Particle Filtering with Rendered Models: A Two Pass Approach to Multi-object 3D Tracking with the GPU

Erik Murphy-Chutorian and Mohan M. Trivedi
Computer Vision and Robotics Research Laboratory
University of California, San Diego
{erikmc,mtrivedi}@ucsd.edu

Abstract

We describe a new approach to vision-based 3D object tracking, using appearance-based particle filters to follow 3D model reconstructions. This method is targeted towards modern graphics processors, which are optimized for 3D reconstruction and are capable of highly parallel computation. We discuss an OpenGL implementation of this approach, which uses two rendering passes to update the particle filter weights. In the first pass, the system renders the previous object state estimates to an off-screen framebuffer. In the second pass, the system uses a programmable vertex shader to compute the mean normalized cross-correlation between each sample and the subsequent video frame. The particle filters are updated using the correlation scores and provide a full 3D track of the objects. We provide examples for tracking human heads in both single and multi-camera scenarios.

1. Introduction

Vision-based 3D-tracking is a key requirement for new applications in surveillance, markerless motion capture, human-computer interaction, and many more. In this paper, we introduce a new real-time procedure for 3D tracking of multiple objects in one or more cameras by utilizing the capabilities of a modern GPU and standards-based OpenGL [1]. This approach uses an appearance-based particle filter to follow each object in an augmented reality that mimics the view-space of one or more cameras. Each object is represented as a texture-mapped 3D model that can be rotated, translated, and rendered to an image. By generating many transformations of each object and comparing them to the observed video frames, the approach can robustly track the movement of each object.

From a real-time vision standpoint, the major challenge of this approach is the ability to generate and evaluate enough samples per image frame to adequately approximate the posterior density at a fast enough framerate to prevent losing track. For real-time framerates and hundreds of particle samples per frame, this requires the system to render a texture-mapped 3D model and provide an expensive image comparison operation thousands of times per second. This level of complexity requires far more computational power than is currently available from a conventional Central Processing Unit (CPU). In contrast, this type of computation is ideally suited for the parallel processing capabilities of a modern programmable Graphics Processing Unit (GPU). The sample generation step requires that a virtual object model be translated, rotated, than rendered as one of more projective images. This is intended usage of a GPU, which contains a highly-optimized, fixed-function pipeline for transforming textured 3D-polygons into a 2D framebuffer. Typically the intended output is a computer display, but recent graphics hardware can alternatively render to off-screen buffers in texture memory.

Once each sample has been rendered, it must be compared to the observed video. Bandwidth limitations make it infeasible to read thousands of samples into CPU memory every second, and CPU resources would be heavily strained by any non-trivial comparison of this much data. We describe a novel solution to both these problems by performing the image comparisons with the massively-parallel programmable pipeline of a modern GPU. Using this approach on a Nvidia GeForce 8800 GT video card, we are able to track an object with over 6000 samples per second, providing a sufficient number of samples to accurately track a 3D object at 30fps. To demonstrate the utility of this procedure, we use this approach to track the head position and orientation of people for intelligent driver support systems and for human interaction analysis in intelligent meeting rooms.

1.1. Prior Work

Rapid performance gains and increases in programmable functionality have made GPUs a compelling platform for many computationally demanding tasks [2]. For computer

978-1-4244-2340-8/08/$25.00 ©2008 IEEE

vision, GPUs enable real-time performance for many computationally demanding applications or simply free up the main processor by offloading calculations to the video card that would otherwise be idle [3]. For example, a well designed GPU-based implementation of the 2D KLT tracker was shown to attain a 20 times improvement over the CPU [4]. Likewise for 2D particle filter tracking, previous work has demonstrated how stream processors can speed up the template matching step [5, 6]. For 3D processing and augmented reality, notable systems have been presented that use the GPU to track 2D points, build a 3D reconstruction of the environment, and then visually augment and display the result [7].

In contrast to these prior works, our tracking system operates from a fundamentally different perspective. Instead of thinking of the GPU as a processing farm and a visualization tool, we use the GPU's rendering abilities to generate virtual samples that themselves guide the tracking procedure. This top-down approach does not require 2D processing nor any correspondence-point matching between video frames. It is intrinsically targeted towards the GPU, since 3D rendering has always been the primary use for GPU. Still, our approach is enhanced by the addition of the programmable pipeline, which is able to evaluate each of the samples after they are rendered with the fixed-function pipeline.

2. Sampling Importance Resampling

We consider tracking rigid objects in a 3D world. At any instance in time they can be represented with respect to a fixed Cartesian coordinate system by their position, (x, y, z) and unit quaternion rotation, (q_t, q_x, q_y, q_z). The choice of the quaternion representation is important for the tracking, because unlike rotation matrices, axis-angle, or Euler angle representations, a weighted average of a set of quaternion rotations yields a single rotation that can be considered the centroid of the set.

The core of our tracking system uses particle filters to maintain track of each object. A particle filter is a Monte Carlo estimation method based on stochastic sampling [8, 9] that, regardless of state model, converges to the Bayesian optimal solution as the number of samples increases towards infinity. The concept is to choose an appropriate set of weights and state samples,

$$\{(c_t^{(l)}, \boldsymbol{y}_t^{(l)}) : l \in 0, \cdots, N-1\}, \tag{1}$$

such that the *a priori* expectation of the state $\boldsymbol{y}_t$ can be approximated from the weighted average [10],

$$\mathrm{E}\big[\boldsymbol{y}_t \mid \boldsymbol{z}_0, \cdots, \boldsymbol{z}_t\big] \approx \sum_{l=0}^{N-1} c_t^{(l)} \boldsymbol{y}_t^{(l)}. \tag{2}$$

Each state sample, $\boldsymbol{y}_t$, contains the position and rotation of the object as well as any parameters, (e.g., velocity or acceleration) used for the particular state model.

In a classical tracking problem, the object's state, $\boldsymbol{y}_t$, is observed at every time step but assumed to be noisy, hence the optimal track can be found by maximizing the posterior probability of the movement given the previous states and observations. In our vision-based tracking problem, instead of observing a noisy sample of the object's state, we observe an image of the object. The observation noise is negligible, but the difficulty lies in inferring the object's state from the image pixels. The solution to this problem can be estimated using an *appearance-based* particle filter with similar construction. We generate a set of state-space samples and use them to render virtual image samples using the fixed-function pipeline of the GPU. Each virtual image can be directly compared to the observed image, and these comparisons can be used to update the particle weights.

Given the existence of a set of N samples with known states, $\{\boldsymbol{y}_t^{(l)} : l \in 0, \cdots, N-1\}$, we can devise the observation vector,

$$\boldsymbol{z}_t = \begin{bmatrix} d(\boldsymbol{y}_t, \boldsymbol{y}_t^{(0)}) \\ \vdots \\ d(\boldsymbol{y}_t, \boldsymbol{y}_t^{(N-1)}) \end{bmatrix}, \tag{3}$$

where $d(\boldsymbol{y}, \boldsymbol{y}')$ is any image-based distance metric. As with a classical SIR application, it is a requirement to maintain and update a set of samples with a known state at every time step. These samples will update the observation vector, and they are conditionally independent of all previous states and observations given the current state.

For a Markovian system that is perturbed by stochastic noise, a Sampling Importance Resampling (SIR) particle filter offers a practical approach and avoids degeneracy by resampling [8]. For a tractable solution to many nonlinear motion models, the samples can be drawn from the state transition density, (4), as long as weights are set proportional to the posterior density of the observation given the samples. This requires a motion model that can be modeled as a first order Markov process, meaning that object state $\boldsymbol{y}_t$ at time t is conditionally independent of its past states given its previous state,

$$p(\boldsymbol{y}_t \mid \boldsymbol{y}_0, \cdots, \boldsymbol{y}_{t-1}) = p(\boldsymbol{y}_t \mid \boldsymbol{y}_{t-1}). \tag{4}$$

A full iteration of the SIR filter is as follows:

1. Update samples:

$$\boldsymbol{y}_t^{(l)} \sim p(\boldsymbol{y}_t \mid \overline{\boldsymbol{y}}_{t-1}^{(l)})$$

2. Calculate weights:

$$c_t^{(l)} = \frac{p(\boldsymbol{z}_t \mid \boldsymbol{y}_t^{(l)})}{\sum_{l=0}^{N-1} p(\boldsymbol{z}_t \mid \boldsymbol{y}_t^{(l)})}$$

3. Estimate current state:

$$\widehat{\boldsymbol{y}}_t = \sum_{l=0}^{N-1} c_t^{(l)} \boldsymbol{y}_t^{(l)}$$

4. Resample:

$$\overline{\boldsymbol{y}}_t^{(l)} \sim \rho(\overline{\boldsymbol{y}}_t \mid c_t^{(0:N-1)}, \boldsymbol{y}_t^{(0:N-1)})$$

The choice of the state model is specific to the type and motion of the objects in question. A basic example could be a constant-velocity model, where $\boldsymbol{y}_t$ consists of a position, rotation, velocity, and angular-velocity, and the motion is updated with linear dynamics and a perturbation of the velocities with an unbiased random process. For other applications, such as the head tracking described in Section 4, a nonlinear motion model can provide a more appropriate construction.

3. Tracking in Augmented Reality

Our approach uses the appearance-based particle filters to track objects in an *augmented reality (AR)*, a virtual environment that mimics the view-space of one or more real cameras.

Each tracked object is initialized by placing a predefined 3D shape model into the virtual world based on an estimate of the object's position and orientation. The choices of model shape and initialization are beyond the scope of this paper, but many techniques are possible. For instance, stereo-disparity or a time-of-flight camera can supply accurate 3D initialization, or the system can simply be initialized with some approximate 3D shape (plane, cube, ellipsoid, etc.). Since the tracking procedure uses the projected appearance of the texture, the approach can tolerate some inaccuracy in the 3D shape model. The system completes the initialization by uploading the most recent video frame into a GPU texture object, and computing texture coordinates for each of the vertices in the shape model. Since the camera image is identical to the viewport, each texture coordinate is computed using the same transformation (i.e. ModelView and Projection matrices) as rendering pipeline.

The SIR particle filter generates a set of state samples, each corresponding to a slight rotation and translation of the object. For each sample, the texture-mapped model is rotated, translated, then rendered in perspective projection. The result resembles small perturbations of the object set against an empty background. To calculate the sample weights, the rendered image is compared to the subsequent camera frame. We perform this comparison using a mean normalized cross-correlation (MNCC) metric, which is performed on the GPU as well.

An overview of this process is presented in Fig. 2. In the remainder of this section we describe these steps in more detail.

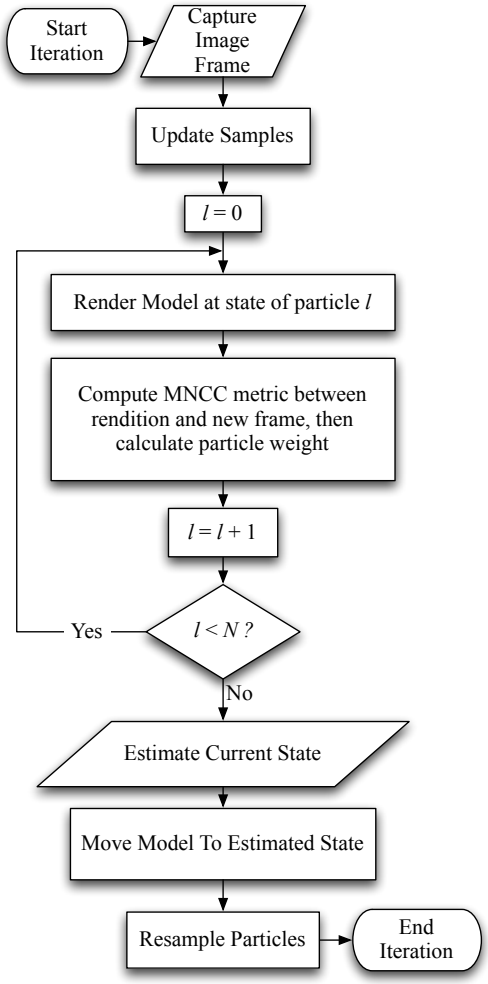

Figure 2. An overview of the tracking algorithm. The diagram contains the processing steps for each video frame.

3.1. Camera Perspective

As a prerequisite to tracking, a virtual projective view-space must be created that mimics the field-of-view of the camera. This requires knowledge of the camera's intrinsic parameters and distortion coefficients, and in the case of multi-camera tracking, the extrinsic parameters that represent the location of the camera relative to a global coordinate system. We refer the reader to [11] for more information on camera calibration. Many software packages are available to estimate and remove the distortion as well as estimate the intrinsic camera parameters that specify a linear projection from world coordinates into camera coordinates.

In our system, the lens distortion is removed from each camera frame, and each undistorted video frame can be modeled as the image from a pinhole camera. The camera

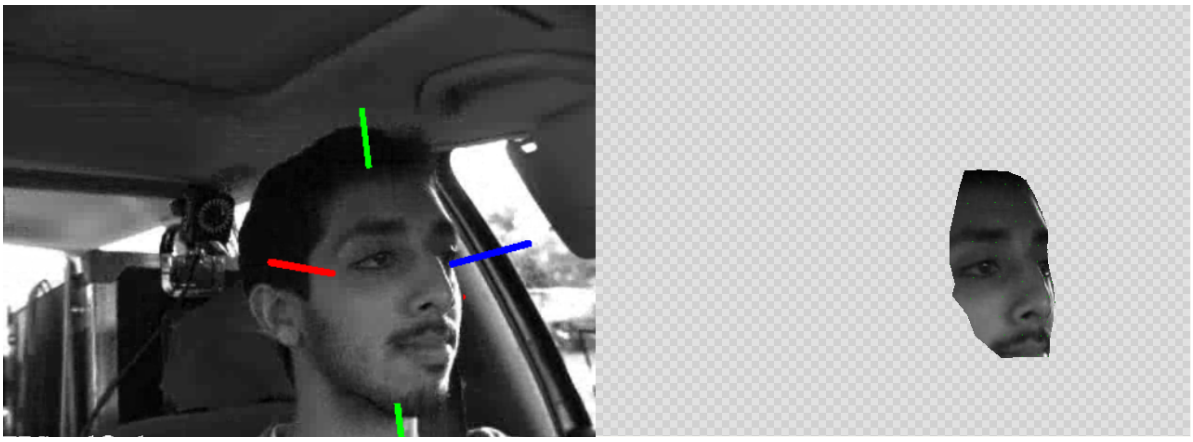

Figure 1. (left) The video frame with the overlaid model axis. (right) The model as rendered by the tracking system with a checkered background for clarity

is parameterized by an intrinsic matrix,

$$\boldsymbol{A} = \begin{bmatrix} f_u & 0 & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix}, \tag{5}$$

which defines the linear transformation between the 3D coordinates of a point in the world to the homogeneous coordinates of a 2D point on the camera image. In this matrix, f_u and f_v are the focal lengths in pixel units, and c_u and c_v specify the principal point and origin of the camera coordinate system, typically near the center of the image. Using a right-handed coordinate system in the focal plane, the x-axis points to the right, the y-axis points downwards, and the z-axis points forward.

The viewing region of a camera can be conceptualized as a rectangular pyramid that extends forward from the camera origin out to infinity. In contrast, most 3D graphics APIs consider the view space to be a *normalized view volume*. This is the interior volume of the viewing pyramid sliced by two planes perpendicular to the focal plane. These planes are known as the near and far clipping planes, specified by their z-axis coordinates, k_n and k_f. To imitate a camera view in a normalized view volume, the transformation matrices in the 3D graphics API must be set such that a projection from world coordinates to viewport coordinates mimics the transformation from world coordinates to the camera's image plane, for all points that lie within the clipping planes. This can be accomplished by adjusting two 4x4 transformation matrices: the *ModelView* matrix that accounts for the position and direction of the camera relative to the model, and the *Projection* matrix that provides the perspective projection. Given the dimensions of the camera image, (w, h), the Projection matrix can be represented in terms of the intrinsic parameters,

$$\boldsymbol{P} = \begin{bmatrix} \frac{2f_u}{w} & 0 & 1 - \frac{2c_u}{w} & 0 \\ 0 & \frac{2f_v}{h} & -1 + \frac{2c_v + 2}{h} & 0 \\ 0 & 0 & \frac{k_f + k_n}{k_n - k_f} & \frac{2k_f k_n}{k_n - k_f} \\ 0 & 0 & -1 & 0 \end{bmatrix}. \tag{6}$$

This equation assumes the camera image has an upper-left origin, and the 3D graphics display has an origin in the lower-left corner of the viewport. To maintain a right-handed coordinate system (OpenGL convention) and point along the positive z-axis, the virtual camera must be rotated a half turn about the x-axis. This requires initializing the ModelView matrix to the following:

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

When using multiple cameras, the position and orientation of the cameras must be known relative to each other. The extrinsic parameters for camera i define a 4x4 matrix, $\boldsymbol{E}_i$, that transforms a point in the global coordinate system to the local coordinate system of that camera. To model this transformation, the ModelView $\boldsymbol{M}_i$ must be multiplied by the extrinsic matrix,

$$\boldsymbol{M}_i = \boldsymbol{M} * \boldsymbol{E}_i. \tag{8}$$

3.2. Computing Sample Weights

As a potential image-comparison metric, normalized cross-correlation (NCC) provides an appealing approach for comparing two image patches, having the desirable property that it is invariant to affine changes in pixel intensity in either patch. Given two image patches specified as M-dimensional vectors of intensity, $\boldsymbol{\phi}$ and $\boldsymbol{\phi}'$, we can specify a NCC-based distance metric as follows:

$$d_{NCC}(\boldsymbol{\phi}, \boldsymbol{\phi}') = 1 - \frac{1}{\sqrt{\sigma_{\phi}^2 \sigma_{\phi'}^2}} \sum_{i=0}^{M-1} (\phi_i - \mu_{\phi})(\phi_i' - \mu_{\phi'}), \tag{9}$$

where μ_{ϕ} is the mean intensity,

$$\mu_{\phi} = \frac{1}{M} \sum_{i=0}^{M-1} \phi_i, \tag{10}$$

and σ_{ϕ}^2 is the variance of the intensity,

$$\sigma_{\phi}^2 = \frac{1}{M} \left(\sum_{i=0}^{M-1} \phi_i^2 \right) - \mu_{\phi}^2. \tag{11}$$

The unit constant and minus sign are introduced in (9) provide a positive distance measure in the range $[0, 2]$.

When the lighting variation is non-affine (e.g., specular reflections, shadowing, etc.), NCC performs poorly as a global image metric, since the transformation cannot be modeled by a global DC-offset and scaling. If the image patches are small enough, however, it is likely that they will be *locally* affine. As a consequence, better invariance to globally non-uniform lighting can be gained by using the average of a series of P small image patch NCC comparisons spread out over the object of interest. This is the basis of the mean normalized cross-correlation (MNCC) metric that we use in our tracking system,

$$d(\boldsymbol{y}, \boldsymbol{y}') = \frac{1}{P} \sum_{p=0}^{P-1} d_{NCC}(\boldsymbol{\phi}_p, \boldsymbol{\phi}_p'). \tag{12}$$

Once we compute the MNCC score with the GPU, we can directly relate these comparisons to the conditional observation probability if we can model the distribution such that it depends only on the current sample,

$$p(\boldsymbol{z}_t \mid \boldsymbol{y}_t^{(l)}) \propto h(z_{t,l}, \boldsymbol{y}_t^{(l)}), \tag{13}$$

where $z_{t,l}$ is the lth component of $\boldsymbol{z}_t$, and $h(\cdot, \cdot)$ is any valid distribution function.

Through experimentation, we found accurate tracking performance by modeling the observation probability as a truncated Gaussian envelope windowed by the displacement between the current sample state and the sample with the smallest MNCC distance. Denote this latter sample as

$$\boldsymbol{y}_t^{(*)} = \left\{ \boldsymbol{y}^{(l)} : l = \underset{l}{\operatorname{argmax}}\ z_{t,l} \right\}, \tag{14}$$

and define the state displacement as

$$s(\boldsymbol{y}, \boldsymbol{y}') = d_P(\boldsymbol{y}, \boldsymbol{y}') + \alpha\, d_A(\boldsymbol{y}, \boldsymbol{y}'), \tag{15}$$

where $d_P(\cdot, \cdot)$ is the Euclidean distance between the position of the samples, α is a parameter that scales the relative contribution of each measure. and $d_A(\cdot, \cdot)$ is the angular displacement computed as the scalar angle between the two quaternion rotations, $\boldsymbol{q}$ and $\boldsymbol{q}'$.

$$d_A(\boldsymbol{y}, \boldsymbol{y}') = \left| 2 \cos^{-1} (q_t q_t') \right| \tag{16}$$

From these definitions, we formally define our distribution model as

$$h_t(z, \boldsymbol{y}) = \begin{cases} 0 & T_z < z \\ 0 & T_s < s(\boldsymbol{y}, \boldsymbol{y}_t^{(*)}) \\ \exp\left(-\frac{1}{2\sigma^2} z^2\right) & \text{otherwise} \end{cases}, \tag{17}$$

where T_z and T_s are scalar thresholds and σ is the standard deviation of the envelope.

3.3. OpenGL Implementation

After each new video frame is captured, it is copied into a texture object on the GPU. Then, for each sample, the objects are rendered to an off-screen framebuffer and compared to the new video texture to calculate the sample weight. We first discuss how this is accomplished for a single object and a single camera.

In the current OpenGL specification, a programmable shader cannot read data from the framebuffer. To overcome this limitation, the Frame Buffer Object (FBO) extension provides the ability to render directly to a texture, which can be sampled in a shader. Multiple textures can be attached to a FBO, and the target textures can be switched by a call to the glDrawBuffer and glDrawBuffers methods. Rather than switching FBO attachments, it is possible to switch between a different FBO for each processing pass, but we have implemented both approaches and noticed a substantial performance gain by switching between attachments rather than between FBOs.

Our method requires two passes to compute the weight for each sample. In the first pass, a 8-bit monochrome rendering texture and 16-bit depth texture are attached to the FBO. Then, the textured 3D model is rotated and translated as described by the sample state, and rendered to these textures using the fixed-function GPU pipeline, (i.e., the standard procedure for rendering an object). Internally, this results in the following steps. Each vertex in the 3D model is multiplied by the ModelView matrix to translate and rotate

Figure 3. (top) Video from four cameras with four tracked heads. The location of the tracks are indicated by the overlaid axes. (middle) The models as rendered by the tracking system with a checkered background for clarity. (bottom) A virtual reconstruction of the scene.

the point to account for the position of the head in relation to the camera. Afterward, each vertex is multiplied by the Projection matrix to convert the 3D coordinates into 2D viewport coordinates with an associated z-value describing the relative depth of the original vertex. At this point the GPU rasterizes the polygons to compute *fragments* – pixels with z-values that correspond to viewport coordinates that are within the boundary of the transformed polygon. Each fragment is assigned a color value (luminance in our case) by interpolating the value from the texture coordinates assigned to each of the neighboring vertices. The final rendered image is computed by applying a depth test to each fragment, discarding the fragment if its relative depth is further from the camera than another fragment at the same viewport location.

In the second pass, our approach computes the mean normalized cross-correlation between the rendered image and the next image frame, using the programmable pipeline of the GPU. We detach the rendering and depth textures from the FBO and attach a 32-bit floating point texture to store the correlation result[1].

Then, the object is rendered again with a few notable exceptions. First, texture-mapping is disabled, causing the GPU to run in a mode where each vertex is assigned an RGB color. Next, the depth test that discards overlapping fragments is disabled and replaced by a blending function that sums the RGB values of each overlapping fragment. To produce the actual calculations, the fixed-function vertex pipeline is replaced with an OpenGL Shading Language (GLSL) vertex shader [12].

A vertex shader is a program that operates on each vertex as it passes through the GPU pipeline. Vertex shaders have the ability to change the position, texture coordinates, and color of the vertex, and they can sample from textures and perform a variety of computational tasks. Modern GPUs contain as many as 256 shaders capable of parallel operation. In our application, each vertex shader calculates an NCC score between image patches in the new image texture (the next camera frame) and the rendered image texture. The shader first calculates the 2D projection of the 3D vertex position. Then it performs the NCC of a 9×9 pixel image patch[2] centered on this location in both textures. Computationally, this requires looping through the pixels of each patch twice – first to establish the mean and standard deviation of each patch, and then to calculate the NCC as described in (9). A vertex shader is limited to standard graphical outputs, but we can overcome this restriction by coding the NCC into the RGB color output of the vertex,

$$RGB_v = \begin{bmatrix} d_{NCC}(\boldsymbol{\phi}_v, \boldsymbol{\phi}'_v) \\ 1.0 \\ 0.0 \end{bmatrix}. \qquad (18)$$

By setting the G component equal to 1.0 and ensuring 32-bit floating-point math with glClampColor, a summation of the output from multiple shaders will effectively count of the number of times that the procedure is performed. Any pixels in the correlation window that lie outside the rendered

[1]The ability to use mixed formats in a framebuffer object is an experimental OpenGL extension, and where not supported, a 32-bit floating point buffer can be used for all three attachments.

[2]We have experimented with different patch sizes and have not seen clear evidence of an optimum size.

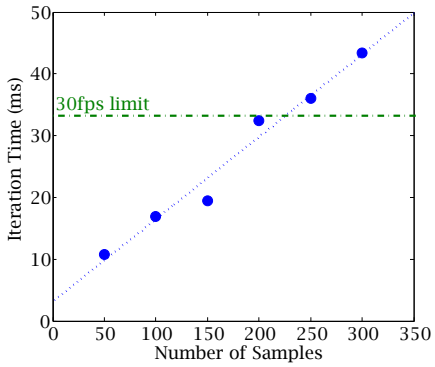

Figure 4. The mean time for each iteration of the tracking system as a function of the number of samples in the particle filter. These results corresponds to single 640x480 camera and a 3D anthropometric facial model made up of 142 triangles.

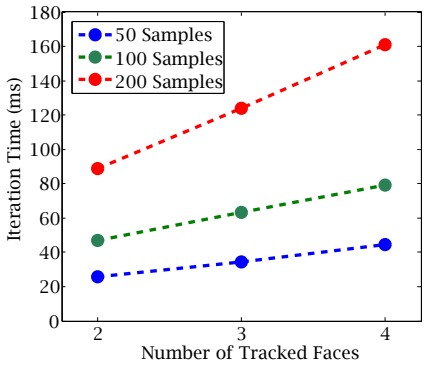

Figure 5. The mean time required for each tracking iteration as a function of the number of tracked objects. These results correspond to four 640x480 cameras with 3D anthropometric facial models made up of 142 triangles apiece.

model are skipped during the NCC calculation, and if fewer than half of the pixels remain, the calculation is terminated and the color is set to $RGB_v = [0.0, 0.0, 0.0]^T$. Additionally, we use the depth texture to perform a manual depth test and similarly discard vertices that are occluded. For the last step in the shader program, the output position of the vertex is set to the lth framebuffer pixel, where l is the particle number. This will make every fragment in the particle overlap, and because of the blending function, this will compute the summation of each NCC result.

This two-pass GPU process is repeated for every particle sample (for our experiments we generate 200 samples[3]), and afterward, the MNCC filtering observation vector, $\boldsymbol{z}_t$, can be derived by decoding the result from each framebuffer pixel, $\boldsymbol{\psi}_t(l)$. This is accomplished by dividing the NCC summation stored in the pixel's R component by the count stored in the G component,

$$\boldsymbol{z}_t = \begin{bmatrix} d(\boldsymbol{y}_t, \boldsymbol{y}_t^{(0)}) \\ \vdots \\ d(\boldsymbol{y}_t, \boldsymbol{y}_t^{(N-1)}) \end{bmatrix} = \begin{bmatrix} \frac{\psi_{t,R}(0)}{\psi_{t,G}(0)} \\ \vdots \\ \frac{\psi_{t,R}(N-1)}{\psi_{t,G}(N-1)} \end{bmatrix}. \quad (19)$$

Once the weights are calculated, the model is set to the estimated state and this two-pass process is repeated one more time to compute a confidence score from the MNCC metric.

To ensure that these calculations run as fast as possible, it is important to minimize the data that must be passed between processor memory and graphics memory, and minimize the total number of graphics calls. Only one image upload (glTexSubImage2D) is necessary to copy the new image frame, one download (glReadPixels) to get the sample weights, and one download (glReadPixels) to obtain the confidence score. Our code makes heavy use of display lists, allowing us to cache all of the vertices and drawing calls that are required to render the model, as well as the state changes between the two fixed-function and programmable-function steps.

[3]We generated a sample set as large as was possible with our hardware for continuous 30 fps performance. From our informal experiments, tracking can be performed with as few as 25 samples with the downside that smaller sample sizes are increasingly prone to jitter and loss of track.

3.4. Multiple Objects and Multiple Views

This approach easily allows for multiple objects and multiple views. For the multi-object scenario, a separate particle filter is maintained for each object, but the number of samples is the same for all of the particle filters. During the rendering step, all of the objects are rendered to the FBO. Since the depth test is enabled, there is no ambiguity for occluded objects – they are simply not visible where the lie beyond another object. For the correlation step, the objects are rendered sequentially, and a uniform variable is set telling the shader the current object index and the total number of objects being tracked. The shader codes the MNCC score into the output pixels such that a different pixel corresponds to each sample of each object. Once this is decoded with the CPU, it is it used to update the particle filters for each object.

For multiple views, (i.e. multiple cameras), we multiplex the camera images into one large tiled image, and upload this to the tracker every frame. The system sets the appropriate transformation matrices for each camera and renders the objects once for every view, using the glViewport command to draw each view in a separate tile. For the correlation step, the viewport is set to span the whole tiled region, and a 2D uniform variable is passed to the shaders to indicate the offset of each tile. The shaders code the results into each pixel such that the MNCC is combined for each sample and object in every view. For each object, the result can be decoded similarly to the single-camera case using (19).

4. Experiments and Evaluation

We have employed this approach for automatic tracking of human heads in natural environments. We use a generic anthropometric head model, bootstrapped by face detection and head pose estimation modules [13]. Fig. 1 shows an example of monocular tracking of a driver's head in real-world driving conditions. Fig. 3 shows an example of head tracking in meeting room scenarios, where multiple people interact while viewed from multiple camera.

In the automotive scenario, knowledge of the driver's head provide critical information that can be used to design driver assistance systems. With this in mind, real-time performance is essential. The camera used for these experiments captures at 30fps, and it is desirable to operate the tracker at the same rate, which is possible with our GPU-based solution. Fig. 4 plots the time for one iteration of the system, given a varying number of particle samples. On our current hardware, approximately 200 samples per frame (6000 samples per second) is possible with this approach.

In Fig. 4 we plot the iteration time for the multi-view, multi-object case in a meeting room scenario. Here it can be seen that the addition of multiple-objects has far less impact on computational complexity than an increase in the sample number. This can be attributed to the extra overhead for each sample from setting and unsetting of GPU attributes and shader variables.

We refer readers interested in the accuracy of the tracker to a recent performance evaluation of head tracking in challenging automotive environments [14].

5. Concluding Remarks

Vision-based 3D tracking helps to bridge the gap between computers and their environments. In this paper, we introduced a novel, real-time approach to multiple-object 3D tracking with the GPU. Our approach uses standards-based OpenGL and the OpenGL Shading Language to follow each object in an augmented reality that mimics the view-space of one or more cameras.

Further extensions to this system could focus on model augmentation, since the initial model represents only the slice of the object that was visible from the perspective of one camera when the model was created. As a possible solution, the model can be augmented over time by adding additional sets of polygons and textures, or alternatively initialized from the views of multiple cameras. Care should be taken to prevent adding polygons that can overlap the existing model and false edges.

Acknowledgments

We thank the University of California Discovery program and the Volkswagen Electronics Research Laboratory for their support of this research. In addition, we thank the our team members in the Computer Vision and Robotics Research Laboratory for their continuing support and assistance.

References

[1] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Version 2 (5th Edition)*. Addison-Wesley Professional, 2005.

[2] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] J. Fung and S. Mann, "OpenVIDIA: parallel GPU computer vision," in *Proc. Int'l. Multimedia Conf.*, 2005, pp. 849–852.

[4] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics," *To Appear: Machine Vision and Applications*, 2008.

[5] A. Montemayor, J. Pantrigo, A. Sánchez, and F. Fernández, "Particle filter on GPUs for real-time tracking," in *Proc. Int'l. Conf. Computer Graphics and Interactive Technologies*, 2004, p. 94.

[6] J. Ohmer, F. Maire, and R. Brown, "Real-time tracking with non-rigid templates using the GPU," in *Proc. Int'l. Conf. Computer Graphics, Imaging, and Visualisation*, 2006, pp. 200–206.

[7] M. Pollefeys, D. Nistér, J.-M. Frahm, A. Akbarzadeh, P. Mordohai, B. Clipp, C. Engels, D. Gallup, S.-J. Kim, P. Merrell, C. Salmi, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewénius, R. Yang, G.Welch, and H. Towles, "Detailed real-time urban 3D reconstruction from video," *Int'l. J. Computer Vision*, vol. 78, no. 2–3, pp. 143–167, 2008.

[8] S. Haykin, *Adaptive Filter Theory*, 4th ed. Prentice-Hall, Inc., 2002.

[9] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking," vol. 50, no. 2, pp. 174–188, 2002.

[10] N. G. Arnaud Doucet, Nando de Freitas, *Sequential Monte Carlo Methods in Practice*. Springer-Verlag New York, Inc., 2001.

[11] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2004.

[12] J. Kessenich, D. Baldwin, and R. Rost, *The OpenGL Shading Language*. 3DLabs, Inc. Ltd., 2006, language Version 1.2.

[13] E. Murphy-Chutorian and M. Trivedi, "Head pose estimation for driver assistance systems: A robust algorithm and experimental evaluation," in *Proc. IEEE Conf. Intelligent Transportation Systems*, 2007, pp. 709–714.

[14] ——, "HyHOPE: Hybrid head orientation and position estimation for vision-based driver head tracking," in *Proc. IEEE Intelligent Vehicles Symposium*, 2008.