# Chapter 11
# Table-Mounted System Calibration

From a user's point of view, calibration of modern eye trackers has improved considerably. The most noticeable improvement is the absence of a chin rest. By allowing constrained head movement (usually within a finite volume), modern eye trackers, such as the Tobii, have rendered the chin rest unnecessary (at the expense of some gaze accuracy, compared with, for example, dual-Purkinje eye trackers that require bite bars). However, even though calibration has been greatly simplified (a most welcome technological advancement to be sure), it is still susceptible to the same problems that plagued older technology (i.e. interference from eyelashes, certain spectacle rims, etc.). This is of course due simply to interference in the camera's image of the eye(s), just as electromagnetic head trackers suffer from interference from nearby metallic surfaces. Because both older and modern table-mounted trackers rely on video cameras to image the user's eye(s), it is still important that users sit at an appropriate distance and level from the camera optics. In some ways, the complexity of calibration has been shifted from the user to the application developer.

Ensuring that users sit at an appropriate position relative to the imaging optics is now mainly a matter of proper visual feedback/notification. For example, Tobii includes a track status window that displays the location of the user's eyes from the camera's point of view. Figure 11.1 shows three instances of real-time feedback: a user sitting nearly optimally in front of the camera at an appropriate level, sitting slightly too far (with the head dropping in elevation as a result) with the head tilted, and with one eye closed. Providing this type of feedback is the responsibility of the programmer although it is not difficult. The most difficult aspect may be when displaying such feedback is appropriate. Should it be provided continuously, even during diagnostic eye movement capture? This is clearly a design decision and depends on the nature of the application being developed. For applications where eye tracking feedback may be a distraction, it seems natural to restrict gaze point and camera eye position display to the calibration sequence, because in this case some form of visible targeting is required anyway.

A.T. Duchowski, *Eye Tracking Methodology*, DOI 10.1007/978-3-319-57883-5_11

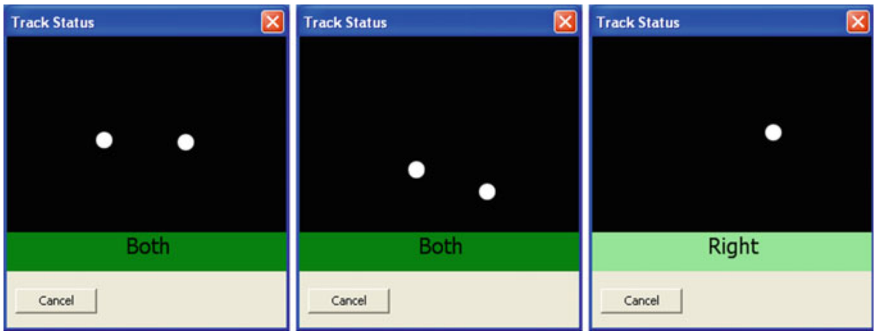

**Fig. 11.1** Tobii eye tracking status window: sitting level; too far back with head tilted; left eye closed

Another tradeoff in calibration complexity appears in the form of eye tracker configuration (more or less). That is, a secondary human operator is no longer needed to manipulate camera controls such as focus, aperture, etc. (at least with trackers such as Tobii's). This is also a welcome advancement, however, calibration from a programmer's perspective is complicated slightly due to the concurrency of the eye tracking (ET) client and event-driven GUI paradigms. That is, because the application now drives the eye tracker and not vice versa (this is highly preferable to the contrary operation of older systems), the application must synchronize both ET and GUI program components. Thus, operational control is gained at the expense of programming complexity.

Synchronization of ET and GUI components can be fairly easily accomplished via mutual exclusion in the program's critical sections. This is a common approach in concurrent programming. Whenever concurrent program components (e.g. threads) write to shared memory, exclusive access can be guaranteed by the use of mutexes (or semaphores). The code listings in Chap. 10 made frequent use of the lone `state→mutex` to access the shared `state→status` bitfield. However, details of calibration synchronization were purposefully postponed to the following sections.

Note that calibration synchronization as described below may perhaps be somewhat overly complicated. Instead, the use of a few more additional bitfields may be sufficient. In this chapter, calibration synchronization is accomplished via conditional waits.

## 11.1 Software Implementation

The design for calibration synchronization with conditional variables was based on Butenhof (1997) client/server programming example. In Butenhof's client/server system, a client requests that a server perform an operation asynchronously on a data element, while the client either waits for the server or proceeds in parallel and checks for the result of the operation at a later time. Note that it would be simplest to force the client to wait for the server by, for example, blocking on a mutex.

The intuitive blocking strategy blocks the ET thread while the GUI thread draws the calibration dot. With drawing completed and with the ET thread unblocked, the ET thread notifies the eye tracker to sample at the calibration location. The trouble with this simple approach is that calibration sampling occurs after the dot has been drawn. If the delay between drawing completion and calibration sampling is excessive, the user, out of boredom or anticipation for the next dot, may look away from the calibration location and thus spoil the calibration. The obvious criterion during calibration is ensuring that the user look at the calibration dot when the eye tracker is sampling at that same location. This may be accomplished by issuing a verbal instruction to this effect to the user prior to calibration. Or, as is suggested here, use calibration dot animation as a means of orienting the user's attention. That is, animate the dot and instruct the eye tracker to sample at the same time.

Adopting Butenhof's client/server model, with the ET thread taking on the role of the client and the GUI thread taking the role of server, the idea is for the ET thread to request the GUI thread to animate a calibration dot, while the ET thread signals the eye tracker to sample at the same calibration coordinates. Thus, ideally, the calibration dot is being drawn at the same time the eye tracker is calibrating.

The concurrent approach to drawing and calibration sampling is depicted in Fig. 11.2. The ET thread sends its request to the eye tracker to sample at the given calibration coordinates immediately after the ET thread signals the GUI thread to start drawing the calibration dot. The GUI thread animates the dot by, for example, pulsating the dot's size to attract and hold attention.

The concurrency between the ET and GUI threads should be considered carefully. It cannot be assumed that animating the calibration dot is either slower or faster than

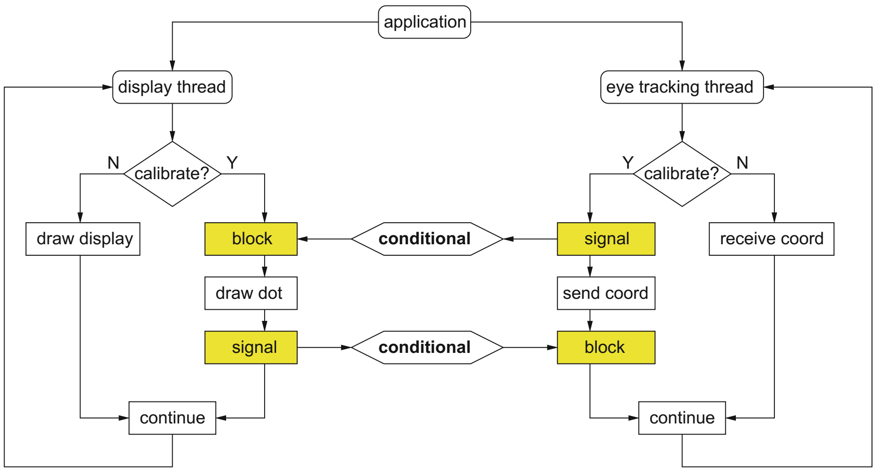

**Fig. 11.2** Tobii concurrent process layout

calibration sampling. It should be clear, however, that once each task is completed by its respective thread, that thread would continue with its next statement. Without a subsequent resynchronization following completion of both tasks, something of a race condition would then ensue. For example, if the GUI thread was faster in drawing the calibration dot than the eye tracker's calibration sampling, the GUI thread would continue to draw the next dot in the sequence. If the GUI thread was much faster than the ET thread, it is possible that it could complete drawing of all dots before the eye tracker had a chance to finish sampling at the very first coordinates. Conversely, if the ET thread was much faster, the eye tracker would be sampling at successive calibration coordinates although the user would still be looking at the very first calibration dot.

To resynchronize both threads after drawing/sampling of a particular calibration coordinate, the ET thread blocks after it has completed sampling at the calibration coordinates. Recall that the function call that is issued by the ET thread, namely `Tet_CalibAddPoint`, is blocking. Thus, we know the eye tracker has completed its calibration sampling when this statement completes. At this point the ET thread blocks itself and waits for the GUI thread to complete drawing the calibration dot. Listing 11.1 highlights the relevant signal and blocking calls issued by the ET thread. Only the highlighted synchronization code differs from Listing 10.14 given previously.

Consider again the GUI thread. In the preceding discussion, it is tacitly assumed that the GUI thread takes longer than the ET thread in calibrating at the given coordinates. What happens if the GUI thread completes drawing before the eye tracker completes sampling? After drawing, the GUI thread signals the ET thread to unblock and then carries on in the loop to the next calibration point. If the next coordinates in the sequence were available, the GUI thread would race ahead of the ET thread. To prevent this race condition, it is blocked before it can start drawing the next dot. In fact, Butenhof's client/server example was chosen because it not only provides a means of synchronization between the processes, it also allows communication of the calibration coordinates. The ET thread controls which calibration point is to be sampled next and passes that information to the GUI thread. If this information is unavailable, the GUI thread must block and wait. Listing 11.2 highlights the relevant blocking and signaling calls made by the GUI thread. The highlighted synchronization code is the only difference from Listing 10.11 given previously.

The synchronization code in Listings 11.1 and 11.2 uses a `daimon` object to control the interprocess calibration synchronization and communication. The *daimon* moniker was chosen in place of Butenhof's *server* to avoid confusion between the eye tracking server and to differentiate from the traditional (UNIX) system *daemon*. The `daimon`, with its C++ interface given in Listing 11.3, was modeled after Butenhof's server, which was designed to process data requests either synchronously or asynchronously. The `daimon` is only used synchronously by the ET thread, hence some of the original asynchronous functionality may be missing.

Data requests are received by the `daimon` as either read, write, or quit operations, along with calibration coordinates $(x, y)$, embodied in object `request`. In its original server instantiation, read requests would be used to read the data that the server had

```
tobii_calibrate()
{
  if(!state→status.calibstarted) {

    // clear (do this only once per calibration!)
    Tet_CalibClear();

    pthread_mutex_lock(&state→mutex);
    state→status.calibstarted = 1;
    pthread_mutex_unlock(&state→mutex);
  }

  if(state→status.calibstarted) {

    // signal gui to draw calib point
    daimon→signal_gui(REQ_WRITE,sync,x,y);

    // get eye tracker to start recording at this point
    Tet_CalibAddPoint(x,y,samples,gazeDataReceiver,NULL,0);

    // block for gui to finish drawing
    if(sync) daimon→block_et();

    // advance to next calibration point
    pthread_mutex_lock(&state→mutex);
    state→calpoint++;
    pthread_mutex_unlock(&state→mutex);

    if( last calibration point ) {

      // signal gui to exit calibration
      daimon→signal_gui(REQ_QUIT,false,-1.0,-1.0);

      pthread_mutex_lock(&state→mutex);
      state→status.calibstarted = 0;
      state→status.calibrating = 0;
      state→calpoint = 0;
      pthread_mutex_unlock(&state→mutex);

      // finished calibrating, tell eye tracker we're done
      Tet_CalibCalculateAndSet();

      // hint: should perform quality check here and save
      // calibration file if calibration good enough
    }
  }
}
```

**Listing 11.1** Tobii thread: calibrate with conditional waits

```
void main_loop(void)
{
  // clear screen
  ...

  // view transformation
  ...

  // object transformation
  ...

  if(state→status.connected) {
    if(state→status.calibrating &&
       state→status.calibstarted) {

      // block until tobii thread advances to new point
      if(daimon→block_gui(&x,&y)) {
        // draw calibration dot at (x, y) coordinates
      }

      // signal tobii thread to advance to new point
      daimon→signal_et();

    }
    else if(state→status.running) {
      // draw left gaze point
      // (normalized coordinates scaled to window size)
      draw_point(state→x_l * w, h-state→y_l * h);

      // draw right gaze point
      draw_point(state→x_r * w, h-state→y_r * h);

      // draw average gaze point
      x = (state→x_l+state→x_r)/2 * w;
      y = (state→y_l+state→y_r)/2 * h;
      draw_point(x,h - y);

      // draw current camera eye point
      // (left eye, right eye, average of both)
      ...
    }
  }
  glutSwapBuffers();
}
```

**Listing 11.2** GUI main loop with conditional waits

```
typedef enum { REQ_READ, REQ_WRITE, REQ_QUIT } operation_t;

class Request {
  public:
  Request(operation_t op, float ix, float iy) : \
          operation(op), x(ix), y(iy) { }

  // friends
  class Daimon;

  // function code
  operation_t   operation;
  float         x,y;
};

class Daimon {
  public:
  Daimon() : \
          synchronous(false), \
          done_flag(false), \
          request_flag(false)
          { pthread_mutex_init(&mutex,NULL);
            pthread_cond_init(&request,NULL);
            pthread_cond_init(&done,NULL);
          };

  void signal_gui(operation_t op,bool sync,float x,float y);
  bool block_gui(float *x,float *y);
  void signal_et(void);
  void block_et(void);

  private:
  deque<Request *> queue;
  bool             synchronous;   // true if synchronous
  bool             done_flag;     // predicate for wait
  bool             request_flag;  // predicate for wait
  pthread_mutex_t  mutex;
  pthread_cond_t   request;
  pthread_cond_t   done;          // wait for completion
};
```

**Listing 11.3** Calibration `daimon` interface

finished processing. In the present case, only the write and quit operations are used by the ET thread (technically, the GUI thread reads the $(x, y)$ coordinates from the `daimon`, but it does so implicitly only when the ET thread has written data). Requests are pushed onto a queue for retrieval by the GUI thread. The queue maintained by the `daimon` acts as a type of conduit for calibration coordinates. Using First-In-First-Out (FIFO) order ensures that coordinates are processed in the proper sequence.

Listing 11.4 lists the GUI blocking and signaling calls. Recall that the GUI thread blocks itself and is signaled by the ET thread. When blocking itself, the GUI thread waits on the conditional variable `request`. Conditional variables are usually used

```
void Daimon::signal_gui(operation_t operation, bool sync,
                        float x, float y)
{
        Request        *req;

  pthread_mutex_lock(&mutex);

  synchronous = sync;

  // add new request to queue
  queue.push_back(new Request(operation,x,y));
  request_flag = true;

  // tell daimon a request is available
  pthread_cond_signal(&request);

  pthread_mutex_unlock(&mutex);
}

bool Daimon::block_gui(float *x, float *y)
{
        Request*        req=NULL;
        bool            gotdata=false;

  pthread_mutex_lock(&mutex);

  // wait for data
  while(!request_flag) pthread_cond_wait(&request, &mutex);

  // possible we missed signal OR request_flag was true
  // check queue before attempting to process data
  if(request_flag && !queue.empty()) {
    // strip from queue
    req = queue.front(); queue.clear();
    switch(req→operation) {
      case REQ_QUIT: break;
      case REQ_READ: break;
      case REQ_WRITE:
        *x = req→x; *y = req→y; gotdata = true; break;
      default: break;
    }
    if(req) delete req; // prevent memory leak
  }
  request_flag = false;

  return(gotdata);
}
```

**Listing 11.4** Calibration daimon implementation: blocking and signaling the GUI thread

for communicating information about the state of shared data, e.g. signaling when a queue is no longer empty, as in the present instance. When a request is made by the ET thread, and a set of calibration coordinates is placed onto the shared queue, the GUI thread is signaled, waking it up to retrieve the coordinates and draw the calibration dot. There are two important subtleties within the `block_gui` routine:

1. The `mutex` is locked upon function entry, but is only released upon exit from the `signal_et` function. Both `block_gui` and `signal_et` calls are made by the GUI thread and should be read as one long function that is broken up by the GUI thread's act of drawing a calibration dot. The `mutex` is kept in its locked state across both functions so that the GUI thread can signal the ET thread via the `done` conditional variable.
2. The `request_flag` is the `request` conditional variable's predicate and is used in the condition variable wait loop. Waiting for a condition variable in a loop protects against multiprocessor races and spurious wakeups (see Butenhof 1997 for details). However, spurious wakeups (e.g. a missed signal) may still occur or the `request_flag` may be set inadvertently. If so, and without any further testing, the GUI thread could possibly attempt to read data from an empty queue. To prevent this, the `request_flag` is tested again as is the state of the queue prior to data retrieval.

Listing 11.5 lists the ET thread blocking and signaling calls. Recall that the ET thread blocks itself and is signaled by the GUI thread. Function `signal_et` should be considered as the tail end of `block_gui`; it is used by the GUI thread to signal (wake up) the waiting ET thread after finishing drawing of the calibration dot. Function `block_et` is what the ET thread calls to block itself. It is roughly symmetrical to `block_gui` in that a condition variable wait loop is used along with a predicate (`done_flag`) linked to the condition variable (`done`).

```
void Daimon::signal_et(void)
{
  if(synchronous) {
    done_flag = true;
    pthread_cond_signal(&done);
  }
  pthread_mutex_unlock(&mutex);
}

void Daimon::block_et(void)
{
  pthread_mutex_lock(&mutex);
  while(!done_flag) pthread_cond_wait(&done,&mutex);
  done_flag = false;
  pthread_mutex_unlock(&mutex);
}
```

**Listing 11.5** Calibration `daimon` implementation: blocking and signaling the ET thread

## 11.2 Summary and Further Reading

This chapter focused on a subtle calibration problem: synchronization of drawing and tracker sampling at successive calibration coordinates in the calibration sequence. The main approach advocated here is to use concurrent process synchronization, namely conditional waits. The use of standardized POSIX threads is suggested, with Butenhof (1997) acting as a suitable text for learning various intricacies of thread programming. Other concurrent programming texts may also be suitable.

Of course, it must again be stated that the simpler form of concurrency via bitfields or flag variables is probably sufficient to achieve good calibration, particularly if proper verbal instructions are given to users.

Beyond the synchronization issue, most of the points of calibration given in Chap. 8 still apply to the table-mounted system. Specifically, it is still important to choose good calibration coordinates and to check calibration drift and error.

Calibration coordinates should be selected to cover the extents of the viewing area. The use of five calibration points at the corners and center is a good strategy, although tracker manufacturers are beginning to suggest the use of as few as two points at the corners. As technology and computer vision algorithms improve, calibration may become obsolete altogether and make way for autocalibrating eye trackers. However, this advancement is still a few years away. For the time being, if accuracy is important to the given eye tracking application, drift and error should still be checked (and reported in eventual results reports or papers).

Calibration error checking can be performed via a strategy similar to the one given in Sect. 8.2.1. This consists of measuring the average distance (error) between the known calibration point coordinate and the average distance of the multiple gaze samples obtained by the eye tracker during calibration sampling. Modern eye trackers, such as the Tobii, facilitate this during calibration by providing validity data obtained during calibration. For example, Tobii sends the **`STet_CalibAnalyzeData`** data structure to the eye tracking application that contains information suitable for calibration error checking. This can be used to measure the error at specific calibration coordinates or provide visual feedback display regarding the “goodness of fit” of the calibration. These indicators can then be used to either recalibrate specific coordinates or to quantify the calibration.