Minimally Invasive Projector Calibration for 3D Applications

Marcel Lancelle, Lars Offen, Torsten Ullrich, Torsten Techmann, Dieter W. Fellner

Institute of Computer Graphics and Knowledge Visualization, TU Graz
Inffeldgasse 16c
8010 Graz, Austria
Tel.: +43 316 873 5401
Fax: +43 316 873 5402
E-Mail: m.lancelle@cgv.tugraz.at

Abstract: Addressing the typically time consuming adjustment of projector equipment in VR installations we propose an easy to implement projector calibration method that effectively corrects images projected onto planar surfaces and which does not require any additional hardware. For hardware accelerated 3D applications only the projection matrix has to be modified slightly thus there is no performance impact and existing applications can be adopted easily.

Keywords: projector calibration, virtual reality

1 Introduction

In every virtual environment like CAVE^{TM} installations or power walls the projected images need to be aligned accurately at the edges from one screen to the next. An accurate manual alignment takes a long time and has to be redone whenever the setup changes or maintenance tasks have to be performed, e.g. a mirror is moved by accident or a light bulb is exchanged etc.

In contrast, the presented algorithm only needs roughly aligned projectors with a little overlap at each edge of the target region to determine the calibration's fine tuning. Using this approach the whole calibration process takes about one minute per projector.

The presented software calibration calculates a calibration matrix based on a few user inputs. The resulting matrix can be integrated seamlessly into a hardware accelerated render pipeline.

The calibration is not perfect as it can only handle affine distortions, but due to its optimal integration into the render pipeline it is a good compromise between high performance and desired accuracy.

2 Related Work

The goal of virtual reality (VR) is the synthesis of a convincing illusion within a virtual environment. The most important aspect in a VR system is the quality of vision because it has the greatest and most immediate impact on the human sensory system [BSdV01]. Besides the displayed content the display quality and especially its correct calibration is of big importance.

Current calibration methods can be subdivided into three groups. The first and most frequently used method is the calibration by hand which is a non-practicable method for large power walls or caves.

Contrary to manual calibration the second group consists of methods with an automatic approach using embedded cameras [LS04], [RB01] or other sensor systems [LDMA+04]. These systems have the great advantage to work out of the box. Unfortunately, they need specialized or modified hardware and projectors.

To overcome this disadvantage the third group uses standard video cameras that are neither embedded in the projector nor rigidly attached to it. These systems are not only suitable for mobile activities and applications [SSM01], but also for fixed installations [BMY05]. Especially in virtual 3D environments in which cameras are already installed – e.g. for tracking purposes – this approach is widely used [SHVG02], [GWN+03].

Our method is a semi-automated calibration procedure without additional hardware. The main difference to most established systems is its seamless integration into the render pipeline, which is similar to an approach by Ramesh Raskar [Ras00] who also uses a projection matrix modification to implement a single pass projector calibration.

3 Calibration matrix

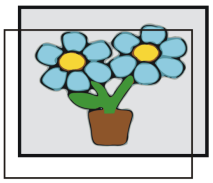
In a hardware accelerated 3D application a point p is transformed to $q = (q_x, q_y, q_z, q_w) = P \cdot M \cdot p$ with the modelview matrix[1] M and the projection matrix P. The perspective division leads to $q_p = (q_x/q_w, q_y/q_w, q_z/q_w, 1)$ where the first two elements are the normalized screen coordinates $q_s = (q_x/q_w, q_y/q_w)$.
We now use a 4x4 calibration matrix C that is applied after all other coordinate transformation matrices but before the perspective division. The corrected point $r = C \cdot P \cdot M \cdot p$ leads to the normalized screen coordinates $r_s = (r_x/r_w, r_y/r_w)$.

[1] We use the OpenGL notation.

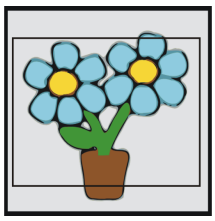
For at least four reference points q_{s_i} the user specifies corrected points r_{s_i}. The calibration matrix

$$C = \begin{pmatrix} a & b & 0 & d \\ e & f & 0 & h \\ i & j & 1 & 0 \\ m & n & 0 & 1 \end{pmatrix}$$

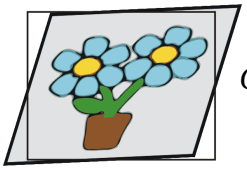
is calculated such that all q_{s_i} will be projected as close as possible to the target positions r_{s_i} (see next section). This matrix is a combination of the matrices shown in Figure 1 ($C = C_T \cdot C_{Sh} \cdot C_S \cdot C_{Tr} \cdot C_R$).



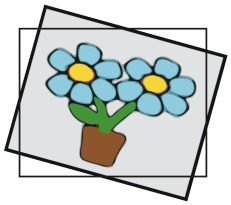
$$C_{Tr} = \begin{pmatrix} 1 & 0 & 0 & d_{Tr} \\ 0 & 1 & 0 & h_{Tr} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



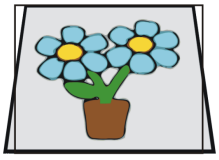
$$C_S = \begin{pmatrix} a_S & 0 & 0 & 0 \\ 0 & f_S & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$C_{Sh} = \begin{pmatrix} 1 & b_{Sh} & 0 & 0 \\ e_{Sh} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$C_R = \begin{pmatrix} a_R & b_R & 0 & 0 \\ e_R & f_R & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$C_T = \begin{pmatrix} a_T & 0 & 0 & d_T \\ 0 & f_T & 0 & h_T \\ i_T & j_T & 1 & 0 \\ m_T & n_T & 0 & 1 \end{pmatrix}$$

Figure 1: Matrices to handle translation, scaling, shearing, rotation and trapezoid deformation. These effects occur when projecting on a planar surface and can be corrected with our calibration method.

a, b, e and f need to be optimized to correct for scaling, shearing and rotation, d and h belong to the translation. Using m and n it is possible to correct for scaling depending on the screen coordinates which represents a trapezoid distortion (see Figure 1).

The latter case is more complex than the previous ones. The distortion is non-linear and additional scaling and translational effects occur when $m \neq 0$ or $n \neq 0$ [FH05]. Also the z coordinates are affected during perspective division so that problems with the clipping planes may occur (in OpenGL pixels with a depth value outside the range $[-1..1]$ are clipped, see [Boa93]). Since a

shearing of the near clipping plane is undesired in most applications we also set i and j to m and n respectively to realign the near clipping plane (which in turn increases distortions of the far clipping plane). See Appendix A for more details.

Typically, a calibration matrix in our setup is close to the identity matrix like in this example:

$$\begin{pmatrix} 1-0.0420 & -0.0042 & 0 & 0.0175 \\ 0.0013 & 1-0.0363 & 0 & 0.0126 \\ -0.0006 & 0.0039 & 1 & 0 \\ -0.0006 & 0.0039 & 0 & 1 \end{pmatrix}$$

4 Calibration procedure

The general idea is to project a mouse cursor on the screen and just click on physical reference positions like the corners of the target screen. The computer then calculates the calibration matrix numerically so that the sum of the square distances of the transformed reference points to the selected positions is minimal.

To get precise physical locations of the reference points we use the corners of the projection. A normal mouse cursor wouldn't be visible at each corner and only allows pixel precise acquisition. Instead, for each corner, we use multiple lines at different angles that are projected and moved by the user into the corner one after another. The average of the pairwise intersections of these lines is used as the position for that corner with sub pixel precision. In our case it is not convenient to use a mouse so we chose a wireless game pad.

The calibration matrix is calculated by optimizing the values a, b, d, e, f, h, m and n with the steepest descent algorithm so that the sum of squared distances from projected points to their target location is minimal (see Appendix B for more details). Afterwards i and j are set to m and n respectively.

5 Using the Calibration

Whenever the projection matrix of the graphics hardware is changed we need to premultiply it with the calibration matrix so that the new projection matrix consists of the combination of the original projection matrix and the calibration matrix. In case of OpenGL the C code could look like this:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glMultMatrixf(C); // <-- new
gluPerspective(60, 1, 0.1, 100);
...
```

This method can also be used to modify aspect ratios or to align the content for tiled displays such as power walls. In the latter case additional modifications are necessary to handle blending and clipping of the overlapping regions.

6 Results

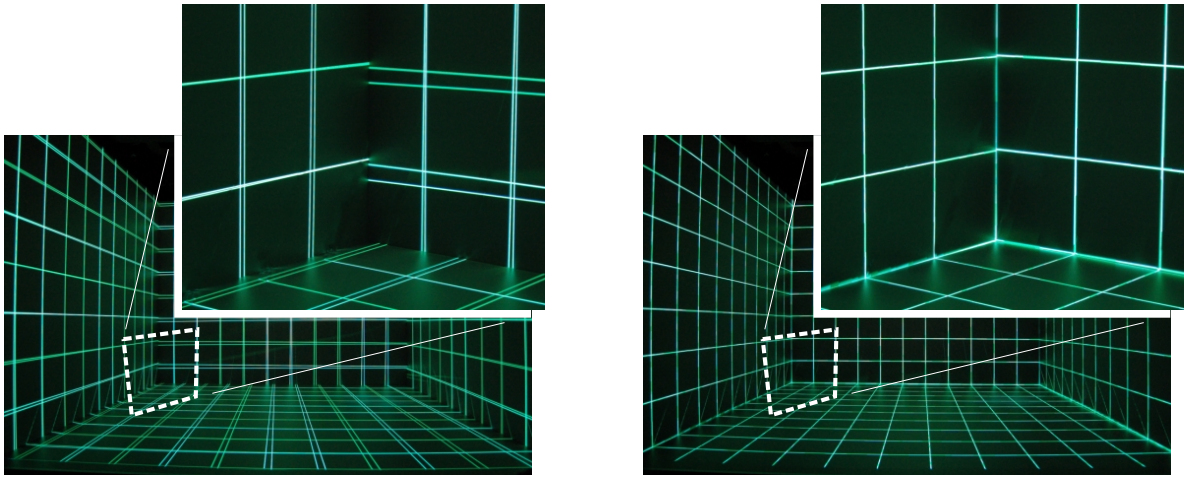


Figure 2: Stereo projections without (left) and with calibration (right) in our virtual environment. With little effort we managed to implement the calibration that greatly improves the visual result of the projection.

There are two major drawbacks with our approach. The first is that some pixels are wasted. The overlapping pixels at the border are projected onto the canvas. Hardware resolution and computing power is thus spent on pixels invisible to the user - however, typical 'waste' in our installation is in the order of 5%. The second drawback is that the calibration can only correct all distortion effects when the projection surface is planar.

The advantages are that

- only little effort and minimal modifications are necessary for 3D applications,
- the calibration process is significantly shortened and, thus, significantly less expensive,
- there is no influence on the performance and
- there is no pixel remapping that would lead to aliasing.

We successfully implemented the calibration for our virtual environment to drastically shorten the maintenance effort and improve the visual quality of the system (see Figure 2).

A Details on shearing of clipping planes

With introducing m and n in the calibration matrix

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ m & n & 0 & 1 \end{pmatrix}$$

the resulting depth value $q = z/w$ will be changed to $q' = z/(mx+ny+w)$.

Raskar addresses this problem by introducing $k = 1-|m|-|n|$ to keep the whole visible range from the case without calibration.

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & k & 0 \\ m & n & 0 & 1 \end{pmatrix}$$

$q = z/w$ becomes $q' = kz/(mx+ny+w) = (1-|m|-|n|)*z/(mx+ny+w)$

But still the near clipping plane is sheared resulting in strong artefacts with intersecting objects, especially with multiple projectors. Thus we decided to realign the front clipping plane with $i = m$ and $j = n$:

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ i & j & 1 & 0 \\ m & n & 0 & 1 \end{pmatrix}$$

$q = z/w$ now becomes $q' = (ix+jy+z)/(mx+ny+w) = (mx+ny+z)/(mx+ny+w)$ so that at the near clipping plane with $d = 1$ also $d' = 1$.

B Details on our calibration matrix optimization algorithm

To help understanding the algorithm and facilitate a reimplementation we provide this very simple source code that we use for the computation of the calibration matrix. Note that there are many possible ways like sophisticated automated approaches.

To use the following c++ code for the optimization you need to fill the array *corners* with the pixel positions transformed into the range $[-1,1],[-1,1]$. After calling the function $optimize()$ the result will be stored in *cm*.

```
// c++ source code snippet for a simple projector calibration,
// M. Lancelle, TU Graz

//   calibration matrix:
//   a b 0 d
//   e f 0 h
//   m n 1 0
//   m n 0 1

// all corners in -1 .. 1 coordinate system
struct pointStruct {double x, y;};
pointStruct dCorners[4] = {{-1,1}, {1,1}, {1,-1}, {-1,-1}};
pointStruct corners[4]; // new corners selected by user
double cm[8] = {1, 0, 0, 0, 1, 0, 0, 0}; // will hold variable
               // components of final matrix:  a, b, d, e, f, h, m, n

// 'cost' function to minimize
double calcValue(double a, double b, double d, double e, double f,
                 double h, double m, double n) {
  double distanceFromCorners = 0; // sum of squared distances
  for (int pnum=0; pnum<4; pnum++) {
    // transformed coordinates (should match 'corners' as good
    // as possible)
    double w = m*dCorners[pnum].x + n*dCorners[pnum].y + 1;
    double tx = (a*dCorners[pnum].x + b*dCorners[pnum].y + d) / w;
    double ty = (e*dCorners[pnum].x + f*dCorners[pnum].y + h) / w;
    double dst = tx - corners[pnum].x;
    distanceFromCorners += dst*dst;
    dst = ty - corners[pnum].y;
    distanceFromCorners += dst*dst;
  }
  return distanceFromCorners;
}

// Simple optimization:  step along steepest gradient with fixed step
// size.  Not brilliant but simple and works :)
void optimize() {
  double stepSize = 0.1;
  double grad[8];
```

```
  while (stepSize > 0.000001) {
    // compute gradient
    double currentValue = calcValue(cm[0], cm[1], cm[2], cm[3],
            cm[4], cm[5], cm[6], cm[7]);
    grad[0] = calcValue(cm[0] + 0.1*stepSize, cm[1], cm[2], cm[3],
            cm[4], cm[5], cm[6], cm[7]) - currentValue;
    grad[1] = calcValue(cm[0], cm[1] + 0.1*stepSize, cm[2], cm[3],
            cm[4], cm[5], cm[6], cm[7]) - currentValue;
    grad[2] = calcValue(cm[0], cm[1], cm[2] + 0.1*stepSize, cm[3],
            cm[4], cm[5], cm[6], cm[7]) - currentValue;
    grad[3] = calcValue(cm[0], cm[1], cm[2], cm[3] + 0.1*stepSize,
            cm[4], cm[5], cm[6], cm[7]) - currentValue;
    grad[4] = calcValue(cm[0], cm[1], cm[2], cm[3],
            cm[4] + 0.1*stepSize, cm[5], cm[6], cm[7]) - currentValue;
    grad[5] = calcValue(cm[0], cm[1], cm[2], cm[3], cm[4],
            cm[5] + 0.1*stepSize, cm[6], cm[7]) - currentValue;
    grad[6] = calcValue(cm[0], cm[1], cm[2], cm[3], cm[4], cm[5],
            cm[6] + 0.1*stepSize, cm[7]) - currentValue;
    grad[7] = calcValue(cm[0], cm[1], cm[2], cm[3], cm[4], cm[5],
            cm[6], cm[7] + 0.1*stepSize) - currentValue;

    // step in direction of gradient with length of stepSize
    double gradSquaredLength = 0;
    for (int i=0; i<8; i++) gradSquaredLength += grad[i]*grad[i];
    double gradLength = sqrt(gradSquaredLength);
    for (int i=0; i<8; i++) cm[i] -= stepSize * grad[i]/gradLength;

    stepSize *= 0.99;
  }
}
```

References

[BMY05] Michael Brown, Aditi Majumder, and Ruigang Yang. Camera-based calibration techniques for seamless multi-projector displays. *IEEE Transactions on Visualization and Computer Graphics*, 11:193 – 206, 2005.

[Boa93] Opengl Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley Publishing Company, 1993.

[BSdV01] R.G. Belleman, B. Stolk, and R. de Vries. Immersive virtual reality on commodity hardware. *Proceedings of the 7th annual conference of the Advanced School for Computing and Imaging*, 7:297–304, 2001.

[FH05] Gerald Farin and Dianne Hansford. *Practical Linear Algebra, A Geometry Toolbox*. A K Peters, Ltd., 2005.

[GWN+03] M. Gross, S. Würmlin, M. Naef, E. Lamboray, C. Spagno, A. Kunz, E. Koller-Meier, T. Svoboda, L. Van Gool, S. Lang, K. Strehlke, A. Vande Moere, and O. Staadt. blue-c: A spatially immersive display and 3d video portal for telepresence. *Proceedings of ACM SIGGRAPH 2003*, 22:819–827, 2003.

[LDMA+04] J.C. Lee, P.H. Dietz, D. Maynes-Aminzade, R. Raskar, and S.E. Hudson. Automatic projector calibration with embedded light sensors. *ACM Symposium on User Interface Software and Technology*, 1:123–126, 2004.

[LS04] B. Li and I. Sezan. Automatic keystone correction for smart projectors with embedded camera. *Proceedings of ICIP '04, International Conference on Image Processing 2004*, 4:2829– 2832, 2004.

[Ras00] Ramesh Raskar. Immersive planar display using roughly aligned projectors. *Proceedings of IEEE Virtual Reality, 2000*, 1:109–116, 2000.

[RB01] R. Raskar and P.A. Beardsley. A self-correcting projector. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2:504–508, 2001.

[SHVG02] Tomas Svodoba, Hanspeter Hug, and Luc Van Gool. ViRoom - Low Cost Synchronized Multicamera System and its Self-Calibration. *Pattern Recognition, 24th DAGM Symposium*, 24:515–522, 2002.

[SSM01] R. Sukthankar, R. Stockton, and M. Mullin. Smarter Presentations: Exploiting homography in camera-projector systems. *Proceedings of International Conference on Computer Vision*, 1:247–253, 2001.