Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Computer Vision Laboratory
Computer Vision Group
Prof. Luc Van Gool

Markerless 3D Augmented Reality

Semester Thesis
Oct. 2002 - Feb. 2003

Autor: **Lukas Hohl & Till Quack**

Supervisor: Vittorio Ferrari

Contents

1 Introduction

1.1 Problem Statement

Augmented Reality overlays information onto real world scenes. Future applications of this technology might include virtual tourist guides, factory-workers who get help for their job via head-mounted displays etc.

In this project we want to place artificial 2D and 3D objects into real video sequences. Questions that arise are where and how to place the object. We propose a system which lets the user decide on the first question. Once the object has been placed in the scene, it should be displayed accurately according to the perspective in the original scene, which is especially challenging in the case of 3D virtual objects. This is to be achieved by uncalibrated 3D augmented reality, i.e. no knowledge about the camera positions nor the scene geometry is given or reconstructed.

Further, the objects shall be positioned in the image sequence using information from the Affine Region Tracker developed at the Computer Vision Laboratory at ETH [1]. The tracker works in markerless environments, such that a natural scene can be tracked without adding any artificial markers. The information obtained by one tracked planar region is sufficient to place 2D textures into the scene and also change its coloring to fit the photometric change of the environment. To display 3D structures, two non-coplanar regions need to be tracked (See section 2.2).

The system should be built using a standard graphics API like OpenGL to support portability.

1.2 Task

We extend the system proposed in [1]. In that work 2D virtual textures are superimposed to planar parts of the scene. Our extensions cover photometric changes in virtual textures, augmentation with virtual 3D objects and the incorporation of OpenGL for computer graphics.

To augment a scene with 2D objects, users can choose the location for a virtual texture in the scene. The texture deforms and moves in order to cope with viewpoint changes. Based on affine transformations these deformations and movements are calculated. Photometric changes in the texture according to the conditions in the environment improve the realistic look. The performance of the system is illustrated by Figure 1 which shows a sequence with out of plane rotation and changing brightness.

3D augmentations require data from two separate tracked regions of the original scene. They need to fulfill only two requirements: First they must be non-coplanar, second they should be close to each other. While the first requirement is crucial, the second one influences only the accuracy of the outcome. It should

Figure 1: *A poster is mapped on the tracked region in the window*

be noted, that these restrictions are not very strong: because the tracked regions can be small, it is not difficult to find regions that fulfill the requirements.

The two tracked regions provide two independent tripels of points in complete correspondence across all frames. From their coordinates it is possible to align the real and virtual coordinate system, or, said in another way, to bring 3D coordinates of the virtual object and 2D image points into correspondence.

Two distinct scenarios were implemented. In the simpler one, the position of the object is directly attached to the two tracked regions. A more sophisticated version lets the user choose the position for the virtual object in the scene. In section 2.2 it will be shown that in general the information given by a user lacks accuracy which also leads to less accurate results in augmentation.

We show that the system performs well in aligning the scene with the 3D object under arbitrary large camera movements. Figure 2 shows two images from a scene augmented with a 3D object.

Figure 2: *An artificial coke can placed into the scene, as seen from two different viewpoints in the scene*

2 2D/3D Augmentation Approach

2.1 2D Augmentations

2.1.1 Affine Transformation

The change of the shape of a tracked region between any two images is defined by a 2D Affine Transformation. In fact, 3 points of the region in an image and their corresponding points in the other image, uniquely determine the Affine Transformation. The Affine Transformation includes Rotation, Shearing, anisotropic Scaling and Translation and it preserves parallel lines. See Figure 3.

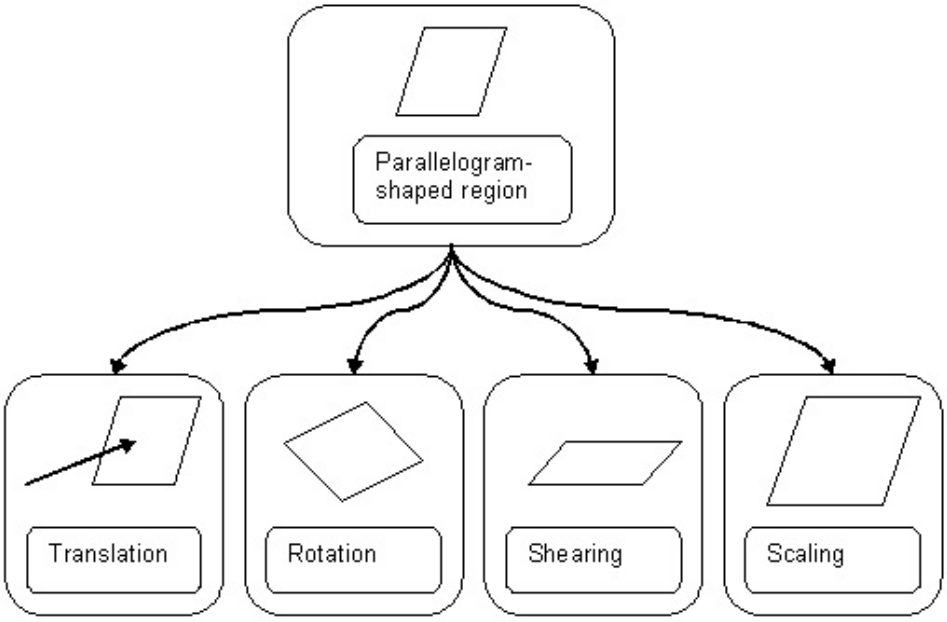

Figure 3: *Affine Transformation: Translation, Rotation, Shearing and anisotropic Scaling*

Taking any 2D point $\mathbf{p}$=(x,y) in its canonical homogeneous coordinate (x,y,1), its transformed point $\mathbf{p}'$=(u,v) is calculated by multiplying the 3x3 Affine Transformation Matrix $\mathcal{A}$ with the 3x1 vector (x,y,1) of the original point $\mathbf{p}$. In general, the 6 unknowns (a_{11},a_{12},...,a_{23}) of the transformation matrix can be fully determined by solving a linear equation system of 6 equations (2 equations per point).

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

2.1.2 Photometric Changes

In order to maximize the realistic impression of the augmented scene, the virtual texture's colors have to be adapted to changing conditions of its environment. Because a region becomes brighter or darker depending on the composition of the light, the position of either the light source, the camera or the object where the region sits on, the texture's color values have to adjust to the observed photometric changes. Therefore the tracked region $\mathcal{R}$ is scanned pixelwise. For each pixel π of the region, the red, green and blue values (R,G,B) are taken and summed up separately. To get the average RGB values (R_{avg},G_{avg},B_{avg}), the total sums of each color R_{tot},G_{tot},B_{tot}, are divided by the total number of pixels Π of the region.

$$R_{tot} = \sum_{\pi \in \mathcal{R}} R(\pi) \qquad G_{tot} = \sum_{\pi \in \mathcal{R}} G(\pi) \qquad B_{tot} = \sum_{\pi \in \mathcal{R}} B(\pi)$$

$$R_{avg} = \frac{R_{tot}}{\Pi} \qquad G_{avg} = \frac{G_{tot}}{\Pi} \qquad B_{avg} = \frac{B_{tot}}{\Pi}$$

To finally get the photometric change of a region between any two images, each average RGB value ($R_{avg,b}$,$G_{avg,b}$,$B_{avg,b}$) of the second image (index b) divided by its corresponding average RGB value ($R_{avg,a}$,$G_{avg,a}$,$B_{avg,a}$) of the first image (index a) defines the scale factor ($\mathcal{F}_{\mathbf{R}}$,$\mathcal{F}_{\mathbf{G}}$,$\mathcal{F}_{\mathbf{B}}$) for each colorband.

$$\mathcal{F}_{\mathbf{R}} = \frac{R_{avg,b}}{R_{avg,a}} \qquad \mathcal{F}_{\mathbf{G}} = \frac{G_{avg,b}}{G_{avg,a}} \qquad \mathcal{F}_{\mathbf{B}} = \frac{B_{avg,b}}{B_{avg,a}}$$

Multiplying RGB values of each pixel of the texture with scale factors ($\mathcal{F}_{\mathbf{R}}$,$\mathcal{F}_{\mathbf{G}}$,$\mathcal{F}_{\mathbf{B}}$), adjusts the color of the texture to suit the photometric changes of the tracked region. This approach allows the virtual texture to appear realistic in the scene.

2.2 3D Augmentations

In this chapter we describe the theoretical concepts behind our system for 3D augmented reality.

For the further steps we differentiate between a "simple" approach and a more sophisticated one: In the first case the object is directly mapped to the location of the tracked region, in the latter one the object's location is determined by the user.First we will recall the information given and then present how to use it to solve the problem.

For the simple *and* the sophisticated approach, the following is given:

1. Two non-coplanar tracked regions from the Tracker (see section 3.1.1). Four non-coplanar points $\mathbf{p'_c}$, $\mathbf{p'_1}$, $\mathbf{p'_2}$, $\mathbf{p'_3}$ are selected from these regions, such that they define the projection of a real world coordinate system. See Figure 4.

2. A 3D virtual object to be placed in the scene. Its bounding box is a parallelepiped that touches the outermost points of the object. It is obtained from the objects vertices as described in section 4. We select four points ($\mathbf{P_c}$, $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$) in 3D that define the virtual coordinate base of the object in 3D. See Figure 4.

Note the notation for points and that we use homogeneous coordinates such that a 2D image point is defined by $\mathbf{p}$=(x,y,1), a point in 3D by $\mathbf{P}$=(X,Y,Z,1).

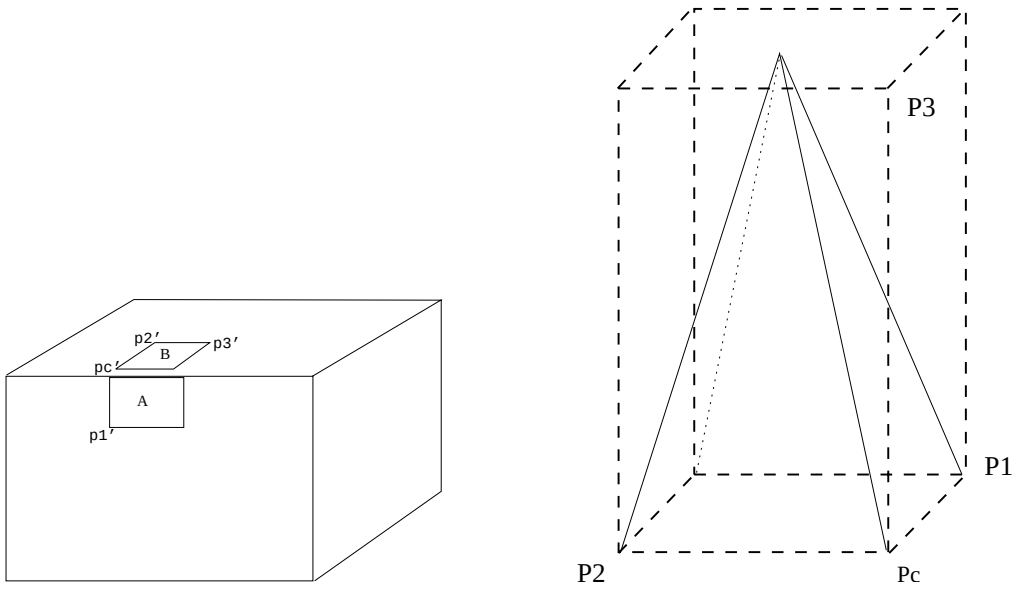

Figure 4: *Two non coplanar regions A,B and the bounding box for an object, a pyramid in this case*

In general a 3D world-point is projected to a 2D image point by a 3x4 projection matrix $\mathcal{P}$.

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \mathcal{P} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

where $\mathcal{P}$ is the 3x4 projection Matrix

$$\mathcal{P} = \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note that the last line is (0 0 0 1) because we use orthogonal projection.

2.2.1 The simple approach to object positioning

To insert the 3D virtual object into the scene, we need to find a projection matrix that maps the bounding box to the correct location in each image. (Each point ($\mathbf{P_c}$, $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$) is projected by $\mathcal{P}$.)

This gives 8 equations for 8 unknowns (p_{11},...,p_{24}). For example the equations obtained from point $\mathbf{P_1}$ are

$$\begin{aligned} x_1 &= p_{11} \cdot X_1 + p_{12} \cdot Y_1 + p_{13} \cdot Z_1 + p_{14} \\ y_1 &= p_{21} \cdot X_1 + p_{22} \cdot Y_1 + p_{23} \cdot Z_1 + p_{24} \end{aligned} \quad (1)$$

In the simple approach the object is directly mapped to the location of the tracked regions, i.e. the points $\mathbf{p'_c}$, $\mathbf{p'_1}$, $\mathbf{p'_2}$, $\mathbf{p'_3}$ are the image-points in equation (1). Thus, for every new frame the projection matrix $\mathcal{P}$ can be calculated with a linear solver and can be used in OpenGL as described in section 4. The positioning of the object during the sequence stays accurate this way, however the user is not given a choice where to place the object.

2.2.2 The sophisticated approach to object positioning

The sophisticated approach lets the user choose the location for the 3D virtual object in the scene. The user-interaction provides us with:

1. Four 2D image points ($\mathbf{p_{ca}}$, $\mathbf{p_{1a}}$, $\mathbf{p_{2a}}$, $\mathbf{p_{3a}}$) in the *first* image of the sequence (selected by the user). They define the projection of the coordinate base of the 3D virtual object in the first image. See Figure 5.

2. Four 2D image points ($\mathbf{p_{cb}}$, $\mathbf{p_{1b}}$, $\mathbf{p_{2b}}$, $\mathbf{p_{3b}}$) in the image plane of *another*, i.e. the last, image of the sequence.They define the projection of the coordinate base of the 3D virtual object in that image.

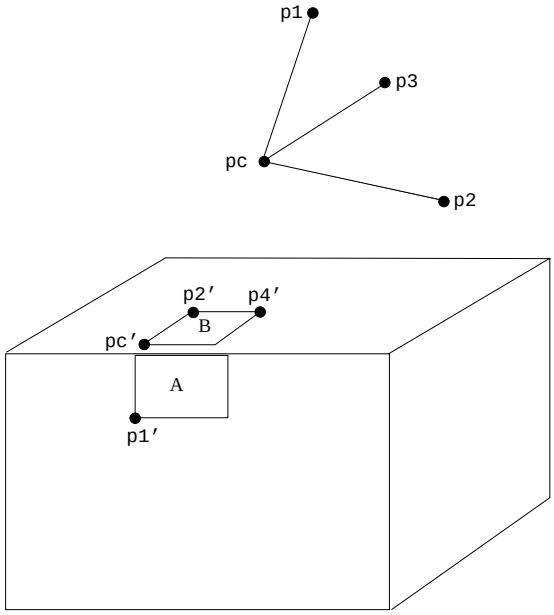

Figure 5: *The tracked regions A, B and the user defined base* $\mathbf{p_c}$, $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$

Before calculating and applying the projection matrix for each image, the correspondence between the coordinate base defined by the user and the tracked regions needs to be established. Put another way, the projected image points of the four bounding box points ($\mathbf{P_c}$, $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$) need to be calculated for each image first.

Remember that $\mathbf{p'_c}$, $\mathbf{p'_1}$, $\mathbf{p'_2}$, $\mathbf{p'_3}$ are four points of the tracked regions each with coordinates (x',y',1). (See figure 5)

For **any** 3D point (X_s,Y_s,Z_s)

$$\begin{aligned} x &= x'_c + X_s \cdot (x'_1 - x'_c) + Y_s \cdot (x'_2 - x'_c) + Z_s \cdot (x'_3 - x'_c) \\ y &= y'_c + X_s \cdot (y'_1 - y'_c) + Y_s \cdot (y'_2 - y'_c) + Z_s \cdot (y'_3 - y'_c) \end{aligned} \tag{2}$$

is the 2D image point.

X_s, Y_s, Z_s are the 3D coordinates of a bounding box point expressed in the 3D space defined by the tracked regions.

This means, to determine the correct image points for every 3D point of the virtual object in each image, X_s, Y_s, Z_s need to be calculated.

Thus, to each point of the bounding box ($\mathbf{P_c}$, $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$) equation 2 with $\mathbf{p_{ca}}$, $\mathbf{p_{1a}}$, $\mathbf{p_{2a}}$, $\mathbf{p_{3a}}$ as "image points" is applied.

This results in 2 equations for 3 unknowns per point (X_s, Y_s, Z_s, $s \in \{c, 1, 2, 3\}$), an underdetermined system. The base from another image ($\mathbf{p_{cb}}$, $\mathbf{p_{1b}}$, $\mathbf{p_{2b}}$, $\mathbf{p_{3b}}$) is needed. (Obviously the user must mark the same points of the scene as in the first frame). This gives 4 equations for 3 unknowns, written as equation system

e.g. for point $\mathbf{P_1}$ of the bounding box

$$\begin{pmatrix} (x'_{1a} - x'_{ca}) & (x'_{2a} - x'_{ca}) & (x'_{3a} - x'_{ca}) \\ (y'_{1a} - y'_{ca}) & (y'_{2a} - y'_{ca}) & (y'_{3a} - y'_{ca}) \\ (x'_{1b} - x'_{cb}) & (x'_{2b} - x'_{cb}) & (x'_{3b} - x'_{cb}) \\ (y'_{1b} - y'_{cb}) & (y'_{2b} - y'_{cb}) & (y'_{3b} - y'_{cb}) \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} = \begin{pmatrix} (x_{1a} - x'_{ca}) \\ (y_{1a} - y'_{ca}) \\ (x_{1b} - x'_{cb}) \\ (y_{1b} - y'_{cb}) \end{pmatrix}$$

The subscripts a and b in the coordinate values refer to the two images.

After doing the same for the three remaining points of the bounding box, we have a set of (X_s, Y_s, $Z_s, s \in \{c, 1, 2, 3\}$) such that equation (2.) can be applied in every image. This gives four image points that define the user-selected base and correspond to the four points from the bounding box. Thus, the projection matrix for each image can be calculated.

This approach gives the user high flexibility because one can chose where to place the object freely. However, the user will generally not be able to mark the exact same points in the two images with different viewpoints on the scene, which results in less accurate augmented scenes. Also, if the tracked regions are very far from the user-selected points, the accuracy of positioning degrades.

3 Software Architecture

3.1 Project Structure

The project is built on three modules:

1. Tracking Tool
2. Texture Mapping Module
3. 3D Object Augmentation Module

All modules are independent, they have their own executables and are separately compiled. The Texture Mapping Module and the 3D Object Augmentation Module ask for a History File. The History File is a plain ASCII file containing the coordinates of the tracked regions. Texture Mapper Module also asks for a texture (to be mapped), which can be of any type (e.g. JPG, BMP). The 3D Augmentation Module asks for a 3D model. See Figure 6.

The History File contains a list of paths where the images of a sequence are saved. Each image path is labeled with a frame number. For every tracked region (it is possible to track more than one region) there is another list (named moving region) with the coordinates of the points and their corresponding frame number. Moreover each region in each frame has a value, which can be either 0 or 1. If it is 0, then the Tracking Tool could fully track the region otherwise it is only a prediction of where the region could be, a so-called ghost region. See Figure 7.

3.1.1 The Tracking Tool

The tracker works on the regions proposed by Tuytelaars and Van Gool [2, 3]. This is a method for the automatic extraction and wide-baseline matching of small, planar regions. These regions are extracted around anchor points and are *affinely invariant*: given the same anchor point in two images of a scene, regions covering the same physical surface will be extracted, in spite of the changing viewpoint. We concentrate on a single region type: parallelogram-shaped (anchored on corner points). These are based on two straight edges intersecting in the proximity of the corner. This fixes a corner of the parallelogram (call it $\mathbf{c}$) and the orientation of its sides. The opposite corner (call it $\mathbf{q}$) is fixed by computing an affinely invariant measure on the region's texture.

Parallelogram-shaped regions are characterised (i.e. completely defined) by any three corners. Thus, a region is completely defined by three points.

A description of the tracking algorithm is given in [1]. Parts of the algorithms have been improved since these publications, but it is outside the scope of this report to document them. The basic scheme of the tracker staid the same and we briefly report it, to help introducing some concepts needed in the rest of the document.

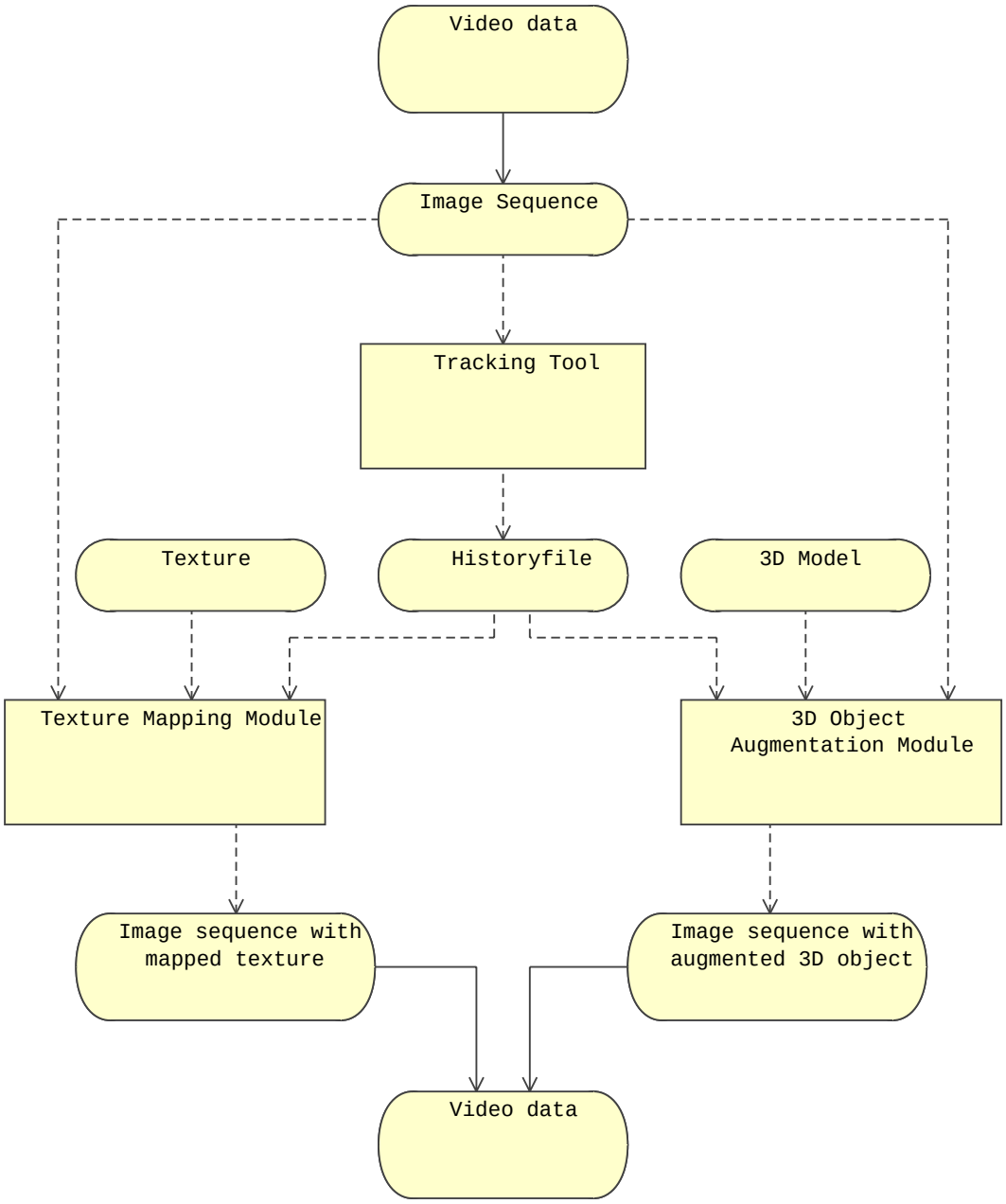

Figure 6: *Process Flow Diagram*

```
Frame #   Filename
  0       /home/lhohl/SEMA/sequences/office/003.jpg
  1       /home/lhohl/SEMA/sequences/office/004.jpg
  2       /home/lhohl/SEMA/sequences/office/005.jpg
  3       /home/lhohl/SEMA/sequences/office/006.jpg


----------------  MovingRegion History  ---------------

Frame # Ghost   Coordinates
  0     0       120, 239          200, 245          124, 266          204, 272
  1     0       117.583, 236.119  200.3, 241.03     120.339, 266.309  203.056, 271.22
  2     0       115.312, 236.857  198.184, 241.467  118.216, 266.716  201.088, 271.326
  3     0       113.364, 236.934  196.068, 241.606  116.186, 267.109  198.891, 271.78


----------------  MovingRegion History  ---------------

Frame # Ghost   Coordinates
  0     0       261, 202          242, 238          153, 192          134, 228
  1     0       262.561, 198.755  241.824, 236.425  151.791, 191.607  131.054, 229.276
  2     1       259.672, 199.356  240.045, 236.488  149.935, 191.756  130.308, 228.888
  3     0       258.017, 199.313  238.294, 236.394  147.243, 192.235  127.52, 229.316
```

Figure 7: *An example of a History File (with 2 tracked regions)*

The general goal of the tracker is to put a region into complete correspondence in all frames of the sequence. This can be seen as the process of finding the three characteristic points in all frames, or, equivalently, as finding the affine transformation between the first frame and every other frame.

We consider tracking a region R from a frame F_{i-1} to its successor frame F_i in the image sequence. First we compute a prediction $\hat{R}_i = A_{i-1}R_{i-1}$ of R_i using the affine transformation A_{i-1} between the two previous frames ($A_1 = I$). An estimate $\hat{\mathbf{a}}_\mathbf{i} = A_{i-1}\mathbf{a}_{\mathbf{i-1}}$ of the region's anchor point[1], is computed, around which a circular search space S_i is defined. The radius (called *follow radius*) of S_i is proportional to the current translational velocity of the region.

The anchor points in S_i are extracted. These provide potentially better estimates for the region's location. We *investigate* the point closest to $\hat{\mathbf{a}}_\mathbf{i}$ looking for the target region R_i. The anchor point investigation algorithm differs for geometry-based and intensity-based regions and can be found in [1]. During the investigation algorithm, the texture of candidate regions will have to be compared to the texture of the region to be tracked for validation. The comparison reference has been chosen to be R in the *first frame* (R_1) of the sequence. This helps to avoid the cumulation of tracking errors along the frames. Since the anchor points are sparse in the image, the one closest to the predicted location is, in most cases, the correct one. If not, the anchor points are iteratively investigated, from the closest (to $\hat{\mathbf{a}}_\mathbf{i}$) to the farthest, until R_i is found (figure 9).

In some cases it is possible that no correct R_i is found around any anchor point in S_i. This can be due to several reasons, including occlusion of the region, sudden acceleration (the anchor point of R_i is outside S_i) and failure of the anchor point extractor. When this happens the region's location is set to the prediction

[1]Harris corners

Figure 8: *Tracking a parallelogram-shaped region*

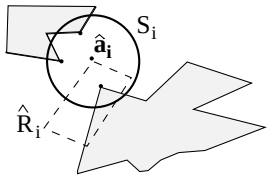

Figure 9: Anchor points (thick dots) are extracted in the search space S_i, defined around the predicted anchor point $\hat{\mathbf{a}}$.

($\mathbf{a_i} = \hat{\mathbf{a}}_\mathbf{i}$), and the tracking process proceeds to the next frame, with a larger S. In this case, the region is said to be a *ghost* in frame F_i. If a region is a ghost for more than 6 frames, it labeled as *lost ghost* and it is abandoned.

To summarize, in order to track region R in frame F_i, the tracker needs the region in the previous frame R_{i-1} (together with A_{i-1} and the follow radius in the previous frame, and all previous frame related information) and the region in the first frame R_1, for texture comparison purposes.

3.1.2 The Texture Mapping Module

The Texture Mapping Module consists of the following class structure (see Figure 10):

`Point`:
The `Point` class is the lowest-level member in the class hierarchy. A point can be either 2D or 3D.

`Region`:
The `Region` class is built on 4 `Point` and an image path. It includes a function called Scanframe(), which scans the region and calculates the average RGB values (see section 2.1.2).

`MovingRegion`:
The `MovingRegion` class is a list of `Region`. It links consecutive regions of a image series to a list.

`AMR_Builder`:
The functionality of the `AMR_Builder` class is to build a new `MovingRegion` based on an already existing `MovingRegion` (moving region of the textfile) and a new start `Region` (defined by the user). It calculates all the affine transformations between consecutive `Region` of the given `MovingRegion` and applies them on the new start `Region`.

`HistoryParser`:

The `HistoryParser` opens and parses the textfile. It parses the list of the paths of the images and its corresponding frame numbers and keeps them in memory for later reference. Next it determines the beginning of a new moving region (see definition above) and and launches the `MR_Parser`.

`MR_Parser`:
The `MR_Parser` parses the moving region and links every image path with its corresponding points of the tracked region. Taking the points (of type `Point`) of a region and its image path, it builds instances of an object called `Region` (see below) which become to a `MovingRegion` (see below).

`TextureImage`:
The `TextureImage` is a container for the texture image.

`ImageLoader`:
The `ImageLoader` class loads the `TextureImage`.

`TextureMapper`:
The `TextureMapper` class gets the `MovingRegion` of the textfile and the user-defined region. It then launches the `AMR_Builder` and the `ImageLoader` and maps the texture using OpenGl functions.

3.1.3 The 3D Object Augmentation Module

The 3D Object Augmentation Module consists the following class structure (see Figure 11):

The following classes already mentioned earlier are also part of the 3D Object Augmentation Module and keep their functionality as in the Texture Mapping Module:

`HistoryParser`
`MR_Parser`
`Point`
`Region`
`MovingRegion`
`ImageLoader`
`TextureImage`

The new classes are:

`ModelLoader`:
The `ModelLoader` class loads 3D custom models.

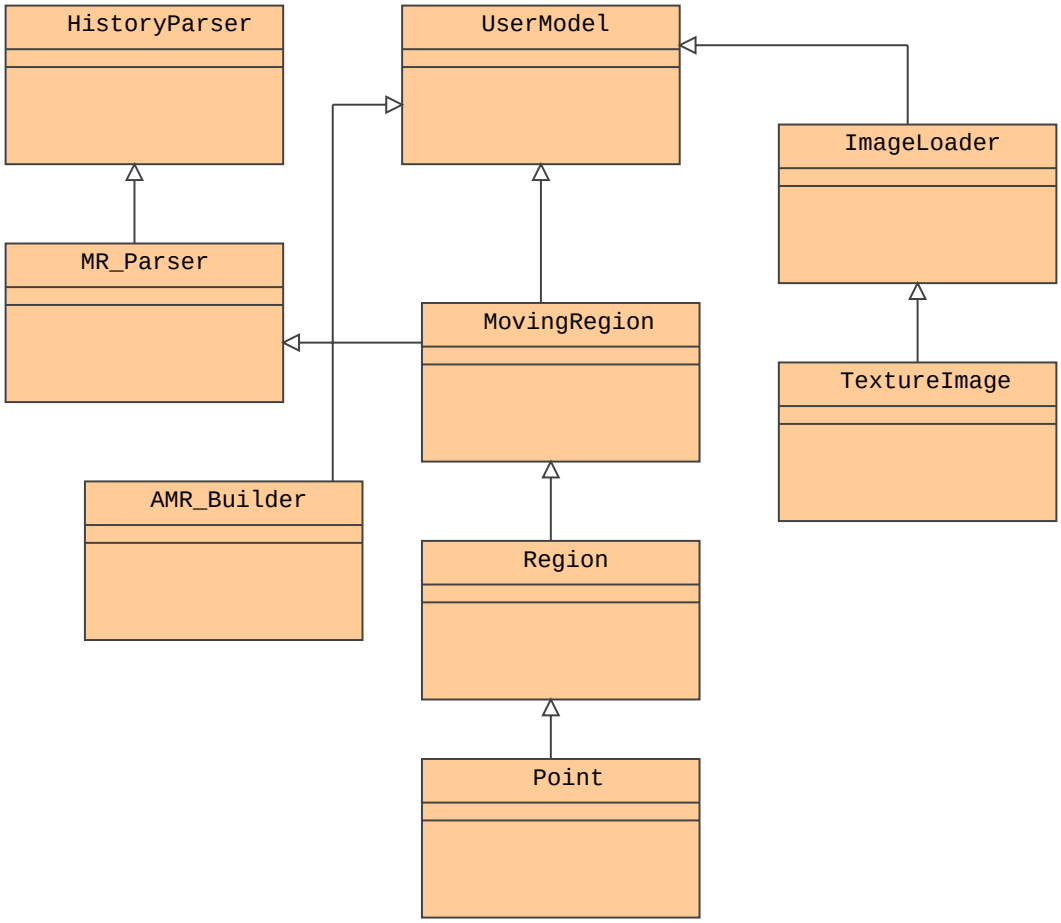

Figure 10: *A simplyfied Class Diagram of the Texture Mapper Module*

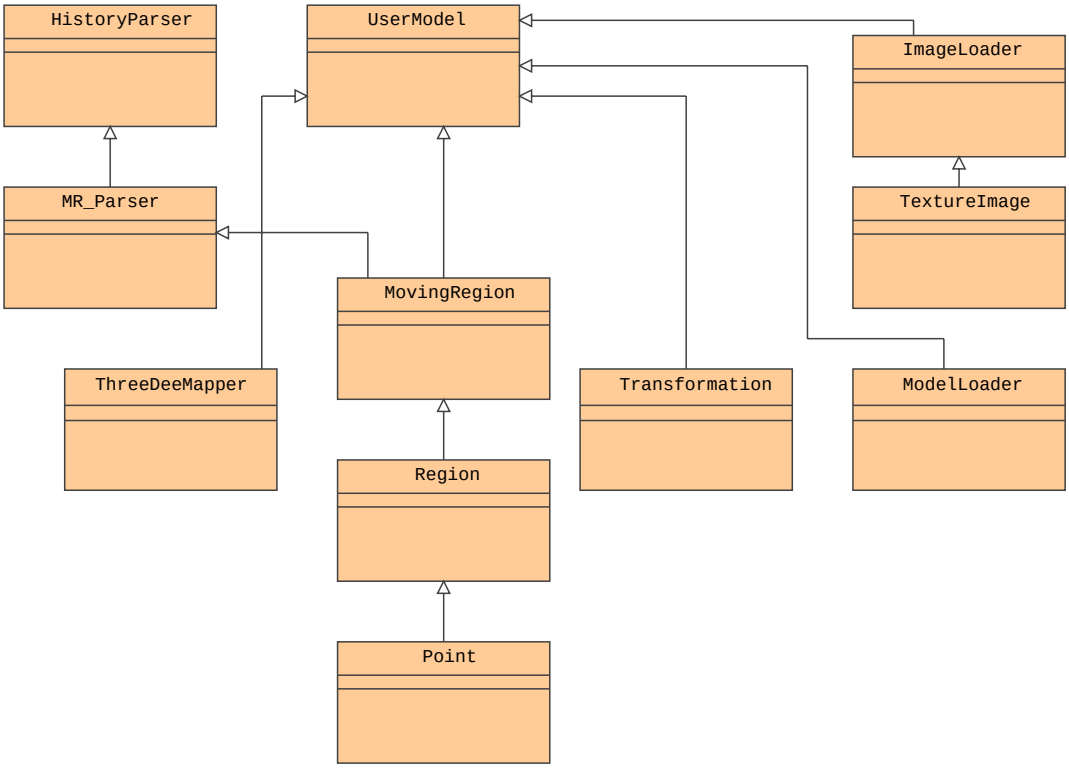

Figure 11: *A simplyfied Class Diagram of the 3D Object Augmentation Module*

`ThreeDeeMapper`:
The `ThreeDeeMapper` class is handling the calculation of the transformation between the 4 points given from 2 non-coplanar tracked regions and the user-defined base for each image.

`Transformation`:
The `Transformation` class calculates the projection matrix mapping 2D points to 3D points.

`UserModel`:
The `UserModel` class is the equivalent to the `TextureMapper` class of the Texture Mapper Module. It gets two `MovingRegion` and augments the scene with the 3D object.

4 Software Implementation

The Software is implemented ANSI C++ and uses some standard libraries and APIs. The application was designed for portability and standards compliance. The following widely deployed libraries were used:

1. OpenGL: To display Graphics.
2. ImageMagick: To load Movies, textures and images in various file-formats.
3. VNL: To solve systems of linear equations. VNL is part of a larger software library (VXL) for Computer Vision.

As described before, the data from the Region Tracker is exported to history files as shown in Figure 7. The parser for the History Files was written in plain C++ without any external libraries to guarantee portability. Each History File can contain several moving regions but only one list of images. Thus the list of images is parsed first as a reference, then each moving region list is parsed and converted to a double-linked list of elements of the class `Region`.

Each instance of the class `Region` contains the location of the tracked region in the current frame, a ghost flag, the path to the file of the current movie frame etc. All this information is used to place the objects in the scene with OpenGL.

4.1 OpenGL

Since its introduction in 1992, OpenGL [4] has become the most widely used and supported 2D and 3D graphics application programming interface (API). OpenGL is available for all common computing platforms, thus ensuring wide application deployment. Also, several extensions to OpenGL are available. We use the GLUT extension, the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing API for OpenGL. Like OpenGL itself GLUT provides a portable API. This means that a single OpenGL program can be written that works on both Win32 PCs and X11 workstations.

A further advantage is the licensing model which allows free use for research purposes. OpenGL for C++ is not object oriented, which makes it difficult to use it properly in C++ programs. OpenGL is designed with the concept to behave like a state machine. State transitions are defined with functions. For example the code for drawing a square is the following:

```
glBegin(GL_QUADS);
glVertex3f(0,0,0);
glVertex3f(1,0,0);
glVertex3f(1,1,0);
glVertex3f(0,1,0);
glEnd();
```

`glBegin()` with the parameter `QL_QUADS` sets OpenGL and square drawing mode, the following vertices are connected to a square. The `glEnd()` call exits the square-drawing mode. In addition, functions for display and user-interaction (like mouse- or keyboard-handler) have to be defined as C-style Callback functions. For instance

```
glutDisplayFunc(myDisplayFunc)
```

needs to be called to register `myDisplayFunc()` as the display function.

This non object-oriented style caused some difficulties during integration into our object-oriented environment. All the functions that are related to OpenGL are now defined within the classes TextureMapper or UserModel for 2D or 3D respectively. All the class members and member-functions had to be defined as static - otherwise the use of the callback-functions as imposed by the OpenGL architecture would not work. This implies that only one instance of the classes UserModel or TextureMapper at a time can be active (Singleton Classes), which is not to strong a limitation in our case, as there is no need for more.

A further challenge that resulted from the architecture of OpenGL and the GLUT was the use of the so called `glutMainLoop()`. It needs to be called to start processing the OpenGL functions. Once started, the `glutMainLoop()` can't be quit. This imposes rather strong restrictions concerning the data exchange with other classes. All runtime calculations have to be done from functions that run within the `glutMainLoop`, which are basically the display function and the mouse- and keyboard- handlers in our case. Within OpenGL several matrices define how the current scene is presented on the screen. The Modelview Matrix positions the object in the world, the Projection Matrix determines the field of view (or viewing volume in OpenGL terms). Finally the viewport defines how the scene is mapped to the screen (position, zoom, etc.).

In spite of the availability of functions that directly rotate, translate etc. one can also set and display the desired scene by accessing the matrices directly.

This is done using the `glLoadMatrix()` function. A typical series of commands would be the following

```
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
glLoadMatrix(new_projection_matrix)
```

with `new_projection_matrix` being a sixteen-element array containing the new matrix values. The matrices become important right at the beginning of the user-interaction process. The points that defined the base (**pc**,**p1**, **p2**, **p3**) need to be transformed from pixel- (window-) coordinates to the corresponding OpenGL object-coordinates. (OpenGL Coordinates are measured in fractions of 1 in x and y direction starting at the center of the window. For the z values a special depth range is defined).

The utility-function `gluUnProject()` returns the correct OpenGL coordinates regarding to the current matrices and a particular depth in z direction.

For the rest of the process the matrices are used the other way around, obviously: The matrices are set based on the calculations as described in section 2.2. The concept of matrix-usage in OpenGL as described above is a little different than the theoretical one. However, we managed to transform our calculations in a way that they fit the OpenGL matrices.
The setting of the matrices is called from the display function. Here, OpenGL offers double-buffered animation. While one framebuffer is displayed, the contents of the other one are calculated in the background. So the process in the display function basically consists of getting the information for the current tracked region, obtaining the new projection matrix, loading it to OpenGL, advancing on step in the list of moving regions and then swapping the framebuffer.

A further OpenGL feature we used is texture mapping. The following steps are performed to map textures to an object:

- Specify and load the texture.
- Enable texture mapping.
- Draw the scene, supplying both texture and geometric coordinates.

A texture contains image data. (In OpenGL textures are restricted to have width and height values that are powers of two). The texture is loaded with an instance of our ImageLoader class.
A texture is initialized using the following series of commands

```
// create texture
glGenTextures(1, (GLuint*) &texture_id);
glBindTexture(GL_TEXTURE_2D, texture_id);   // 2d texture (x and y size)
// scale linearly when image larger or smaller than texture
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, 3, timage->sizeX, timage->sizeY, 0, GL_RGB,
GL_UNSIGNED_BYTE, timage->data);
```

The last command assigns a texture that is scaled linearly when the image is smaller than the texture, level of detail 0 (normal), 3 components (red, green, blue).

To draw a textured square also the texture coordinates have to be given

```
glBegin (GL_QUADS);
glTexCoord2f (0.0f,0.0f);
glVertex3f (0, 0, 0);
glTexCoord2f (1.0f, 0.0f);
```

```
glVertex3f (1, 0, 0);
glTexCoord2f (1.0f, 1.0f);
glVertex3f (0, 1, 0);
glTexCoord2f (0.0f, 1.0f);
glVertex3f (1, 1, 0);
glEnd ();
```

A remark should also be made on how to import custom 3D Models into our application. While it is easy to rebuild simple objects consisting only of few planes (like cubes) every time the framebuffer is swapped, it is not useful to do this for complicated objects with several thousand vertices. OpenGL offers the so called display lists which allows to precompile these objects and then call them by a unique list id. OpenGL does not have a particular file format for 3d objects like 3DS for Autodesk's 3d Studio for instance, the only available format are the display lists mentioned before.

Most models that can be found in the Word Wide Web for esample, are usually offered in 3DS or some other format, but not in OpenGL code. Thus one needs a converter. *DeepExploration* from *Right Hemisphere* turned out to be very useful. Several File Formats can be exported directly to OpenGL display lists. After exporting, the code must be edited to make it object oriented and the model can then be loaded by calling this class. We augmented our scene with models consisting of over 25000 vertices. We have added some sample models to the application that can be chosen by mouse-click.

It should also be mentioned that we obtain the bounding box as it is described in section 2.2 on page 8 by determining the minimal and maximal x y and z values of all vertices, while the display list is built.

While calculation and the display of 3D objects is rather fast, the loading of the background images (i.e. the images from the original scene) slows performance. There are basically two ways possible to add a background to a OpenGL environment. Either the background image is written directly to the framebuffer with the function `glPixelWrite()` or the image is mapped as a texture to a rectangle behind the scene. While it is said that the latter one is faster, it has a limitation, too: Textures can only have width and height that are powers of two. One could add some black space to each image so that it gets the right size, and after displaying it clip the scene so that it fits the screen. We decided for the first way for reasons of simplicity.

4.2 ImageMagick

We need images from several file formats in our program. The movie frames are to be loaded, the texture images, too.

OpenGL does not offer any built-in functionality to load image files. Instead of writing loaders for each file type manually we wrote a class that accesses

Magick++, the C++ interface to ImageMagick [5]. ImageMagick allows loading, writing and converting of nearly any kind of image file-format (over 80 file formats are supported). Note, that the interface to ImageMagick not only allows to load image data for textures and frames, we also implemented a function that exports the augmented sequence to any file format supported by ImageMagick.

ImageMagick is also widely deployed and also available for many computing platforms. This, again, ensures portability.

4.3 VNL

While the process of setting the matrices in OpenGL was described beforehand, nothing has been said about how to solve the sstems of linear equations of 2.2.

The calculation of the projection matrices and the solution of the equations in section 2.2 both need solving of linear equation systems. For this purpose the VNL library from VXL [6] was used. VXL (the Vision-something-Libraries) is a collection of C++ libraries designed for computer vision research; vnl is a library with numerical containers and algorithms, in particular vnl provides matrix and vector classes with operations for manipulating them.

5 Results

We present 3 image sequences in total that demonstrate the qualities of our system.

Two examples illustrate the results for 2D Augmented Reality, i.e. texturemapping and photometric changes.

Figure 12 illustrates a virtual number mapped to a tram. The sequences show the trackers strength to follow the tram even around a curve which means out of plane rotation for the tracked region. As a result of an affine transformation the number can be mapped to any other place and is transformed accordingly.

Figure 12: *A virtual number mapped on the tram*

Figure 13 shows the effects of photometric changes. The light moves over the poster, the patch mapped to the poster changes it's brightness accordingly while also changing shape and position correctly.

A further example illustrates our results for 3D Augmented Reality in Figure 14. The object is always positioned correctly, while the camera moves backwards and rotates around the object at the same time.

The experiments confirmed that non-calibrated Augmented Reality in 2D and 3D can be achieved relying on the concepts presented in section 2 and on OpenGL for implementation. The experiments thus showed that the system can accurately augment natural scenes with 2D and 3D virtual objects at user-selected positions under general motion conditions and without any artificial markers.

Figure 13: *The effects of photometric changes*

Figure 14: *A scene augmented with the 3D Model of a Buddha statue*

6 Conclusions

With this thesis we successfully continued the work of Ferrari et al. [1]. Based on the data exported from the Tracking Tool we were able to port the system for 2D augmentation to a widely used standard graphics API. We enhanced the visual appeal by introducing features like the photometric change in the superimposed textures according to their environment. We brought the system literally to a new dimension by introducing 3D virtual objects as augmentations. The system showed good results and the code is portable to various platforms due to the standard API's and libraries used.

References

[1] V. Ferrari, T. Tuytelaars, and L. Van Gool. Markerless augmented real-time affine region tracker. Proceedings of the IEEE and ACM International Symposium on Augmented Reality:87 – 96, 2001.

[2] T. Tuytelaars and L. Van Gool. Contend-based image retrieval based on local, affinely invariant regions. Third Int. Conf. on Visual Information Systems:493–500, 1999.

[3] T. Tuytelaars and L. Van Gool. Wide baseline stereo based on local, affinely invariant regions. British Machine Vision Conference:412–422, 2000.

[4] http://www.opengl.org

[5] http://www.imagemagick.org

[6] http://vxl.sourceforge.net