BullsEye: High-Precision Fiducial Tracking for Table-based Tangible Interaction

Clemens Nylandsted Klokmose[1,2] clemens@cs.au.dk
Janus Bager Kristensen[3] jbk@cavi.au.dk
Rolf Bagge[3] rolf@cavi.au.dk
Kim Halskov[2,3] halskov@cavi.au.dk

[1]Department of Computer Science, Aarhus University, DK-8200 Aarhus N, Denmark
[2]Center for Participatory IT, Aarhus University, DK-8200 Aarhus N, Denmark
[3]Center for Advanced Visualization and Interaction, Aarhus University, DK-8200 Aarhus N, Denmark

ABSTRACT

This paper proposes a series of techniques for improving the precision of optical fiducial tracking on tangible tabletops. The motivation is to enable convincing interactive projection mapping on tangibles on the table, which requires a high precision tracking of the location of tangibles. We propose a new fiducial design optimized for GPU based tracking, a technique for calibrating light that allows for computation on a greyscale image rather than a binarized black and white image, an automated technique for compensating for optical distortions in the camera lenses, and a tracking algorithm implemented primarily in shaders on the GPU. The techniques are realized in the BullsEye computer vision software. We demonstrate experimentally that BullsEye provides sub-pixel accuracy down to a tenth of a pixel, which is a significant improvement compared to the commonly used reacTIVision software.

Author Keywords

Fiducial tracking; Computer vision; Tangible tabletops; Tangible computing

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

Over the last decade tabletop computing has gained significant interest in both academia and the industry. A popular technique for implementing tracking on an interactive table is to use computer vision, either by having one or more cameras mounted beneath a semi transparent surface, or by integrating optical sensors in a display surface[1]. Such optical tracking not only enables finger-based interaction, but also tracking of physical objects on the table, so-called tangibles. This is typically done by detecting shapes or unique fiducial markers attached to the bottom of the tangibles.

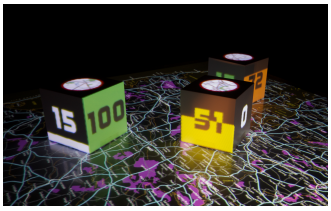

Figure 1: Interactive projection mapping on tangibles on an interactive tabletop.

A novel extension of tangible tabletops is to use interactive projection mapping on tangibles on the table [4], hereafter referred to as *tangible 3D tabletops*. This technique creates the illusion that the tangibles are display surfaces themselves (Figure 1). Although, in order to create and maintain a convincing illusion, the tracking of fiducials on the table must be extremely accurate.

ReacTIVision is one of the few freely available computer vision frameworks particularly designed for tabletop fiducial tracking[11][2]. To build custom tangible 3D tabletops, reacTIVision is currently the best candidate for fiducial tracking. However, we have found that the tracking technique applied in reacTIVision is too inaccurate for interactive projection mapping. Commercial alternatives to reacTIVision exist, such as Microsoft's PixelSense API or MultiTaction by MultiTouch Ltd but these frameworks are closed source and tightly coupled to specific hardware platforms.

In this paper we will present a computer vision framework, BullsEye[3], with a set of novel techniques for fiducial tracking to provide an accuracy high enough to convincingly enable interactive projection mapping on a tangible tabletop. While BullsEye is designed to enable tangible 3D tabletops it is also a framework for general tabletop fiducial tracking. Therefore, the presented improvements of fiducial tracking will benefit a regular tangible tabletop application based on optical tracking

[1]A technology used in Microsoft's PixelSense `http://www.microsoft.com/en-us/pixelsense`

[2]The fiducial tracking library *libfidtrack* from reacTIVision is also used in other frameworks such as in the Community Core Vision framework (`http://ccv.nuigroup.com`)

[3]BullsEye is available for download at `http://cavi.au.dk/bullseye`.

ITS 2014, November 16–19, 2014, Dresden, Germany.

ACM 978-1-4503-2587-5/14/11 ...$15.00.
http://dx.doi.org/10.1145/2669485.2669503

Figure 2: The main components of the tangible 3D tabletop. **1)** Bottom mounted projector for the surface of the table. **2)** Camera for tracking fingers and objects on the table. **3)** Objects on which the projection mapping is performed. The objects are equipped with fiducials and tracked by the bottom mounted camera. **4-5)** One or more top-mounted projectors for the projection mapping.

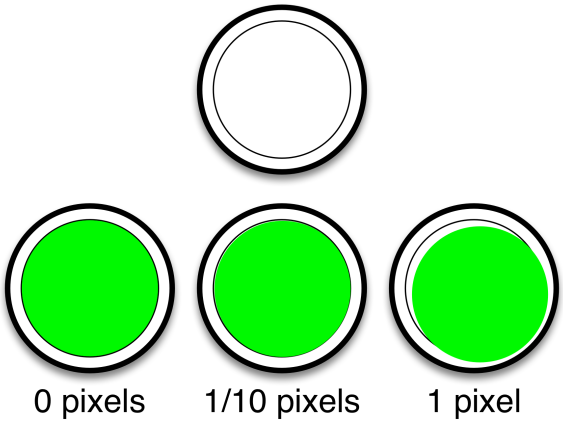

Figure 3: On a tangible similar in shape and size to a checker piece (top), a perfect projection of a green color is shown *bottom left*, *bottom right* shows how the projection with a worst case 1 pixel horizontal and vertical offset on a 100 cm long and 80 cm wide table and a camera with a resolution of 1280 by 1024 pixels. *Bottom middle* shows the worst case offset with the 1/10 pixel precision. On printed paper the offsets should approximately match what would be experienced on a physical table.

as well. The core advantage of BullsEye compared to reacTIVision is that BullsEye offers *subpixel precision* down to an average of one tenth of a pixel compared to the around one pixel precision we measure from reacTIVision. BullsEye furthermore facilitates significantly more precise and automated compensation for optical distortions in the optics of cameras. BullsEye does feature finger tracking, but in this paper we solely focus on the improved fiducial tracking to accommodate the precision required for interactive projection mapping.

In the next section we will present the requirements that spawned the need for a new tracking framework, followed by an overview of the central ideas employed in BullsEye and how they differ from those of reacTIVision. We will then present details on the implementation of BullsEye followed by an evaluation of its performance and compare it to reacTIVision.

Requirements

Interactive projection mapping on a tangible tabletop involves synchronizing fiducial tracking on the table with the rendering of a 3D scene displayed by one or more top-mounted projectors together with the image from the table projector or display (see Figure 2). To do this a virtual representation of an object must be mapped to the location and rotation of a physical object in real time. In figure 1 tangibles are used to navigate statistical data on a map by projecting the data relating to the position of the cubes on the map onto the cubes themselves.

In initial prototypes of tangible 3D tabletops we used reacTIVision for fiducial tracking. We experienced that the tracking accuracy was not high enough for this purpose. We have measured the error of reacTIVision to be between 0.5 and 1.5 pixels when detecting the position of a fiducial on a table with a 1280x1024 pixels camera input (See Experimental Evaluation). A one pixel error with a 1280x1024 camera input results in a metric precision of 0.78 mm when projecting on a 100 by 80 centimeter surface. This may seem as a high accuracy, and it is adequate for most tangible tabletop applications, but for the purpose of projection mapping the inaccuracy is easily discernible (as illustrated in figure 3). With BullsEye we aim for improving accuracy, and we demonstrate that a tenth of a pixel is a realistic aim. A tenth of a pixel offset is still discernible with the naked eye (as seen in figure 3 bottom middle) but a significant improvement over reacTIVision's accuracy.

Optical distortion in the lens of the camera used for tracking can result in the tracked position being offset from the actual physical position of the object. The offset may differ across the area of the table. In order to provide a uniform precision across the surface of the table, any optical distortion must be compensated for through a calibration. To align the tracking and compensate for optical distortions reacTIVision applies a technique where a set of grid points are manually aligned to a calibration grid that is printed and placed on the table (Figure 6). Our experience is that the process of manually aligning the grid is laborious and error prone, and even if the alignment is done extremely meticulously the compensation is too coarse. With BullsEye we aim to automate this calibration, and make sure the distortion compensation is even across the surface of the tangible tabletop.

It is essential that the latency between the physical movement of a tangible and the update of the position of the projected virtual representation is kept as low as possible. Therefore, a requirement for any fiducial tracking software is that the tracking must be achieved within the time between frames coming from the camera.

With BullsEye we also want to support multiple cameras, which can enable us to construct large interactive table-tops (inspired by [18]), and we want to support multiple different types of cameras including PixelSense based cameras.

FIDUCIAL TRACKING IN BULLSEYE

In this section we present three central techniques applied in BullsEye: A fiducial design optimized for GPU tracking, an improved calibration of light to enable computation on grayscale values for increased precision, and an improved technique for compensating for optical distortions. The techniques will be compared to those of reacTIVision. At the end of the section we will go through the details of the shader steps involved in BullsEye's fiducial tracking algorithm.

Fiducial design

The tracking algorithm in BullsEye is primarily implemented in shaders on the GPU in contrast to reacTIVision's traditional CPU-based algorithm. A GPU implementation has some significant performance benefits compared to a CPU implementation as tracking can be parallelized if the fiducial design allows for it.

Tracking a fiducial involves three basic steps:

1. Find the center coordinates
2. Find the rotation angle
3. Extract the unique identifier

The reacTIVision tracking algorithm [2] is based on a topological fiducial recognition approach adopted and improved from d-touch [3]. The center of a reacTIVision fiducial (figure 4 right) is computed by the average centroid of black and white leafs in the topology of the fiducial (i.e. by the centroids of the fully black and fully white dots). The rotation is computed as a vector from the center of the fiducial to the average centroid of the black leafs. The unique identifier of a fiducial is extracted by constructing a region adjacency graph from the fiducial and computing a depth sequence string which uniquely identifies the fiducial. In other words; reacTIVision creates a tree-like structure representing how black regions contains white and vice versa, which can be serialized to a unique identifier (find more details in [2]). The reacTIVision fiducial tracking algorithm could possibly be implemented on the GPU; however, we decided to opt for a simpler fiducial design particularly designed for GPU based tracking.

The BullsEye fiducials (figure 4 left) are designed so that the three tracking steps can be easily implemented with shaders executed on the GPU. A BullsEye fiducial consists of a central white dot surrounded by a solid black ring and one or more data rings again surrounded by a solid white ring inside a black ring with three white studs. The data rings encode the fiducial id in binary form (white is 1 and black is 0). Figure 4 (left) has a seven bit data ring with the id 0100001 in binary or 66 in decimal. In the outer black ring there is a stud pointing up right, and two pointing down left. The studs indicate the rotation of the fiducial and should visually be interpreted as an arrow passing through the fiducial showing the orientation. Hence, both orientation and id are fairly easily human readable. A ring-based shape was chosen as rings—given a center, radius and ring width—are easily identifiable on the GPU with relatively few texture lookups. This is advantageous since texture lookups are expensive operations. BullsEye fiducials are initially found by searching for white dots surrounded by a white ring with a fixed width at a fixed radius. By computing the center of the fiducial, the rotation can be computed by sampling for the position of the three white studs. Finally by knowing the rotation, the id can be read from the data rings in the fiducial.

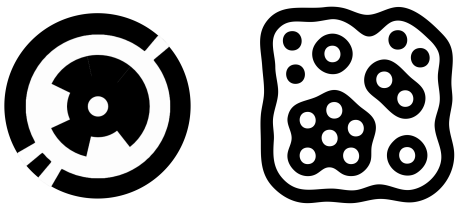

Figure 4: A BullsEye fiducial (left) and a reacTIVision fiducial (right) (reprinted with permission)

Grayscale representation

One of the initial steps of the reacTIVision tracking algorithm is to perform binary thresholding on a grayscale version of the input image; given a threshold a grayscale pixel is either made black or white. We want to avoid binarization as it effectively means throwing away information and hereby loosing accuracy (as illustrated in figure 5).

To compute the center of the grayscale represented rectangle in figure 5(b), one can use a average of the center positions of the pixels weighted by their greyscale values, which in perfect conditions would give an accuracy proportional to the bit depth of the greyscale image. In the less than ideal conditions that we have when tracking fiducials on a real table, the grayscale image that we get from the camera image is noisy and influenced by background light and reflections. Noise resulting in a variation at a pixel of only a few grayscale values can through binarization result in a pixel changing from black to white; hereby offsetting the tracked center of a fiducial quite significantly. When computing the center from a weighted average of grayscale values, the same noise of a few grayscale values at a pixel will only have a minimal effect on the tracking. However, different lighting condition across the table can result in an erroneous position when based on a weighted average: If a fiducial is partially in a brighter lit area of a table, the position may be slightly offset towards the brighter lit area.

Similar to reacTIVision, BullsEye compensates for variations in lighting conditions across the table. ReacTIVision uses a snapshot of the background light on an empty table to compute a binarization threshold of the image segmented into 32 by 32 pixel tiles. In BullsEye we normalize the greyscale spectrum *per pixel* through calibration of light. The calibration involves continuously storing the brightest white and darkest black seen at a pixel in a texture and subsequently using that texture as input to a normalization of the grayscale values of the camera image during tracking. The darkest

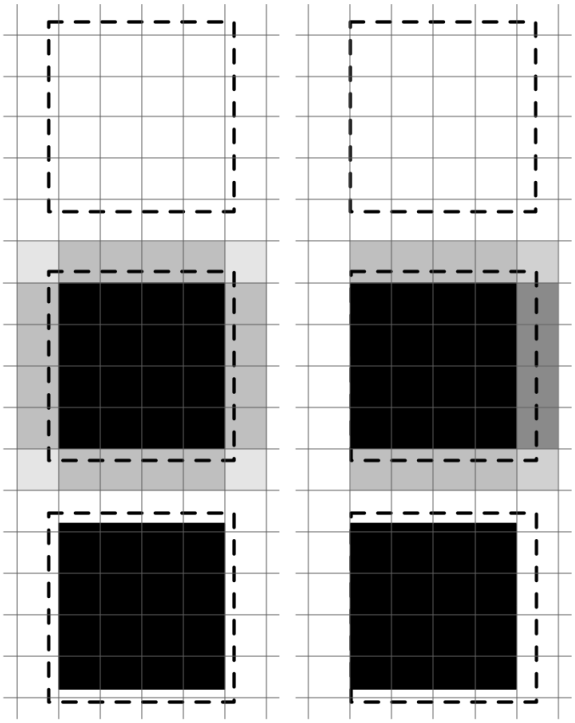

Figure 5: **Top**: The dotted rectangle illustrates how a physical rectangle on a table corresponds to the pixels of the camera image (the background grid). **Center**: Grayscale representation of the camera input. **Bottom**: Binary representation of the camera image. The rectangle has moved approx. a fifth of a pixel from **a** to **b**. Notice how the binary representation stays the same from a to b.

black is provided by an empty table, while the brightest white can be provided by moving a white object across the whole area of the table. The grayscale calibration texture for a table with two cameras can be seen in figure 9.

The result is that in BullsEye we can rely on computing weighted averages, hence achieving sub-pixel precision with a high noise tolerance.

Compensation for optical distortion

Similar to reacTIVision, BullsEye uses a printed calibration grid as input to the geometry calibration, but the alignment to the grid is automated rather than manual. The BullsEye calibration grid (Figure 7) consists of black dots in a uniform grid on a white background. By tracking the location of the black dots in the uniform grid on the printed sheet a position mapping texture can be produced that subsequently is used to correct the image from the camera.

The position mapping texture is computed in four steps implemented as a CPU algorithm: In the first step the user is prompted to click on a part of the input image showing the grid. We now perform a calibration of light where we search for the darkest and lightest spots on the image in a predefined area surrounding the user's click. This calibration is

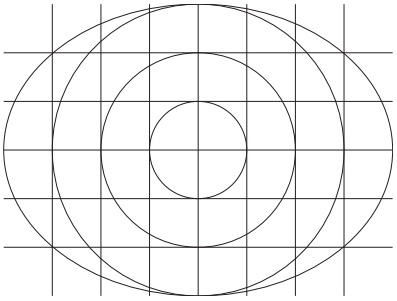

Figure 6: reacTIVision's geometry calibration grid

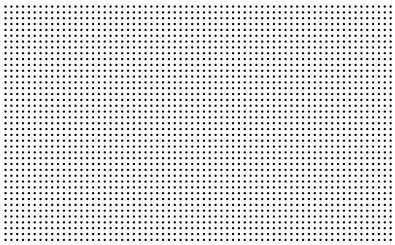

Figure 7: BullsEye's geometry calibration grid

used to compute a binarization threshold of the image to distinguish dots from the white background. In the second step we perform a blob extraction to track all the black dots on the printed grid. We apply a blob extraction algorithm inspired by connected-component labeling [6]. In step three we iterate through all the extracted blobs and for each blob we search for its immediate neighbor blobs to the left, right, top and bottom. We can now store the average distance between blobs and produce a data structure representing a uniform virtual grid of dots, where each dot stores its corresponding position in the original image. This grid may not cover the whole input image, or there may be holes in the grid. To compensate for this, the user can choose to automatically expand the grid. Through interpolating between the positions of neighboring blobs in the virtual grid we in step four generate a mapping texture. For each pixel corresponding to a pixel in a distortion corrected image, the mapping texture stores the position of where the pixel value should be looked up in the original camera input. The distortion corrected image can be combined from multiple camera inputs, hence the origin of the virtual grid is user configurable. The mapping texture is used as input to the shaders in the rendering pipeline as described below.

Tracking implementation

BullsEye does realtime image processing using shaders on the GPU to manipulate images in realtime. The core of the tracking algorithm in BullsEye is implemented as subsequent shaders in a rendering pipeline. BullsEye is implemented in Java and the OpenGL Shader Language [16].

Pixel shaders are pieces of code executed per pixel on an output texture. For each pixel the shader can sample in one or more input textures to produce the output color of the pixel.

Figure 8 illustrates the shader steps involved in the rendering pipeline of BullsEye. The input texture to the first shader

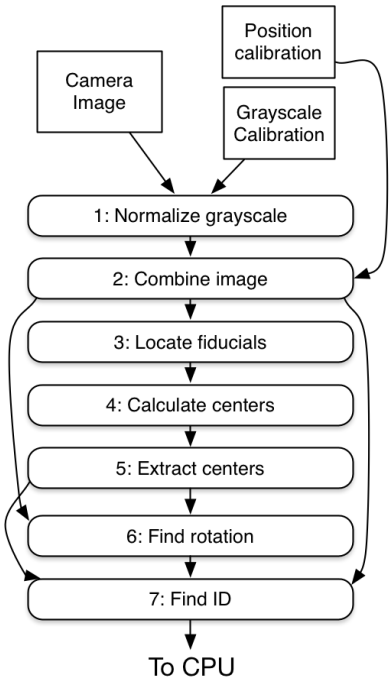

Figure 8: BullsEye's shader steps

Normalize Grayscale is the image from the camera converted to grayscale and the texture representing the grayscale calibration. Each pixel is normalized given the maximum and minimum light levels recorded in the calibration. Shader 2 combines the normalized images from one or more cameras into a combined image based on the per-camera distortion correction mapping texture which contains information about where a given pixel from the source image should be mapped to in the combined image. When combining camera inputs, overlap is handled by only using the input from one of the cameras in the overlapping area. The actual textures involved in the first two shader steps in a setup with a large table with two cameras is shown in figure 9.

To locate a BullsEye fiducial, shader 3 outputs the likelihood that a pixel contains the center of a fiducial. This is done by sampling at known distances in a circle around the given pixel and computing a color difference between what should be black and what should be white if the pixel contained the fiducial center; if we assume a pixel is the centre, we know the distance to the rings comprising the BullsEye fiducial and can compute and average difference between what should be black and what should be white. Listing 1 shows pseudocode for the BullsEye implementation of shader step 3. The output of this shader is a texture where the red color intensity represents the likelihood of the pixel being the center of a fiducial. The pixel with the highest red color intensity is assumed to contain the center of the fiducial. An input as shown in figure 10 (left) will produce an output as shown in figure 10 (right). The blue component represents the accumulated color difference in the outer white ring and inner black ring, and is used to filter off false positives.

Listing 1: Pseudo-code for locating fiducials (shader 3). *Input*: Combined and normalized image ($combinedTexture$) and the current texture coordinates ($textureCoords$). The constant $RADIUS$ is the radius to the outer edge of the outer white ring which is given through user configuration, and the constant $RING\_WIDTH$ is the width of a ring. *Output*: Likelihood of pixels containing the center of a fiducial

```
texColor = combinedTexture.colorAt(textureCoords)

if texColor.red < THRESHOLD:
  return BLACK
else:
  outsideSampleRadius = RADIUS + RING_WIDTH / 2
  outerSampleRingRadius = RADIUS - RING_WIDTH / 2
  innerSampleRingRadius = RING_WIDTH

  totalColor = totalCenterDifference = 0
  previousCenterColor = combinedTexture.colorAt(
       textureCoords + innerSampleRingRadius).red
  previousOutsideColor = combinedTexture.colorAt(
       textureCoords + RADIUS * RING_WIDTH * 1.5).red

  for i = 0; i<32; i++:
    radians = i * (2.0 * PI) / 32;
    dirVector = (cos(radians) * (1 / combinedTexture.size.
         x), sin(radians) * (1 / combinedTexture.size.y))

    outside = textureCoords + outsideSampleRadius *
         dirVector
    outerRing = textureCoords + outerSampleRingRadius *
         dirVector
    innerRing = textureCoords + innerSampleRingRadius *
         dirVector

    outSideColor = combinedTexture.colorAt(outside).red
    outermostColor = combinedTexture.colorAt(outerRing).
         red
    innermostColor = combinedTexture.colorAt(innerRing).
         red

    totalColor += min(outermostColor - outsideColor,
         texColor.red - innermostColor)
    totalCenterDifference += abs(innermostColor -
         previousCenterColor)+abs(outsideColor-
         previousOutsideColor)
    previousCenterColor = innermostColor
    previousOutsideColor = outsideColor
  totalColor /= 32
  totalCenterDifference /= 32

  return (max(totalColor, 0), 0, totalCenterDifference)
```

Shader 4 samples the center likelihood texture produced by shader 3 in an area around the given pixel coordinates and computes an average center position weighted by the likelihood value of the sampled pixels. If a sampled pixel has a higher center likelihood than the pixel at the given coordinates in the likelihood texture, the output pixel at those coordinates will be black. This way the output texture will contain only one pixel with color per tracked fiducial, where the color represents the center position of the fiducial[4]. Listing 2 shows the pseudocode for shader 4. The result of shader 4 is a primarily black texture. In the example of figure 9 shader 4 would produce a very large texture with just 8 pixels actually containing relevant information.

[4]In the unlikely situation that two pixels have the exact same floating point likelihood value, a filtering ensuring that no fiducials overlap will be applied on the CPU at a later stage.

Listing 2: Shader 4 pseudo-code. *Input*: Center likelihood texture ($likelihoodTexture$) and the current texture coordinates ($textureCoords$). Output: a texture with centers of fiducials in sub-pixel precision.

```
texColor = likelihoodTexture.colorAt(textureCoords)

circleLikeliness = texColor.red
edgeDiff = texColor.blue

if circleLikeliness <= CIRCLE_LIKELINESS_THRESHOLD ||
     edgeDiff >= FALSE_POSITIVE_FILTER_THRESHOLD:
        return BLACK
else
  pixelSize = 1 / likelihoodTexture.size
  position = vec2D(0.0,0.0)
  totalSum = 0

  for y = -SEARCH_AREA; y <= SEARCH_AREA; y++:
    for x = -SEARCH_AREA; x <= SEARCH_AREA; x++
      sampleCoord = textureCoord + vec2D(x , y) *
           pixelSize
      sampleColor = max(0, likelihoodTexture.colorAt(
           sampleCoord).red)

      if sampleColor > circleLikeliness:
        return BLACK

      position += samplePosition * sampleColor
      totalSum += sampleColor

  position /= totalSum
  totalSum /= pow(2.0 * SEARCH_AREA + 1.0, 2.0)

  return (position.x, position.y, totalSum);
```

As an optimization we apply a geometry shader in shader step 5 to compress the full sized texture produced by shader 4. The compressed output texture contains a block representation of the whole surface. Each row of pixels in the output texture represents a block of 64x64 pixels of the input texture and stores the position of up to eight fiducial centers contained in the block. Color values represent the coordinates of the fiducial centers. Therefore, for an input texture of 1024x768 pixels, the compressed texture will be 192x8 pixels, hence making the subsequent shader steps significantly faster. The maximum of eight centers per block is set to reflect the physical limitations of how many fiducials that can be placed within the area corresponding to 64x64 pixels in the camera image.

Given the centers from shader 5, the rotations are found by sampling the the block representation of the centers and the combined image from shader 2. The output of shader 6 is a block representation like that of shader 5 but containing orientations. This shader step is the most computationally intensive step, but the computations are only performed on pixels representing a fiducial. To compute the orientation a coarse estimate is found by iterating a full circle in 256 steps, and for each iteration computing a likelihood score for the given angle being the approximate orientation of the fiducial. The likelihood score is computed as the difference between the color where the white studs should be and the outer black ring. Given the coarse estimate a precise orientation is computed by iterating 20 degrees around the coarse estimate in 100 steps and computing an average weighted by the orientation likelihood at each step. Listing 3 shows the pseudocode

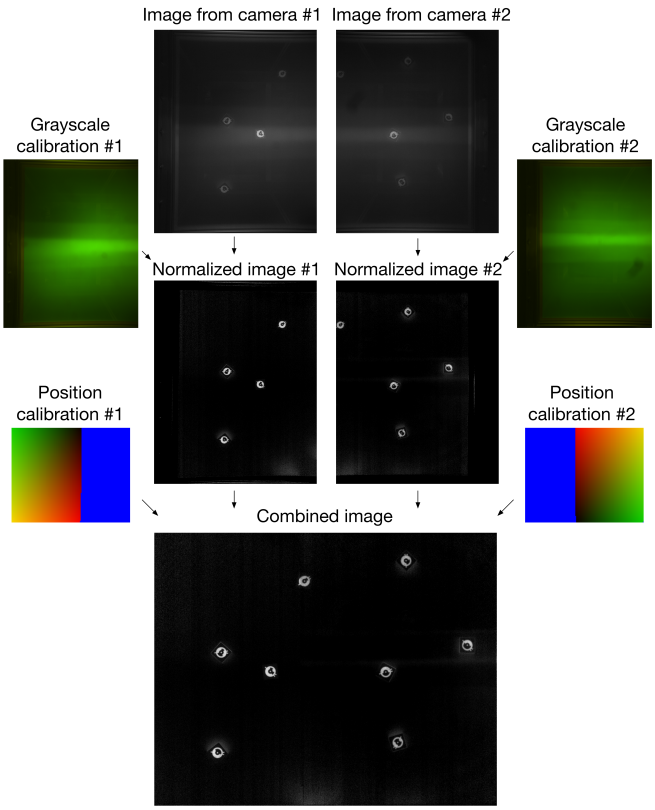

Figure 9: BullsEye's initial image processing

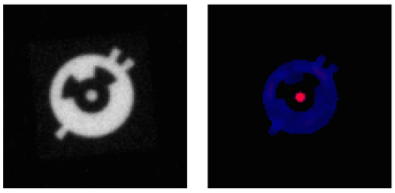

Figure 10: BullsEye's circletracking

for this shader step. The rotation could in principle be computed from a single stud, however, we opted for the fiducial design with multiple studs to increase accuracy and noise tolerance.

The final shader 7 can use the centers, rotations and the combined image to find the id of a fiducial and output a texture encoding positions, rotation and ids of fiducials.

The resulting texture is extracted to the CPU, where it is parsed and filtering and stabilization is performed. Finally the tracking data is translated to TUIO events [12] and emitted to a given application hostname and port.

BullsEye User Interface

BullsEye features a user interface that enables real-time tweaking of the parameters of the tracking (Figure 11). Furthermore the user interface allows for easy calibration of both light and geometry. Input from multiple cameras can be com-

Listing 3: Shader 7 pseudo-code. Input: Extracted centers and combined, normalized image. Output: Rotations of fiducials in sub-pixel precision.

```
center = texture.colorAt(textureCoords)

function getOrientationLikeliness(radians, center):
  /* Sample color in combined texture at front stud,
       besides front stud, at the two back studs and
       between the back studs relative to given radians
       and center */

  return min(frontStudColor - besidesFrontStudColor, (
       backStud1Color+backStud2Color)/2.0 -
       betweenBackStudsColor)

//Get a course orientation estimate
maxLikeliness = 0
bestRotation = 0
for int i = 0; i<256; i++:
  radians = i * (2.0 * PI) / 256
  likeliness = getOrientationLikeliness(radians, center)

  if currentValue > maxValue:
    maxLikeliness = likeliness
    bestRotation = rotation

//Find precise orientation
startRadians = bestRotation - 10 * (180/PI)
endRadians = bestRotation + 10 * (180/PI)

for float r = startRadians; r < endRadians; r+= (2.0 * PI)
      / 100:
  likeliness = getOrientationLikeliness(r, center)
  rotationWeightedSum += r * likeliness
  totalLikeliness += likeliness

weightedRotationAverage = rotationWeightedSum /
     totalLikeliness

return (weightedRotationAverage, 0, 0)
```

bined and their placement in relation to each other can be configured visually.

Limitations

BullsEye requires that the tag size is known, and it only operates with a single tag size at a time. ReacTIVision is agnostic towards tag size because of its topological fiducial recognition algorithm. It is possible to extend the BullsEye algorithm with support for multiple tag sizes. This would require the different supported tag sizes to be specified through configuration. Multiple tag sizes would have a drawback on performance since each tracking step using the tag size as input would have to be repeated per different tag size.

BullsEye employs a geometry shader for one of the shader steps, and only graphics cards supporting Open GL 3.2 or newer has support for geometry shaders. This leaves out running BullsEye on a credit-card computer such as the Raspberry Pi, but it will run on most popular integrated graphics cards such as the Intel HD Graphics series[5].

EXPERIMENTAL EVALUATION

In order to provide a ground truth for the positions and rotations of fiducials, we simulated a camera input where we could control the position of a fiducial. In order to compare the precision of BullsEye with reacTIVision we generated videos for both types of fiducials. Figure 12 shows a snapshot of the video with a BullsEye fiducial. These videos were fed to reacTIVision and BullsEye as were they input from a camera.

The videos were generated using Blender[6]. In Blender we modeled a 3D scene consisting of a camera placed below a flat surface together with two light sources. On the flat surface we placed a rectangle with a fiducial texture, and animated the fiducial diagonally across the table on a known path with a fixed rotation. The animation consisted of 600 frames over a period of 30 seconds (20 frames per second). The videos were generated with a resolution of 1280x1024 pixels.

We furthermore generated a video where the fiducial was shrunk over time, to provide data on how small a fiducial BullsEye and reacTIVision are capable of tracking[7].

To simulate an image from a real camera we added random noise and warped the geometry based on a lens profile from a

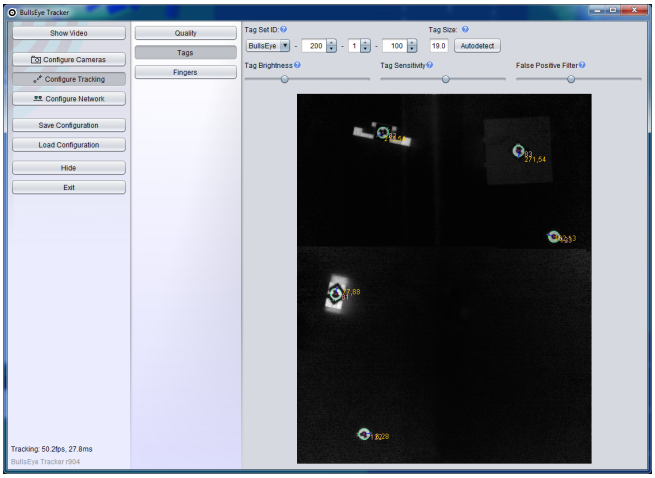

Figure 11: The user interface for BullsEye showing configuration options for fiducial tracking.

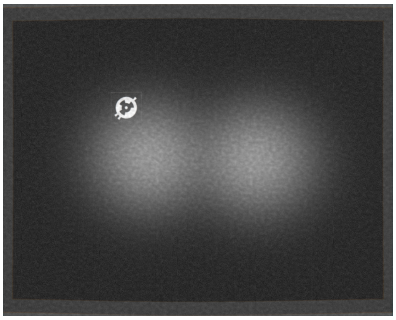

Figure 12: Snapshot from simulated camera input for BullsEye.

[5] http://www.intel.com/support/graphics/sb/CS-033757.htm

[6] http://www.blender.org

[7] For BullsEye we manually reconfigured the size of fiducials per frame.

Attribute	Mean	Standard deviation	Maximum
X	$0.37px$	$0.3px$	$1.28px$
Y	$1.35px$	$0.76px$	$2.9px$
Rotation	$0.54°$	$0.31°$	$1.32°$

Table 1: reacTIVision Absolute Error

Attribute	Mean	Standard deviation	Maximum
X	$0.08px$	$0.06px$	$0.29px$
Y	$0.07px$	$0.05px$	$0.27px$
Rotation	$0.12°$	$0.09°$	$0.48°$

Table 2: BullsEye Absolute Error

real camera[8]. When rendering the video from the 3D scene, the output was anti-aliased using a Mitchell-Netravali [14] filter.

To calibrate reacTIVision we used a simulated video input showing the reacTIVision calibration grid. The calibration of reacTIVision was done manually with our best possible efforts. To calibrate BullsEye we similarly used a simulated video input showing the BullsEye calibration grid, and let BullsEye calibrate automatically. The black calibration was performed with a simulated video showing an empty table, and for the white calibration a pure white rectangle was inserted on top of the table in the 3D scene.

In order to provide a fair comparison between the tracking of reacTIVision and BullsEye, we disabled a post-tracking stabilization step in the reacTIVision code that would filter out sub-pixel changes to a tracked fiducial position and rotation changes below three degrees.

Results

Figures 14, 13, and 15 show the error rates of reacTIVision and BullsEye for X-coordinates in pixels, Y-coordinates in pixels and rotation in degrees, respectively.

Table 1 shows statistics for the absolute error rates of reacTIVision, while table 2 show the statistics for BullsEye.

We see that BullsEye on average provides subpixel precision down to below a tenth of a pixel, and the maximum error rate measured were below a third of a pixel. The error rate of rotation is down to around a tenth of a degree in average with a maximum of half a degree.

Compared to reacTIVision, BullsEye is significantly more accurate with a 5 times more accurate average X-coordinate tracking, a 20 times more accurate average Y-coordinate tracking, and on average 4.5 times more accurate rotation tracking. Furthermore, the standard deviations in the error rates of BullsEye are significantly lower than those of reacTIVision.

[8]We used the lens profile from a Sony E-mount 18-200mm f/3.5-6.3 zoom lens.

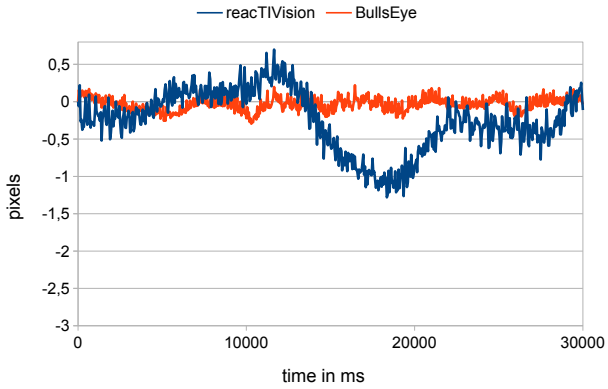

Figure 13: X-Coordinate error

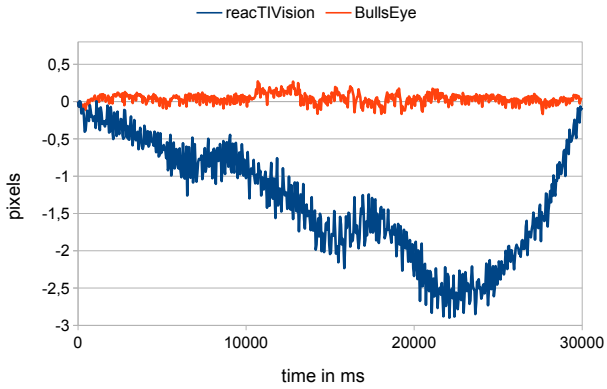

Figure 14: Y-Coordinate error

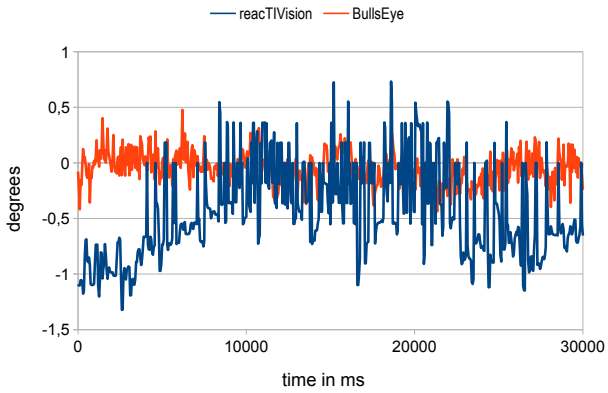

Figure 15: Rotation error

When tracking shrinking tags BullsEye lost tracking of a fiducial when the size of the fiducial (including padding) got below 29 pixels in diagonal size. This corresponds to 1.9% of the table diagonal or 2.4cm in diagonal on a 100 times 80 cm table. In comparison reacTIVision lost tracking of a fiducial when the size got below 57.3 pixels in diagonal which corresponds to 4.7 cm a similar sized table. It should be noted that reacTIVision in our initial measurements could track fiducial sizes equally as small as BullsEye. This, however, was a result of reacTIVision assuming that if it detects anything at the same location in consecutive frames it has the same id.

Discussion

In the generated video we used a standard 16:9 aspect ratio. This means that the resolution of Y-coordinates is almost half the resolution of X-coordinates. Our data therefore indicates that BullsEye is significantly more accurate on lower resolutions than reacTIVision. This may be attributed to a combination of a more inaccurate calibration of reacTIVision at lower resolutions, and that the effects of binarization are emphasized at lower resolutions.

The measurements of the X and Y coordinates are taken as the fiducials moved diagonally across the surfaces of the table. Curves in the coordinate plot therefore implies inadequate compensation for optical distortion. It is notable that the BullsEye plots of the X and Y coordinates do not curve like the reacTIVision plots do. The error rates of reacTIVision are products of the tracking algorithm combined with the less than optimal lens distortion calibration. An inaccurate calibration can explain the curves in the plots of the error rates for reacTIVision. Given a perfect geometric calibration of reacTIVision, which in theory should remove the curves in the error rates, the overall amplitude of the error rate in BullsEye would still be significantly lower than reacTIVision's.

The presented results are products of a simulated video input. We were not able to provide a ground truth for computing absolute error rates for tracking on a real video input. We have, however, observed similar improvements in relative tracking accuracy from reacTIVision to BullsEye by tracking on real video from one of our tabletops as those we see on our simulated input. Here a reacTIVision and BullsEye fiducial was placed side-by-side on a single tangible that was then moved across the tabletop.

EXPERIENCES AND PERFORMANCE

Besides being used in numerous research prototypes and student projects BullsEye has been used for two full scale tangible 3D tabletop installations deployed in the wild: Tangible Urban Planning demonstrates how tangible 3D tabletops can support collaborative activities in urban planning and development projects [5] (figure 16, top). Projected Play is a tangible 3D tabletop installation developed for LEGO World, a four-day in-door entertainment held in the city of Copenhagen and attracting more than 40.000 visitors in 2013 [9] (figure 16, bottom left). BullsEye has also been used for more traditional tangible tabletop applications such as the RADAR Table (figure 16, bottom right); a tangible music interface recently deployed at the 2011 SPOT music festival in Aarhus, Denmark, as well as been presented at numerous conventions and conferences [10].

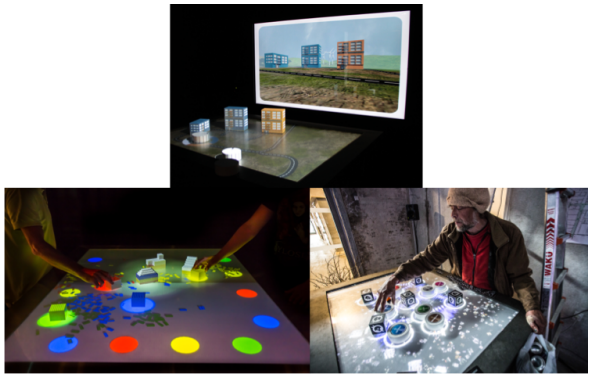

Figure 16: Examples of projects using BullsEye: Tangible Urban Planning (top), Projected Play (bottom left) and the RADAR table (bottom right).

We have used BullsEye for multiple different sized tabletops, both small with a single camera for tracking and larger with two cameras. We have implemented a PixelSense driver for BullsEye and installed BullsEye on our lab's Samsung SUR40 table. While we haven't been able to perform a similar comparison with the PixelSense tracking using simulated input as we did with reacTIVision, our experience is that the tracking on the SUR40 when using BullsEye is more accurate, and more robust to changing light conditions.

We have not performed a systematic evaluation of the performance of BullsEye. However, we have not experienced that BullsEye is a performance bottleneck in our tangible 3D tabletops. The largest interactive table of our lab uses two 50 frames per second input cameras with a combined resolution of 2560 by 1024 pixels. Running on a consumer grade PC[9], BullsEye can easily keep up with the input from the cameras (i.e. perform the tracking within 20 ms). We have had more than 100 fiducials placed simultaneously on this particular table without any significant decrease in tracking performance.

RELATED WORK

Improving the precision of fiducial tracking for interactive tabletops has not gained significant interest. However, some improvements of the original reacTIVision algorithm have been proposed. Topolo Surface [15] combines the topological approach of D-Touch [3] and reacTIVision [11] with an angle based encoding of fiducial IDs. The primary goal is to simplify the fiducial generation, allow for a wider ID range, and to encode checksums of IDs in the fiducials. The angle based encoding of fiducial IDs is similar to the one in BullsEye, and BullsEye could easily be extended with checksums on fiducial IDs. Topolo Surface is CPU based, and relies on a binarization step similar to that of reacTIVision. Unfortunately the paper does not evaluate the precision of the tracking, nor is the software available for comparison.

[9]Intel Core i7-3770 3.4GHz CPU, Nvidia GeForce GTX680 GPU and 4gb RAM

LightTracker [8] is a software framework for manipulating the image-processing pipeline at runtime particularly designed for touch, but can be extended for fiducial tracking as well (e.g. by integrating libfidtrack from reacTIVision). LightTracker provides a manual calibration with some attention to possible lens distortion. The light calibration technique of LightTracker is based on a similar idea to the one applied on BullsEye using an intensity map. The tracking in LightTracker is CPU based, although designed to exploit multiple threads to increase performance.

Fiducial tracking is not only used for tangible tabletops, but central in augmented reality [1] as well [13, 7]. For augmented reality, fiducials must be tracked in three dimensions with a movable camera, hence the tracking algorithm is quite different as for tabletop tracking and must compute distance to fiducials and their rotation in an extra dimension. Shibata and Yamamoto [17] demonstrate how to achieve subpixel precision in tracking fiducials for augmented reality using edge detection on the GPU. The paper benchmarks against the tracking in ARToolkit [13], and achieves similar improvements in precision as when we compare BullsEye to reacTIVision.

CONCLUSION

In this paper we have presented a series of techniques for improving the precision of optical fiducial tracking on tangible tabletops implemented as part of the BullsEye computer vision software. We have demonstrated that subpixel precision down to a tenth of a pixel can be achieved through a combination of a GPU based tracking algorithm, a fiducial design optimized for GPU tracking, a calibration of light allowing for computation on grayscale values and an automated technique for compensating for optical distortions in camera lenses.

ACKNOWLEDGEMENTS

We thank the staff at CAVI: Jonas Oxenbøll Petersen for work on the applications of BullsEye, and Peter Friis for his hardware expertise. We thank Henrik Korsgaard and Brian Bunch Christensen for comments and critique. This research has been supported by the interdisciplinary research center for Participatory IT at Aarhus University.

REFERENCES

1. Azuma, R. T., et al. A survey of augmented reality. *Presence 6*, 4 (1997), 355–385.
2. Bencina, R., Kaltenbrunner, M., and Jorda, S. Improved topological fiducial tracking in the reactivision system. In *Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*, IEEE (2005), 99–99.
3. Costanza, E., Shelley, S. B., and Robinson, J. D-touch: A consumer-grade tangible interface module and musical applications. In *Proceedings of Conference on Human-Computer Interaction (HCI03)*, Springer (2003).
4. Dalsgaard, P., and Halskov, K. Tangible 3d tabletops: combining tangible tabletop interaction and 3d projection. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*, ACM (2012), 109–118.
5. Dalsgaard, P., and Halskov, K. Tangible 3d tabletops. *interactions 21*, 5 (2014), 42–47.
6. Dillencourt, M. B., Samet, H., and Tamminen, M. A general approach to connected-component labeling for arbitrary image representations. *J. ACM 39*, 2 (Apr. 1992), 253–280.
7. Fiala, M. Artag, a fiducial marker system using digital techniques. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 2, IEEE (2005), 590–596.
8. Gokcezade, A., Leitner, J., and Haller, M. Lighttracker: An open-source multitouch toolkit. *Computers in Entertainment (CIE) 8*, 3 (2010), 19.
9. Halskov, K., Dalsgaard, P., and Stolzes, L. Analysing engaging experiences with a tangible 3d tabletop. In *Proceedings of Advances in Computer Entertainment (ACE) 2014 (to appear)*, ACM (2014).
10. Hansen, N. B., and Halskov, K. Material interactions with tangible tabletops: a pragmatist perspective. In *Proceedings of the 8th Nordic Conference on Human-Computer Interaction (to appear)*, ACM (2014).
11. Kaltenbrunner, M., and Bencina, R. reactivision: a computer-vision framework for table-based tangible interaction. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, ACM (2007), 69–74.
12. Kaltenbrunner, M., Bovermann, T., Bencina, R., and Costanza, E. Tuio: A protocol for table-top tangible user interfaces. In *Proc. of the The 6th Int'l Workshop on Gesture in Human-Computer Interaction and Simulation* (2005).
13. Kato, H., and Billinghurst, M. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Augmented Reality, 1999.(IWAR'99) Proceedings. 2nd IEEE and ACM International Workshop on*, IEEE (1999), 85–94.
14. Mitchell, D. P., and Netravali, A. N. Reconstruction filters in computer-graphics. In *ACM Siggraph Computer Graphics*, vol. 22, ACM (1988), 221–228.
15. Nishino, H. Topolo surface: A 2d fiducial tracking system based on topological region adjacency and angle information. *Journal of Information Processing 18* (February 2010), 16–25.
16. Rost, R. J. *OpenGL shading language*. Addison-Wesley Professional, 2004.
17. Shibata, N., and Yamamoto, S. Gpgpu-assisted subpixel tracking method for fiducial markers. *Journal of Information Processing 22*, 1 (2014), 19–28.
18. Wang, S., Bevans, A., and Antle, A. N. Stitchrv: multi-camera fiducial tracking. In *Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction*, ACM (2010), 287–290.