) system is a software platform for distributed multiuser virtual environments that has been developed by DCE (Distributed Collaborative Environments) group at SICS (Swedish Institute of Computer Science).

We make the assumption that the virtual model is a fairly good model of the real world. If this is true and we know how a point in the 3-dimensional world is projected onto the video image and where the camera is situated in the real world it would be possible to predict where in the video image different objects will appear. We will know which part of the video image that should be the texture of a certain object.

We realize this is a strong assumption but the main part of the research is in the methodology and tests are done in cases where this assumption should hold for the basic structure of the room. Updating the model and other synchronisation problems are beyond the scope of this paper.

The objects in the virtual world that will take video image textures are called ‘Windows On the World’ or WOW since they are small windows in the virtual world that allows the visitor look out into the real world.

Points projected to the image plane through the camera will be affected by both distortion in the camera system and perspective transformation. There for the theory of point transformations in 3-dimensional coordinate systems and between 3-dimensional and 2-dimensional systems are necessary. We will also need a camera model that describes how points in a 3-dimensional system will be distorted when transformed to the 2-dimensional image plane. As explained in chapter 11.0 ’Results’ we also need the theory for image resampling.
3.0 Virtual worlds

The term virtual reality (VR) is used for a great deal of different situations and can spread from for example text-only adventure games to totally immersive computer generated virtual environments. A good definition of VR is given in [2]. The authors define a VR environment as “one in which the user is totally immersed in a completely synthetic world, which might mimic some properties of the real world but it can also exceed the bounds of physical reality”. The difference between VR systems is the amount of immersion which stretches from those displayed on an ordinary computer monitor to systems using helmets with a large field of view and a body suit with sensitivity feedback.

Another not yet so wide spread term that is also used in a numerous of ways by different people is augmented reality which is a mix of reality and VR. An example of a augmented reality system is a repair tool for printer servicemen. Through a pair of glasses the serviceman can see what parts of a printer that is non-functional by overlaying graphics over the real scene. The real scene is augmented with information about the printer errors. When VR is enhanced with real images one talks of augmented virtuality.

There is no sharp borders between the concepts of reality, augmented reality, augmented virtuality and virtuality, instead it can be seen as a continuum spreading from totally real and totally virtual, as proposed by P. Milgram in[3]. The continuum starts at reality, spreads through augmented reality and augmented virtuality to virtuality as shown in figure 1. Everything between total reality and total virtuality is called mixed reality and includes both augmented reality and virtuality.

Augmented reality is reality enhanced with some computer graphics. The main part is real and the computer graphics only enhance for example some properties. In augmented virtuality real images are brought into the virtual world as an example a texture of some real object.

FIGURE 1. The mixed reality continuum.
4.0 The platform

A virtual environment is a realtime simulation of a real or imaginary scene where users can navigate and interact. A fully interactive multi-user virtual environment allows users over a network to collaborate, meet and work within the virtual world. DIVE (Distributed Interactive Virtual Environment) is such a system, an experimental development software platform for development and research with 3-dimensional virtual environments. It is an internet-based multi-user system where participants can navigate in 3D space, see, meet and interact with other users and applications. The first DIVE version appeared in 1991 and the platform has developed since then. The latest version is 3.1.0 which was released summer 1996.

A participant in a DIVE world is called an actor, and is either a human user or an automated application process. The actor can browse a world through a rendering application called a visualizer (the default is currently called Vishnu). The visualizer renders a scene from the viewpoint of the actor’s eye.

One of the great advantages of DIVE is the DIVE/Tcl[4] interface, through which different behaviours can be added easily to objects. This allows the users to interact with objects in the virtual world. Interfaces to other applications or real world objects can be easily built and brought into the virtual world.

DIVE supports two different methods for distributing textures over the network to other DIVE applications, filename distribution or image distribution. It is necessary to distribute the textures to let other users see them. The distribution of the textures filename sends only the name of the file over the network and therefore it works only when all applications share the same disk. The file must be in the DIVEPATH, which is a UNIX environment variable that tells DIVE where to look for files, see the DIVE manual[5] for more information. The image distribution method sends the whole image over the network.

Different graphics libraries are used to render the view in the visualizer. DIVE uses different graphics libraries on different platforms and for Silicon Graphics computers DIVE supports two different graphics library systems, gl and OpenGL[6]. In the future it will only be OpenGL on all platforms.
5.0 Textures

To give a 3D scene or virtual world a more natural look, one can use textures. A texture is a bitmap that is mapped onto a surface of an object, like painting the surface with the bitmap. There are numerous ways how to map the texture onto the 3D surface of the object.

In DIVE which uses a rather simple method but in the same way very flexible, the mapping is controlled by texture vertices. The method is the same as the VRML1.0[7] (Virtual Reality Modelling Language) standard for texture-mapping. VRML1.0 defines the lower left corner of the bitmap as the origin in the texture coordinate system and the point (1,1) is the upper right corner. Each vertex of a polygon is given a coordinate in the texture coordinate system, a texture vertex. The texture will be mapped on to the surface so that each texture vertex is mapped to its corresponding surface vertex. If a texture vertex has a coordinate that is outside the texture, a value less than 0 or greater than 1, the texture will be repeated, see figure.

FIGURE 2. The left polygon has texture repeated vertically, the upper texture vertices has a y-value of 2. The right polygon has the original bitmap as a texture.
6.0 The robot

We have a video camera mounted on top of a mobile robot. The robot is model B21 from Real World Interface Inc.[8] The B21 is designed specially for research and development and it is therefore a robust construction with an easy way to add peripheral devices and it has a variety of sensors.

The robot has three distinct parts, the lower part or the base, the middle part or the enclosure and the top, the console, see figure.

![FIGURE 3. The RWI robot.](image)

The base section, which does not rotate as the robot moves around, holds equipment for driving and steering. The power unit for the whole robot is also contained in the base. Four synchronously steered and driven wheels let the robot move around. A processor called MCP (Motion Control Processor) frees the main CPU in the enclosure from the task of managing motion and position which is done by dead reckoning.

The enclosure holds the main CPU which is a 486 based PC which runs UNIX. The computer communicates with the base and can get information from a large number of sensors spread all over the robot. The computer can also be connected to a network through an ethernet connection.

The console which is the topmost section of the robot can hold a notebook for an easy interface to the robot. On the console different peripheral equipment can be mounted such as a video camera.

A DIVE steering and control device has been developed for the B21 robot[9]. A model of the robot will appear in the virtual environment and the user can easily steer the robot around in the real world and see the model move simultaneously around in the same way in the virtual world. By clicking on the robot the user opens up the control window with four buttons to steer the robot forward, backward, left and right.
7.0 Transformations

In this chapter the fundamentals of transformations will be presented. We will look at how to transform points between 3D coordinate systems. This is needed to transform object points to the world coordinate system and then to the coordinate system of the camera. We will also look at how to transform points from 3D to 2D systems through perspective projection.

7.1 3D to 3D

A point \( P = (x, y, z) \) in a 3D coordinate system can be transformed in many different ways, but we will just look at the three basic transformations, translation, scaling and rotation. To translate a point to a new position we add the translation in the different directions of the coordinate axis \( T_x, T_y, T_z \) to the point.

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} = \begin{bmatrix}
    x \\
    y \\
    z
\end{bmatrix} + \begin{bmatrix}
    T_x \\
    T_y \\
    T_z
\end{bmatrix} = P + T \tag{7.1.0.1}
\]

To translate an object we could apply equation 7.1.0.1 to every point of the object. Since each line of an object is built up by an infinite number of points this would take an infinitely long time, but since translation is a linear operation it is sufficient to translate the endpoints and draw lines afterwards between them. This is also true for scaling and rotation.

A point can be scaled (stretched) along the axis by multiplying the coordinate with a scale factor.

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} = \begin{bmatrix}
    s_x & 0 & 0 \\
    0 & s_y & 0 \\
    0 & 0 & s_z
\end{bmatrix} \begin{bmatrix}
    x \\
    y \\
    z
\end{bmatrix} = S \cdot P \tag{7.1.0.2}
\]

If \( s_x = s_y = s_z \) the scaling is said to be uniform and if they are not equal it is said to be differential. Notice that the scaling is about the origin and a object not centred will move towards the origin when scaled to smaller size and away if scaled to larger size.

For translation and scaling there is no difference between a right-handed and a left handed system however for rotation there is a difference. From now on we will assume that we always have a right-handed systems if nothing else is stated. The definition of positive rotation in a right-handed system is such that when looking from the positive part of an axis towards the origin positive rotation is in the counterclockwise direction. A rotation of \( \gamma \) around the z-axis is
This can easily be verified, the x-axis, $[1 \ 0 \ 0]^T$, will result in the y-axis, $[0 \ 1 \ 0]^T$, when rotated 90° around the z-axis.

The x-axis rotation matrix is

$$
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 \\
  0 & \cos \alpha & -\sin \alpha \\
  0 & \sin \alpha & \cos \alpha
\end{bmatrix} \cdot \begin{bmatrix}
  x \\
  y \\
  z
\end{bmatrix} = R_x \cdot P
$$

The y-axis rotation matrix is

$$
\begin{bmatrix}
  x \\
  y' \\
  z'
\end{bmatrix} = \begin{bmatrix}
  \cos \beta & 0 & \sin \beta \\
  0 & 1 & 0 \\
  -\sin \beta & 0 & \cos \beta
\end{bmatrix} \cdot \begin{bmatrix}
  x \\
  y \\
  z
\end{bmatrix} = R_y \cdot P
$$

These three rotation matrices can be combined to one matrix

$$
\begin{bmatrix}
  r_{11} & r_{12} & r_{13} \\
  r_{21} & r_{22} & r_{23} \\
  r_{31} & r_{32} & r_{33}
\end{bmatrix} = R
$$

Where

$$
\begin{align*}
  r_{11} &= \cos \gamma \cos \beta \\
  r_{12} &= \sin \gamma \cos \alpha + \cos \gamma \sin \beta \sin \alpha \\
  r_{13} &= \sin \gamma \sin \alpha - \cos \gamma \sin \beta \cos \alpha \\
  r_{21} &= -\sin \gamma \cos \beta \\
  r_{22} &= \cos \gamma \cos \alpha - \sin \gamma \sin \beta \sin \alpha \\
  r_{23} &= \cos \gamma \sin \alpha + \sin \gamma \sin \beta \cos \alpha \\
  r_{31} &= \sin \beta \\
  r_{32} &= -\cos \beta \sin \alpha \\
  r_{33} &= \cos \beta \cos \alpha
\end{align*}
$$
We say that a matrix with the following properties

1. Each row is a unit vector.

2. Each row is perpendicular to the others (their dot product is zero).

3. It has a determinant of 1.

is called \textit{special orthogonal} and will preserve angels and lengths. Transformation composed of an arbitrary sequence of special orthogonal matrices is called \textit{rigid-body transformation} because the body or object being transformed is not distorted in any way. Rotation and translation are rigid-body transformations. An \textit{affine transformation} is one that preserves parallelism, but not lengths and angels. Scaling and any arbitrary sequence of rotation, translation and scaling are affine transformations. The perspective transformation, which we will look at in chapter 7.2 ‘3D to 2D, or projections’, is not affine.

Rotation and scaling is done by multiplying but translation is done as an addition. We would like to be able to treat all three transformations in a consistent way so they can easily be combined. If we express the coordinates as homogeneous coordinates all three transformations will be multiplications. Homogenous coordinates were first developed in geometry and have been applied subsequently in graphics. Numerous graphics subroutine packages and display processor work with homogenous coordinates.

Homogeneous coordinates has an fourth coordinate added to each point and points will be represented as a quadruple \((x, y, z, w)\). Two sets of homogeneous coordinates \((x, y, z, w)\) and \((x', y', z', w')\) represent the same point if and only if one is a multiple of the other. As an example \((2, 3, 4, 5)\) and \((4, 6, 8, 10)\) represent the same point. At least one of the homogeneous coordinates must be nonzero and points with \(w = 0\) is called points at infinity. If \(w \neq 0\) we can divide with it and get the cartesian coordinates of the homogeneous point \((x/w, y/w, z/w, 1)\).

Translation in homogeneous coordinates will be

\[
\begin{bmatrix}
x'
y'
z'
1
\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
\]

and scaling along the axes

\[
\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
\]
and finally rotation

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{bmatrix} =
\begin{bmatrix}
  r_{11} & r_{12} & r_{13} & 0 \\
  r_{21} & r_{22} & r_{23} & 0 \\
  r_{31} & r_{32} & r_{33} & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

7.1.0.10

where the \( r_{xx} \) is the same as in equation 7.1.0.6. When combining transformations it is important to notice the order in which they are applied. Notice that there is a difference in first rotating an object and then translating it compared to first translating and then rotating.

7.2 3D to 2D, or projections

To transform a point in a 3D coordinate system to a 2D system the point is projected onto a 2D plane. Projection, as the model of imaging systems, is also described in chapter 9.1 'Pin hole camera model'. In this chapter we will derive a 4x4 matrix to project homogeneous coordinates. The 4x4 matrix allows us to combine the projection with any transformation which is convenient. To project a 3D point \((x, y, z)\) onto a 2D projection plane, \((x_p, y_p, d)\) at distance \(z = d\) and normal to the z-axis we use similar triangles as in chapter 9.1 'Pin hole camera model'.

\[\text{FIGURE 4. Perspective projection a) view along the y-axis and b) along the x-axis.}\]

From figure we get the following ratios

\[
\frac{x_p}{x} = \frac{d}{z} \quad \frac{y_p}{y} = \frac{d}{z}
\]

7.2.0.1

which we can rewrite as

\[
x_p = \frac{x}{z/d} \quad y_p = \frac{y}{z/d}
\]

7.2.0.2
The distance \( d \) is just a scale factor. It is the division with \( z \) that yields the perspective, distant objects becomes smaller than closer ones. All values of \( z \) are allowed except \( z=0 \). Points can be behind the centre of projection on the negative \( z \) axis or between the centre of projection and the projection plane. The projection can be expressed as a 4x4 matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix} = M_{per}
\]

Transforming the point \( P = (x, y, z, 1) \) with \( M_{per} \) yields the homogeneous point

\[
\begin{bmatrix}
x \\
y \\
z \\
1/d
\end{bmatrix}^T
\]

Dividing with \( W \) and dropping the fourth coordinate will give the 3D coordinate

\[
\left( \frac{x}{W}, \frac{y}{W}, \frac{z}{W} \right)
\]

Which is the same as equation 7.2.0.2.

Orthographic projection, is a special case of the perspective projection when the distance between the projection point and the plane tends to infinity. All objects are projected onto to a 2D plane with no distortion of their \( x \) and \( y \) coordinates. The projection plane is located at \( z=0 \) and the projection is straightforward, point \( P \) project as

\[
x_p = x  \quad y_p = y  \quad z_p = 0
\]

and expressed as a matrix

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} = M_{ort}
\]
8.0 Image Processing

To get an image of a scene into the computer memory from a camera the signal from the camera must be converted to a sampled signal in an A/D-converter and quantized to fit the computer. This image acquisition can be divided into three distinct steps where each step will affect the image in a specific way. It can be seen as three cascaded filters applied to the input image.

![Image acquisition system](image)

Due to the point spread function, PSF, of the camera, \( h(x, y) \), the output from the camera is a degraded version of the scene image. If the PSF is identical in all directions it is said to be rotationally-symmetric and furthermore if it has the same shape through the whole image it is said to be spatially-invariant. If the two-dimensional PSF can be divided into two one-dimensional filters, \( h(x, y) = h_x(x, y) \cdot h_y(x, y) \), it is said to be separable. In practice the PSF is not usually separable, spatially-invariant and rotationally-symmetric and as a result most imaging systems induce geometric distortion in addition to blurring.

The degraded image is bandlimited due to the PSF, the high frequency components are attenuated. Since visual detail directly corresponds to spatial frequency it follows that the degraded image will have less detail.

Due to the Nyquist criterion the signal that should be sampled must be bandlimited to \( f_s/2 \), where \( f_s \) is the sampling frequency, to avoid aliasing. The sampling process will give a non-bandlimited signal.

To reconstruct the image the sampled signal should be filtered with an ideal lowpass filter which is the same as convolving the signal with a sinc-function.

8.1 Image Resampling

The process of transforming an image from one coordinate system to another is called image resampling. The transformation between the two coordinate systems is described by a transformation function. Each pixel is transformed in the same way as described in 7.0 ’Transformations’. This gives us two possibilities to calculate the new image, transform every point in the new image back to the coordinate system of the original image and read the value of each pixel or transform the original image to the new coordinate system. Due to the fact that images are build on integer lattices both of these methods give problems. As for the last method the pixel from the original image is not guaranteed to fit into the pixels of the new image and therefore we can not say if all pixels in the new image have a value. The first method gives us a similar problem,
when transforming the integer positions in the new image to the original coordinate system these will not generally coincide with the integer lattice of the original image. The solution to this problem is to build up the image as a continuous function rather than a discrete function, to reconstruct the image. To get the new image we transform the integer lattice of the new image to the original image and use it to sample. The process is divided into two steps, image reconstruction and sampling.

8.1.1 Image Reconstruction

To reconstruct an image the optimal solution is to convolve the sampled data with the sinc function as stated in 8.0 ‘Image Processing’. But the sinc function is not a spatially limited function and the convolution will be an infinite sum. Instead one uses an interpolation function to rebuild the original image. This will not give the exact original image, rather a filtered version of the original but it will be continuous. The interpolation can be done in several ways and we will study the two most simple ones, which also are the two methods OpenGL and gl uses to resample textures.

Nearest neighbour

This is the simplest interpolation method seen from computational standpoint. Each interpolated pixel is given the value of the nearest pixel in the original image. The interpolation can be achieved by convolving the image with a one-pixel wide rectangle in the spatial domain. The two-dimensional interpolation function is defined as:

\[
p_n(x, y) = p_0(x) \cdot p_0(y)
\]

where

\[
p_0(x) = \begin{cases} 
1 & 0 \leq |x| < 0.5 \\
0 & 0.5 \leq |x|
\end{cases}
\]

The fourier transform of the rectangle function is the sinc function and therefore this interpolation has poor frequency response relative to the ideal lowpass filter.

Bilinear interpolation

With this method straight lines are adapted through the pairs of neighbours, see figure.

FIGURE 6. Linear interpolation of four pixel values.
The pixel values of the four neighbouring points are \( f(x_0, y_0) \), \( f(x_1, y_0) \), \( f(x_0, y_1) \) and \( f(x_1, y_1) \). A straight line through the two points at \( y = y_0 \) is given by the following equation:

\[
f_{x_0}(x, y_0) = f(x_0, y_0) + \frac{f(x_1, y_0) - f(x_0, y_0)}{x_1 - x_0} \cdot (x - x_0)
\]

8.1.1.3

The line through the points at \( y = y_1 \) is given by:

\[
f_{x_1}(x, y_1) = f(x_0, y_1) + \frac{f(x_1, y_1) - f(x_0, y_1)}{x_1 - x_0} \cdot (x - x_0)
\]

8.1.1.4

A straight line between these two lines at an arbitrary \( x \) coordinate is given by:

\[
f(x, y) = f_{x_0}(x, y_0) + \frac{f_{x_1}(x, y_1) - f_{x_1}(x, y_0)}{y_1 - y_0} \cdot (y - y_0)
\]

8.1.1.5

which is our interpolation function. For an image where the distance between the points is constant and equal to 1, the interpolation function will be:

\[
f(x, y) = f_0 + (f_2 - f_0)\Delta y + (f_1 - f_0)\Delta x + (f_3 + f_0 - f_1 - f_2)\Delta x\Delta y
\]

8.1.1.6

where \( f_0 = f(x_0, y_0) \), \( f_1 = f(x_1, y_0) \), \( f_2 = f(x_0, y_1) \) and \( f_3 = f(x_1, y_1) \). Furthermore \( \Delta x = (x - x_0) \) and \( \Delta x = (x - x_0) \).

In the spatial domain linear interpolation is equal to convolving the samples with the following interpolation kernel:

\[
p(x, y) = p_1(x) \cdot p_1(y)
\]

8.1.1.7

where

\[
p_1(x) = \begin{cases} 1 - |x| & 0 \leq |x| < 1 \\ 0 & 1 \leq |x| \end{cases}
\]

8.1.1.8

This corresponds to a reasonably good low-pass filter in the frequency domain. Its response is superior to that of the nearest neighbour, in particular the side lobes are far less prominent, indicating improved performance in the stopband. Nevertheless, a significant amount of spurious high-frequency components leak into the passband resulting in aliasing. Also the passband is moderately attenuated, which gives smoothing. Linear interpolation is the most widely used interpolating algorithm for reconstruction since it produces a fairly good result at reasonable cost.
In figure 7 the two algorithms are compared. They are produced from a square grid when a quadrangle with four arbitrary vertices is mapped onto a square. It can be seen that the nearest neighbour algorithm produces a rough image and that the image from the bilinear algorithm is much smoother.

**FIGURE 7.** Difference between a) nearest neighbour and b) bilinear interpolation.
9.0 Camera

To be able to predict where in the video image objects will appear we must come up with a model for the camera and calibrate the parameters for this model.

9.1 Pin hole camera model

Point projection as described in chapter 7.2 ‘3D to 2D, or projections’, is the fundamental model for the transformation wrought by imaging systems such as our eye, a camera and numerous other devices. In a first order approximation these systems behave as a pin hole camera. The scene is projected through one single point onto an image plane see figure.

![FIGURE 8. An image projected through one single point.](image)

The image plane in figure 8 lies behind the point of projection and the resulting image is reversed. A more intuitive way of doing this is to recompose the geometry so that the point of projection corresponds to a viewpoint behind the image plane where the observer is thought to be. This also results in a right side up image, see figure.

![FIGURE 9. An image projected on the image plane infront of the viewpoint.](image)
The mathematics is still the same.

The camera coordinate system \( c - x_c y_c z_c \) is a right-handed 3-D coordinate system with the z-axis pointing in the viewing direction. The image coordinate system \( i - x_i y_i \), has the same y-axis as \( c - x_c y_c z_c \) but the x-axis points in the opposite direction of the \( c - x_c y_c z_c \) x-axis. The view point is positioned at the origin of the camera coordinate system. The image is located at \( f \), also called the focal length, and consists of the plane \( z_c = f \) upon which the scene is projected. The intersection point of the optical axis, the camera z-axis, and the image plane is called the principal point. As an object approaches the viewpoint its projection becomes larger and larger. The projection is the same as in 7.2 '3D to 2D, or projections’ and this gives us

\[
\begin{align*}
  x_i &= -f \frac{x_c}{z_c} \\
  y_i &= f \frac{y_c}{z_c}
\end{align*}
\]

We get a minus sign in the first equation because we project points to the image coordinate system which has its x-axis pointing in the opposite direction of the x-axis in the camera coordinate system.

### 9.2 Calibration

An image that is taken of a scene through a camera is a distorted model of the real scene. The most obvious effect comes from the perspective projection to the 2D image plane but there is also distortion from the camera lens, the CCD array, to mention some factors. Our camera model must also take care of these distortions. The parameters in the camera model cannot be measured easily and therefore it is necessary to make camera calibration, to calibrate the parameters that affect the image. This has been studied intensively during the last two decades in both the photogrammetry and computer vision communities. Basically one can divide it in to two different kinds of parameter calibrations, calibration of extrinsic and intrinsic parameters.

The extrinsic parameters, external parameters, belong to the setup of the camera, it is the estimation of camera position, the rotation and translation between the coordinate system of the camera and the 3D world coordinate system.

The intrinsic parameters, internal parameters, include the optical and electronic properties of the camera such as focal length, principal point, lens distortion, the aspect ratio of the pixel array etc.

The ex- and intrinsic parameters must be known all the time. It is difficult, or maybe impossible, to do real time measurements, therefore we must come up with some other methods. If the electrical and optical properties of the camera do not change during or between each run the intrinsic parameters will not change and can therefore be calibrated once.
For a camera which does not move, the extrinsic parameters can also be calibrated once but if the camera moves it is necessary to update these parameters, which can not be done through calibration in realtime. But rotation and position are easily measured units and if we know the absolute position and rotation or the relative changes it will be enough. From our RWI robot we can get accurate position changes from a known start position for a limited amount of time and movement. We use these values to add to the start position achieved through the calibration process.

The following derivation of a calibration algorithm is from a paper written by Mengxiang Li [10].

The transformation of an object point, in world coordinates, to camera coordinate system is given by the following equation.

\[
\begin{bmatrix}
  x_c \\
  y_c \\
  z_c 
\end{bmatrix} = R \begin{bmatrix}
  x_w - T_x \\
  y_w - T_y \\
  z_w - T_z 
\end{bmatrix}
\]

9.2.0.1

where \((x_c, y_c, z_c)\) is the object point in camera coordinate system and \((x_w, y_w, z_w)\) is the point in the world coordinate system. \(T\) is the translation between the two coordinate systems and \(R\) is the rotation matrix as defined in equation 7.1.0.6. By using the expression from equation 9.1.0.1 for the pinhole camera projection we will get the in photogrammetry so called collinearity equations:

\[
\begin{align*}
  x_i &= -\frac{r_{11}(x_w - T_x) + r_{12}(y_w - T_y) + r_{13}(z_w - T_z)}{r_{31}(x_w - T_x) + r_{32}(y_w - T_y) + r_{33}(z_w - T_z)} \\
  y_i &= \frac{r_{21}(x_w - T_x) + r_{22}(y_w - T_y) + r_{23}(z_w - T_z)}{r_{31}(x_w - T_x) + r_{32}(y_w - T_y) + r_{33}(z_w - T_z)}
\end{align*}
\]

9.2.0.2

If we transform \((x_p, y_p)\) into the pixel coordinate system \(p - x_p y_p\) we will get,

\[
\begin{align*}
  x_i &= (x_p + v_x - x_p0 - dx)x_s \\
  y_i &= (y_p + v_y - y_p0 - dy)y_s
\end{align*}
\]

9.2.0.3

\(v_x, v_y\) are errors of the measurement of \(x\) and \(y\), \(x_p0, y_p0\) is the principal point, \(s_x, s_y\) are the scale factors of the pixel system in \(x\) and \(y\) directions respectively and \(dx, dy\) are the lens distortion components which consist of two parts, the radial and the tangential distortions.

\[
\begin{align*}
  dx &= dx_r + dx_t \\
  dy &= dy_r + dy_t
\end{align*}
\]

9.2.0.4

We use two models which are often used in photogrammetry.
\[ dx_r = (x - x_{p0})(a_1 r^2 + a_2 r^4 + a_3 r^6) \]
\[ dy_r = (y - y_{p0})(a_1 r^2 + a_2 r^4 + a_3 r^6) \]

\[ x_i = [p_1(r^2 + 2(x - x_{p0})^2) + 2p_2(x - x_{p0})(y - y_{p0})](1 + p_3 r^2) \]
\[ y_i = [p_2(r^2 + 2(y - y_{p0})^2) + 2p_1(x - x_{p0})(y - y_{p0})](1 + p_3 r^2) \]

\[ a_1, a_2, a_3 \] are the radial lens distortion parameters, \( p_1, p_2, p_3 \) are the tangential distortion parameters and \( r = \sqrt{(x - x_{p0})^2 + (y - y_{p0})^2} \) is the radial distance from the principal point.

Substituting equation 9.2.0.3 and 9.2.0.4 into equation 9.2.0.2 and let \( f_x = \frac{f}{s_x} \) and \( f_y = \frac{f}{s_y} \) we get the following:

\[
\begin{align*}
   x + v_x &= x_0 + dx_x + dx_r + f_x \left( r_{11}(x_{u-T_x}) + r_{12}(y_{u-T_y}) + r_{13}(z_{u-T_z}) \right) = P(\Phi) \\
   y + v_y &= y_0 + dy_y + dy_r - f_y \left( r_{31}(x_{u-T_x}) + r_{32}(y_{u-T_y}) + r_{33}(z_{u-T_z}) \right) = Q(\Phi)
\end{align*}
\]

where \( \Phi \) is the parameter vector:

\[ \Phi = [x_0, y_0, a_1, a_2, a_3, p_1, p_2, p_3, f_x, f_y, T_x, T_y, T_z, \alpha, \beta, \gamma]^T \]

This can be solved by a least squares technique by minimizing \( \sum_{i=1}^{n} (v_{x_i}^2 + v_{y_i}^2) \), where \( n \) is the number of calibration points. \( P(\Phi) \) and \( Q(\Phi) \) are non-linear functions of \( \Phi \) and the minimization is therefore a non-linear optimization problem. One way to solve it is to linearize \( P(\Phi) \) and \( Q(\Phi) \) with some initial value \( \Phi_0 \) and solve for \( \Delta \Phi \).

Then by adding \( \Delta \Phi \) to \( \Phi_0 \) as a new initial value and repeat the process until some convergence criterion is fulfilled.

This algorithm is implemented in a program called ‘camcal’ by Mengxiang Li at CVAP, KTH.
10.0 The Application

To test the idea we have written an application in the DIVE system. The program is a DIVE process that captures images from a camera and generates textures which are distributed out in the virtual environment. Here follows a short description of the methods employed in the application, for readers who are more interested using the application can read chapter 13.0 'User Manual' and chapter 14.0 'System Manual' or get in contact with the author.

When starting the program it reads a DIVE entity description file that defines the camera object. An actor is associated with the camera object and the actor is given a tcl/tk interface [11]. Several properties in the camera object specify the camera parameters that have been achieved through camera calibration with the ‘camcal’-program. More, some properties specify how the program will behave. The program connects to the video device and grabs the first image and calculate the textures of the ‘windows on the world’ objects and distributes them. This procedure will continue until the program has ended.

The operations of the program can be divided in two different parts, the video image grabbing and the texture generation.

To get images over a room or scene we use an ordinary CCD video camera. We have used two different cameras, one Canon Ci-20P with a 8.5 mm 1:1.5 lens and one ELMO EM-102 PAL with a 7.5 mm 1.1.6 lens. These are ordinary off-the-shelf cameras with an ordinary BNC video out connector. To grab the images into the computer we used a video board from Silicon Graphics(SGI) on an Indigo2 machine and the VL video library. The VL is a collection of device-independent C language calls for Silicon Graphics workstations equipped with video options and give you the ability to blend graphics and video, digitize video, output graphics as video and more.

To generate textures it was necessary to write a own function, see 11.0 'Results', that warps an image using either nearest neighbour or linear interpolation. If a part of the texture is not seen by the camera it will be transparent. This gives the possibility to have several cameras and applications running. A layer of WOW's can be made and if a part of a texture is seen by one camera and not by another, the texture will be seen through the others even if it is behind them. A window is opened up on start-up and part of it will become a graphical area where the video image will be drawn and wire-frames of the different WOWs will be drawn over the video image. This is a form of augmented reality, some real objects, the WOWs, are emphasized to the user. To see the augmented virtuality a DIVE visualizer like vishnu is needed to browse the virtual world with the generated textures.
11.0 Results

An application that embodies the ideas in this paper has been developed, though some problems with camera calibration still exists. The application is not dependent on the calibration and this part can be done and added later. It was not possible to do this within the time limit of the work. Figure shows a result of the application.

FIGURE 10. The DCE lab at SICS as a virtual model with WOWs. The windows, the computer screen and a whiteboard are WOWs.

The way to generate the textures can be divided into two different solutions. For the first one the whole video image is used as the texture bitmap. Each vertex of a WOW surface is transformed to the image plane. These points can now be used as texture vertices and the graphics library will take care of the transformation of the image. In the other solution a new image is generated and will be used as a texture. As explained in chapter 11.2 'Texture generation.' the last one is the only possible solution for the graphics library we use.

By using the first method we calculate the texture coordinates in the video image for each object and distribute the texture vertices and the whole image. The advantage of this method is that it becomes fast because we do not have to calculate the new texture, instead it is taken care of by the graphics library, which will do it in hardware if it is possible. One disadvantage is that the parts of the WOW which are not seen by the camera will have the wrong texture. The part of the WOW which is not seen will have a texture belonging to some other part of the video image because it is impossible to turn off the texture repetition.

The second method where a new texture is produced for every object and then distributed has also some advantages and disadvantages. The advantages are that the process when producing the texture can be controlled, the parts of the texture that are not seen by the camera can be controlled and the resolution of the texture can be controlled. As shown in chapter 11.2 'Texture generation.', the first one is the most important, and due to the graphics library problem this is the only possible method. A disadvantage is that when many object will have textures a great deal of computation is needed and a lot of data is sent over the network.
11.1 Calibration

The camera calibration was done at CVAP KTH where they have a calibration cube, a special cube with lines in perpendicular directions on each face, see figure b). The calibration worked very well and the parameters for both cameras seemed very good, see table 1. The parameters has the same notations as in 9.2 'Calibration'.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Canon Ci-20P</th>
<th>Precision</th>
<th>ELMO EM-102 PAL</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>-4.208168e-40</td>
<td>9.83605e-25</td>
<td>-1.219067e-40</td>
<td>7.79496e-25</td>
</tr>
<tr>
<td>a2</td>
<td>-2.017732e-50</td>
<td>8.92567e-35</td>
<td>-4.650947e-51</td>
<td>7.07349e-35</td>
</tr>
<tr>
<td>p1</td>
<td>+1.208334e-32</td>
<td>1.03255e-19</td>
<td>-1.096666e-33</td>
<td>8.18284e-20</td>
</tr>
<tr>
<td>p2</td>
<td>-4.992338e-33</td>
<td>1.03255e-19</td>
<td>-1.608381e-33</td>
<td>8.18284e-20</td>
</tr>
<tr>
<td>p3</td>
<td>+3.339209e-59</td>
<td>1.03255e-19</td>
<td>+1.051780e-62</td>
<td>8.18284e-20</td>
</tr>
<tr>
<td>fx</td>
<td>902.6474(pix)</td>
<td>3.71835(pix)</td>
<td>773.3572(pix)</td>
<td>3.94400(pix)</td>
</tr>
<tr>
<td>fy</td>
<td>981.1244(pix)</td>
<td>3.92064(pix)</td>
<td>847.9290(pix)</td>
<td>4.29444(pix)</td>
</tr>
<tr>
<td>x0</td>
<td>393.6012(pix)</td>
<td>2.66463(pix)</td>
<td>380.8698(pix)</td>
<td>2.55839(pix)</td>
</tr>
<tr>
<td>y0</td>
<td>294.4288(pix)</td>
<td>2.87784(pix)</td>
<td>276.9079(pix)</td>
<td>3.22069(pix)</td>
</tr>
<tr>
<td>Tx</td>
<td>446.2929(obj)</td>
<td>2.48740(pix)</td>
<td>423.2658(obj)</td>
<td>2.81139(pix)</td>
</tr>
<tr>
<td>Ty</td>
<td>941.8604(obj)</td>
<td>2.77146(pix)</td>
<td>901.0929(obj)</td>
<td>3.06401(pix)</td>
</tr>
<tr>
<td>Tz</td>
<td>443.1627(obj)</td>
<td>2.19802(pix)</td>
<td>385.1109(obj)</td>
<td>2.64190(pix)</td>
</tr>
<tr>
<td>alpha</td>
<td>-47.4503(deg.)</td>
<td>0.18798(deg.)</td>
<td>-51.1067(deg.)</td>
<td>0.24238(deg.)</td>
</tr>
<tr>
<td>beta</td>
<td>31.0997(deg.)</td>
<td>0.12955(deg.)</td>
<td>32.2597(deg.)</td>
<td>0.14198(deg.)</td>
</tr>
<tr>
<td>gamma</td>
<td>29.3006(deg.)</td>
<td>0.10777(deg.)</td>
<td>32.8014(deg.)</td>
<td>0.13075(deg.)</td>
</tr>
</tbody>
</table>

The radial and tangential distortions are very small, and at the largest distances these distortions will not even be near the size of a pixel. Therefore to save computation time they are not include in the used camera model because they will not effect the result.

The extrinsic parameters from the calibration can not be used when the setup is changed. To get extrinsic parameters they were calibrated by hand. The position of the camera is measured and the rotation of the camera is managed by setting the camera to be parallel with the floor, vertical to the walls and not rotated around its own axis. The most important parameters are the rotation of the camera around x- and y-axes. The slightest rotation around these axes will make a rather big error. An error in rotation of say 1 degree will give an error in that direction of 1.7 cm per meter in distance, meaning that an object farther away will be more misplaced than one near. If the object is three meters away the position will be 5.1 cm wrong in the direction of rotation.
FIGURE 11. A small rotation error gives a large placement error on a large distance.

It is very hard to estimate the rotation of the camera by just looking at it. A much easier procedure is at start up of the application to set the initial rotations to zero and then rotate the camera until an outline of the object fits the real object, a rather good fitness could be accomplished, see figure.

FIGURE 12. Texture made of a whiteboard, it is cut out as the line indicates. The camera is rotated by hand.

The calibrated parameters works well for only one object in the middle of the video image but with several objects spread out over the video image as in figure problems start to appear. The textures are displaced into the middle of the video image.

FIGURE 13. Several objects in the view. Due to difference in framegrabbers the program will cut out wrong part of the video image. The textures are displaced into the middle of the image.
The resolution of the images from the framegrabber at CVAP is not equal to the one we have at SICS. To be able to use the parameters from the calibration at CVAP, which where in pixel units, the parameters must be unitless. This is accomplished by dividing the \( f_x \) and \( x_0 \) parameters with the image width and \( f_y \) and \( y_0 \) with the height. The coordinates from the equations is unitless and an image will stretch from point \((0,0)\) to \((1,1)\). To get pixel coordinates for the image taken with the SGI video card, the coordinate is multiplied with the width and height of image from this card.

Another problem with different framegrabbers is that they usually have different timings. The time elapsed from the sync-pulse from the video to the framegrabber starts to capture the image may differ a lot. The image from CVAP and one from the SGI are showed in figure.

![FIGURE 14. A video image captured from the a) SGI framegrabber and the b) CVAP framegrabber.](image)

As can be seen there is a big difference in the amount of blank space on the left edge, on the image from the SGI approximately 15 pixels and for the other 75 pixels. This affects the intrinsic parameters. The \( f_x \) and \( x_0 \) parameters are the most affected ones since the images is most affected in the x-direction, but there are blank lines also in the vertical direction in the two images but they are approximately the same for both framegrabbers about one or two lines. To get correct calibration values the camera must be calibrated with the framegrabber to be used.

The calibration cube is very sensitive and can therefor not be transported easily, and it is the same for the computer with the framegrabber. Another solution would be to calibrate with some other calibration object. The ‘camcal’ program allows to specify calibration points in a 3D-coordinate system and their corresponding 2D-coordinates. This gives the possibility to make calibrations without a special object, just some well known points. Due to some unexplainable errors we have not been able to test this feature. Experiments with another program that is build on an algorithm developed by R. Tsai [12] has been done but it did not work well. The program we used that implemented the Tsai algorithm has a lot of restrictions on the placement of the world origin and the camera model is somewhat different. The program does not generate any error messages but the parameters from it are wrong, not at all near the ones that you expect. There was no more time within this project to investigate these things further.
11.2 Texture generation.

As mentioned before there are two possibilities to generate a texture, warp a piece of the video image to a new image or calculate new texture vertices. The last method which will give really fast texture generation if a large number of Wows are used, does not work for OpenGL or gl, due to the way they map textures. Every object is internally divide into triangles. A square for example with the vertices (-0.5,-0.5), (0.5,-0.5), (-0.5, 0.5) and (0.5, 0.5) can be divided into the two triangles (-0.5,-0.5), (0.5,-0.5), (-0.5, 0.5) and (-0.5, 0.5), (0.5,-0.5), (0.5, 0.5). OpenGL or gl does not specify how the this tessellation will be done and it can be different for different implementations. If an image of a grid is used as a texture on a polygon and we want to have it a stretched in one corner the texture vertices would be as specified in table 2. The vertices of the two triangles will have the same texture vertices as those in the square, see table 2.

<table>
<thead>
<tr>
<th>Square</th>
<th>Texture Vertices</th>
<th>Left Triangle</th>
<th>Right Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-0.5,-0.5)</td>
<td>(0,0)</td>
<td>(-0.5,-0.5)</td>
<td></td>
</tr>
<tr>
<td>(-0.5, 0.5)</td>
<td>(0,0.8)</td>
<td>(-0.5, 0.5)</td>
<td>(-0.5, 0.5)</td>
</tr>
<tr>
<td>(0.5, -0.5)</td>
<td>(1,0)</td>
<td>(0.5, -0.5)</td>
<td>(0.5, -0.5)</td>
</tr>
<tr>
<td>(0.5, 0.5)</td>
<td>(1,1)</td>
<td>(0.5, 0.5)</td>
<td></td>
</tr>
</tbody>
</table>

The texture will be mapped on each triangle instead of on the square and this will not give the desired result, see figure .

![Texture Mapping](image_url)

**FIGURE 15.** Texture mapping. a) as OpenGL and gl does it and b) as it would be preferred.

As a result of this we must use image warping.
12.0 Improvements

The application, the result of this work, is a good platform for further projects. The image warping can be implemented for an image processing library that has hardware support for different operations. Such libraries exist for Silicon Graphics computers. This would improve the run-time of the application.

For camera calibration there is a lot that can be done. First would be to make the calibration work for any kind of object and find out how well it works. As a further extension, the calibration can be done on each start up, if not for all parameters but at least the camera position and rotation to find initial values for them. Later, during a run, the position and rotation can be calibrated to check and update the position of the camera.

With edge detection a more robust application could be created and even if the virtual and real worlds do not synchronize, a good result could be achieved. Probably some image processing hardware is needed to achieve an acceptable speed.

The application does not know the orientation of a surface and therefore it cannot have texture on just one side. If the normal to the surface defines which side the texture should be on, the application could be extended to not generate textures if the object is seen from behind.

As the application works now it generates textures for surfaces that are in front of it, objects that have a positive z-coordinate in the local coordinate system for the camera. The application could be improved to only generate textures for objects within the field of view for the camera.

If an object obscures the WOW the application ignores it, this could be changed and the obscured part of the texture could be made transparent. This will slow down the application because for each pixel in the texture it is necessary to find out if it is obscured or not.
13.0 User Manual

The executable program is called ‘video_texture’ and takes several parameters, all specific for this application and are described in table 3. Every other argument is described in the DIVE manual[5] under Configure Options.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-camera xxxx]</td>
<td>string</td>
<td>“camera.vr”</td>
<td>The name of the camera object file.</td>
</tr>
<tr>
<td>[-top xxxx]</td>
<td>string</td>
<td>No top object</td>
<td>The name of the top object if the camera should be a sub-object.</td>
</tr>
<tr>
<td>[-initscript xxxx]</td>
<td>string</td>
<td>“video_texture_init.tcl”</td>
<td>Initialisation script for tk window and user interface.</td>
</tr>
<tr>
<td>[-directory xxxx]</td>
<td>string</td>
<td>“”</td>
<td>The directory where images will be saved. Should be part of DIVE-PATH.</td>
</tr>
<tr>
<td>[-x xxxx]</td>
<td>float</td>
<td>0.0</td>
<td>Start position of the camera relative the top object.</td>
</tr>
<tr>
<td>[-y xxxx]</td>
<td>float</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>[-z xxxx]</td>
<td>float</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>[-rot_x xxxx]</td>
<td>float</td>
<td>0.0</td>
<td>Start rotation of the camera relative the top object</td>
</tr>
<tr>
<td>[-rot_y xxxx]</td>
<td>float</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>[-rot_z xxxx]</td>
<td>float</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>[-size xxxx]</td>
<td>float</td>
<td>1.0</td>
<td>Size of camera object, actually scaling of camera object.</td>
</tr>
</tbody>
</table>

When starting the program it reads the camera object file and creates the camera object in the world. If the object should be a sub object it is created as one. This can be useful when the real camera is attached to a real object that moves around and the same object exist in the virtual world. Also an actor is associated to the camera object, see 13.2 ‘Camera actor’. The program will continue with reading a tcl/tk initialization file to open up a tk-window. In this window menus can be added, graphics drawing be present and so on. The default “video_texture_init.tcl” file opens a window with menus, a graphics area and two message bars. Once this is done the main loop will start and depending on the camera properties it will start capturing video images and producing textures to those WOW’s who should have textures from the camera that the program is using.

13.1 Camera object

The camera object is the representation of the camera in the virtual world and it’s specified in a DIVE entity description file. As default, if nothing else is given as an argument, the program will try to load a file called ‘camera.vr’. This file must be in the DIVEPATH. After the file has been loaded the object is created in the world and distributed. An actor is also associated to the camera object, see 13.2 ‘Camera actor’. Through properties you specify how the camera and the program should work. All camera properties are summarized in table 4. You can change the properties while the program is running and in that way control the program, as an example disconnect the video device by setting the ‘connect’ property to ‘off’.
13.1.1 ‘fx’, ‘fy’, ‘xc’ and ‘yc’ properties

These properties must be set, otherwise the program will terminate. Without them the program can not calculate the textures. These are the camera parameters you get from the camera calibration, see 9.2 ‘Calibration’. ‘fx’ and ‘fy’ has the same notation but ‘xc’ and ‘yc’ is equal to x0 and y0 respectively.

13.1.2 ‘zoom numerator’ and ‘zoom denominator’ properties

These properties specify how the video image should be scaled, or the size of it. The original video size is multiplied with (zoom numerator/zoom denominator). If the zoom numerator is set to 1 and the zoom denominator to 4 and the original video size is 720x576 pixels as for the Indigo2 Video for Silicon Graphics the video image will become 180x144.

13.1.3 ‘width offset’ and ‘height offset’ properties

These properties specify where the actual video image begin, actually for removing the black borders that might appear in the video image due to timing problems. The width offset does not work on the Silicon Graphics Video Library.

13.1.4 ‘continuous capture’ property

Specifies if the program should continuously capture new video images at the rate specified with the ‘minimum rate’ property.

13.1.5 ‘connect’ property

If the connect property is set to ‘off’ the program will not connect to the video and it might be used by other programs. The Silicon Graphics Video Library can handle shared video resources and there for the program can still be connected to the video when it’s used by another program.

13.1.6 ‘draw in window’ property

When this property is set to ‘on’ the program will draw the video image and wire-frames of the WOW’s in the Tk window. No drawing at all, not even window updating will be done when the property is set to ‘off’, which will make the program to run as fast as possible. For Silicon Graphics, the order of how images are placed in memory is different for the OpenGL and Video Library so when this property is set to ‘off’ it will prevent the program from flipping the image, which takes time.

13.1.7 ‘save video’ property

If this property is set to ‘on’ each new video image will be saved to a file. The file format is ppm (portable pix map). The filename will be the camera object name followed by a time, _HH_MM_SS, and a ‘.ppm’ suffix, for example ‘canon_camera_10_31_25.ppm’. The file will be saved in the directory specified with the -directory argument. The default is the current directory ‘’. 
### 13.1.8 ‘minimum rate’ and ‘maximum rate’ property

When the camera object is moving the program will capture new video images at a maximum rate specified with the ‘maximum rate’ property. It’s specified as images per second so a rate of 0.1 will give a new picture every 10 seconds. The ‘minimum rate’ property gives the rate at which the program will capture images when doing continuous capturing.

#### TABLE 4. Camera properties. All properties must be GLOBAL_PROP.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Value</th>
<th>Default value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fx</td>
<td>FLOAT_TYPE</td>
<td>Any nonnegative float</td>
<td>None</td>
<td>Camera fx, see 9.2 'Calibration'</td>
</tr>
<tr>
<td>fy</td>
<td>FLOAT_TYPE</td>
<td>Any nonnegative float</td>
<td>None</td>
<td>Camera fx, see 9.2 'Calibration'</td>
</tr>
<tr>
<td>xc</td>
<td>FLOAT_TYPE</td>
<td>Any nonnegative float</td>
<td>None</td>
<td>Camera fx, see 9.2 'Calibration'</td>
</tr>
<tr>
<td>yc</td>
<td>FLOAT_TYPE</td>
<td>Any nonnegative float</td>
<td>None</td>
<td>Camera fx, see 9.2 'Calibration'</td>
</tr>
<tr>
<td>zoom numerator</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>1</td>
<td>Video zoom numerator.</td>
</tr>
<tr>
<td>zoom denominator</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>1</td>
<td>Video zoom denominator.</td>
</tr>
<tr>
<td>left blank</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>0</td>
<td>The number of columns on the left side of the video image that the program will ignore.</td>
</tr>
<tr>
<td>right blank</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>0</td>
<td>The number of columns on the right side of the video image that the program will ignore.</td>
</tr>
<tr>
<td>top blank</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>0</td>
<td>The number of rows at the top of the video image that the program will ignore.</td>
</tr>
<tr>
<td>bottom blank</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>0</td>
<td>The number of rows at the bottom of the video image that the program will ignore.</td>
</tr>
<tr>
<td>continuous capture</td>
<td>STRING_TYPE</td>
<td>'on'/'off'</td>
<td>'off'</td>
<td>Turn on/off continuous video capturing.</td>
</tr>
<tr>
<td>save video</td>
<td>STRING_TYPE</td>
<td>'on'/'off'</td>
<td>'off'</td>
<td>Turn on/off saving of the video image.</td>
</tr>
<tr>
<td>draw in window</td>
<td>STRING_TYPE</td>
<td>'on'/'off'</td>
<td>'off'</td>
<td>Turn on/off drawing in the Tk window. Video image and wireframe of WOW’s is drawn.</td>
</tr>
</tbody>
</table>
13.2 Camera actor

An actor is associated with the camera object and it will have the same name as the camera object with ‘_actor’ added, for example ‘canon_camera_actor’. The camera actor have a set of new tcl-commands added to the original DIVE tcl commands. All of these start with the prefix ‘wow_’.

13.2.1 wow_register_click

Usage: wow_register_click command ?format?

Register a Tcl command with the next mouse button press event in the graphics area. The ‘command’ will be called with two arguments, the x and y coordinate where the button was pressed. With ‘format’, which is optional, you can specify if the arguments shall be in pixel units or image coordinates stretching from 0 to 1. ‘pixels’ will give pixel coordinates and anything else including not specifying the ‘format’ will give image coordinates.

13.2.2 wow_load_video

Usage: wow_load_video imageURL

Loads an image from the file specified with imageURL. This image will be used for the texture creation. To not overwrite this image with a new one from the video camera disconnect the camera with the ‘connect’ property.

13.2.3 wow_get_video_size

Usage: wow_get_video_size

Returns the size of the image that is used for texture generation. The return value has the following format ‘width height’.

13.2.4 wow_wc2ic

Usage: wow_wc2ic xw yw zw

Transforms the world point (xw,yw,zw) to image coordinates using the camera model. The lower left corner of the image is (0,0,0,0) and the upper right is (1,0,1,0). The format of the return value is ‘x y’.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Value</th>
<th>Default value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect</td>
<td>STRING_TYPE</td>
<td>‘on’/’off’</td>
<td>‘on’</td>
<td>Turn on/off connection to video camera.</td>
</tr>
<tr>
<td>minimum rate</td>
<td>FLOAT_TYPE</td>
<td>Any nonnegative float</td>
<td>0.1</td>
<td>The rate used when continuously capturing video.</td>
</tr>
<tr>
<td>maximum rate</td>
<td>FLOAT_TYPE</td>
<td>Any nonnegative float</td>
<td>2</td>
<td>The rate used when the camera moves.</td>
</tr>
</tbody>
</table>

TABLE 4. Camera properties. All properties must be GLOBAL_PROP.
13.2.5 wow_do_texture


Generates an image by mapping any quadrangle to a quadrate. ‘filename’ is the name of the image file. ‘array’ is an array of four points defining the quadrangle. Each point is an array of two numbers the x- and y-coordinate of the point. The first point defines the point to map to quadrate point (0,0), the second point (0,1), the third point (1,0) and the last point(1,1). With the ‘interpolation’ argument you can decide what kind of interpolation to use, either ‘nearest’ for nearest neighbour interpolation or ‘linear’ for linear interpolation. If not specified, the size of the generated image will be 128 by 128 pixels. The ‘size’ argument should be an array as ‘{width height}’.

13.2.6 wow_get_list_of_wows

Usage: wow_get_list_of_wows

Returns a list of all WOW’s that get textures from the camera object associated with this actor.

13.3 Windows On the World (WOW)

A ‘Window On the World’ is an object in the DIVE world that has some special properties and will be given a texture from the video camera. To make an object a WOW you give the object a property called ‘video texture’ and it should be of string type and the value is the name of the camera which shall give the object textures. The property must be global. You can also specify if the texture should be saved as a file by giving the property ‘save texture’ and set it’s value to ‘on’. It’s also a string type property and should be global. To turn off the saving set the value to ‘off’. If you do not want to update a WOW set the global string type property ‘freeze’ to ‘on’ and to start update set it to ‘off’. The WOW’s can have two more properties, the global integer type ‘x size’ and ‘y size’. These properties specify the size of the texture image in pixels. The default value for both is 128 pixels. See table 5 for a summary of the different properties a WOW object can have.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Value</th>
<th>Default value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>video texture</td>
<td>STRING_TYPE</td>
<td>Camera name</td>
<td>None.</td>
<td>Specifies that this object should be a WOW and the camera to take images from.</td>
</tr>
<tr>
<td>save texture</td>
<td>STRING_TYPE</td>
<td>‘on’/’off’</td>
<td>‘off’</td>
<td>Turn on/off texture image saving.</td>
</tr>
<tr>
<td>freeze</td>
<td>STRING_TYPE</td>
<td>‘on’/’off’</td>
<td>‘off’</td>
<td>Turn on/off texture updating.</td>
</tr>
<tr>
<td>x size</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>128</td>
<td>The width of the texture image.</td>
</tr>
<tr>
<td>y size</td>
<td>INT_TYPE</td>
<td>An integer greater than zero.</td>
<td>128</td>
<td>The height of the texture image.</td>
</tr>
</tbody>
</table>
14.0 System Manual

The program is implemented as a DIVE application and is a part of the DIVE source tree, in the DIVE root directory. The directory that holds all the files is called 'video_texture/' and most of the c-files are placed here, only files that are dependent on a specific architecture are placed elsewhere. The files for handling video are in 'video_texture/video/' and graphics functions in 'video_texture/graphics/'. The files will just briefly be described here, for more complete information look in the files where each function is described.

In the directory ‘video_texture/data’ there are some DIVE and tcl files. The DIVE entity include file for camera objects ‘video_camera.vh’ defines a camera with a tcl interface and a visualisation of the field of view. As an example of how it can be used can be seen in ‘camera.vr’ file. The tcl initialisation file ‘video_texture_init.tcl’ is read by the application and executed at start-up. It opens up the window, specifies menus and the graphics area where drawings will take place. The file ‘wow.tcl’ is a small example how the properties in a wow object can be used. When the user clicks on the wow it sets the property ‘freeze’ to on and do not get any new textures.

To compile the source code go to ‘video_texture/’ directory and execute gmake. An executable file called ‘video_texture’ will be generated. See DIVE manaul[5] how to retrieve the DIVE source and install it.

14.1 video_texture.c

The main() function. Creates the actor, parses command-line arguments, connects to the world and do all initialisation.

14.2 camera.c, camera.h

The actual main program that handles the texture updating. A camera structure is defined and it will be used by every function that needs any information about the camera. It includes all the values of the camera properties and some more information as the video image. The function ‘camera_init’ starts up a never-ending loop which will take care of the updating of textures. To update when the camera moves, a callback function is registered for coordinate changes of the camera object. A function that wants to update textures notifies the loop function by signalling the condition variable ‘camera->do_update_cv’. The ‘camera_init’ function also reads all properties from the camera object and sets the instances of the camera structure to the right values.

14.3 properties.c, properties.h

The header file defines all the names of the camera properties. The function ‘prop_init’ initialize the variables in the camera structure that reflects the properties and registers a callback function for property changes. All other functions are to set and update the camera structure variables according to the properties. If a property is changed by the camera DIVE process, it will not be taken care of by these functions, so one must change both the property and the corresponding variable in the camera structure.
14.4 pix_op.c, pix_op.h

These functions do simple image processing that flip the image and reverse internal pixel byte order. There are also functions for reading and writing pixels. This is the implementation of the image resampling.

14.5 texture.c, texture.h

These are the functions to generate the textures. ‘cut_texture’ can transform any shape quadrangle to a square, ‘new_texture’ takes an object in the world and distributes a new texture for it. Also the grabbing of new video images is done here, the architecture independent part.

14.6 winodw.c window.h

These are the functions to open the graphics window and take care of management of it. ‘window_init’ reads the tk initialize script which should open up a window with a graphics area. Adds Xevents to this window to do window redrawing and resizing.

14.7 wow_tcl.c, wow_tcl.h

The c-implementation of the wow-tcl-commands. The function ‘add_tcl_cmds’ adds the tcl-commands to the tcl interpreter. Each new tcl command is explained in 13.0 'User Manual'.

14.8 timestamp.c, timestamp.h

Only one function, that writes the time into a string. This function is used when saving files and the name of the file will include the time.

14.9 transform.c, transform.h

Transformation functions to transform one point in the object or world coordinate system to the image coordinate system. It is in these functions where the camera model is implemented.

14.10 video.c, video.h

These functions are architecture and hardware specific. Only a Silicon Graphics version of the ‘video.c’ file has been implemented. ‘video.h’ is machine independent and can there for be used for every implementation. A video device is opened or started with the function call ‘open_video’ then a new video frame is grabbed with ‘get_video_frame’ and finally the video device is closed with ‘close_video’.

14.11 save_video.c, save_video.h

The function ‘save_videopic’ is almost the same as ‘divepic_save_ppm’, except that it takes care of the extra instances of videopic that is not presented in the divepic struc-
tue. The format of the image data in the file can either be ASCII characters or rawbits, which is decided with a flag.

14.12 graphics_fn.c, graphics_fn.h

Only a OpenGL implementation has been done of these graphics functions. The headerfile is independent of graphics system and the functions defined are for controlling the graphics area.

14.13 draw.c, draw.h

These are the functions to draw different graphic elements such as, the video image, the perspective projection of objects, a cross mark and coordinate axes are implemented in the ‘draw.c’ file. As for the graphics functions these are also dependent on the graphics system and only OpenGL is implemented.
15.0 Literature and references


