Lecture IX
THINWIRE VISUALIZATION ARCHITECTURE

§1. Introduction

- We use the client-server model of computation. The server holds one or more very large images that can be viewed by several clients simultaneously. These images may range from 100 MB to terabyte size.
- The client has general computing and display capability, such as in a modern workstation. The server only needs general computing capability.
- The **fundamental assumption** is that the main bottleneck is the bandwith wire connecting the server and client, hereafter known as the **thinwire**.
- One consequence of the thinwire assumption, when necessary, we assume the server and client may be assumed to be infinitely powerful. This extreme arises when we have no power to improve the bandwidth of the network, but can afford to purchase supercomputing power at both ends of the wire.
- In practice, the graphics display at the client may be a bottleneck as well. We try to balance these two requirements.
- Our main goal is to design a client that is extensible, instrumented and tunable.
- In the following implementation, the graphical interface is based on the OpenGL graphics API and the network protocol is based on TCP/IP. Multithreading will be employed.
- References include [2] and [1].

§2. General Issues

- The key technique of our approach is the use of **multi-foveated images**, that is images in which the resolution varies over the extent of the image.
- Such images are ideally suited for thinwire settings because of their flexibility in supporting reduced bandwidth images.
- The flexibility is also enhanced by certain psycho-physiological factors in visualization – most striking of which is the fact that primate vision is foveated.
- The support of multi-foveated images will require completely different techniques, depending on whether we consider raster images or vector data. We shall be interested in both kinds of data. The former will be based on wavelet techniques; the latter on generalizations of LOD hierarchies.
- The second principle of our design is the notion of **responsiveness**. This is defined as the ability of a system to respond naturally and usefully to user requests in the face of (1) bandwidth that is unreliable (possibly dropping to zero temporarily), and (2) unpredictable changes in user visualization objectives.
- The internet is an example of (1). In contrast, there is a large literature treating I/O or secondary memory bottlenecks. Two difference from our situation is that (i) I/O bandwidths are actually quite reliable, and (ii) I/O bandwidths may be one or two orders of magnitude more than the kinds of bandwidths we assume. So the techniques used for the I/O bottlenecks may no longer work.

- The basic idea we exploit in designing responsiveness is the concept of **amortization**. We rely on the fact that the **average bandwidth** has reasonable minimum value. In times of plenty, we store up some extra information which can be released in times of poor bandwidth. Hence, we must design data structures to accomodate such "extra information" that we accumulate over and on top of the explicit user requests. What makes this possible is our ability to finely match the transmitted data to the visual requirements of the user.
- An example of unpredictable changes in visualization objectives, we imagine a user that starts out trying to visualize a part of a large image, but quick changes mind and jumps to another part of the image.
- Thus, the kind of "predictability" used in walk-thru or fly-thru navigation cannot be exploited here. More generally, we are focused on "eye action" (such as foveating or saccade) which are much more rapid than the "body motion" in walk-throughs.
- The third theme in our system is **instrumentation**. How do performance of our client scale with size of image? How to tune the client program? How to measure/instrument the client program? How to have effective visualization on REALLY thin wires?
- It is interesting to see that instrumentation in the context of visualization adds a new dimension to networking considerations. For instance, for purely network considerations, it is reasonable to ask a question such as "what is the best way to send 1 terabyte through this thinwire?". Presumably we can empirically or theoretically model the system and choose the right network parameters. But for visualization, sheer bandwidth considerations is not enough. Rather, we need to ask for the ability to respond to user's changing requirements.

§3. Overview of System

- We begin to discuss the design of a responsive visualization client-server architecture. The design is based on multithreading.
- The Server holds one or more large images or maps that users may wish to visualize, using a Client Program. The Server can service several Client Programs simultaneously. The Client Interface supports four basic visualziation commands: Pan, Zoom, Foveate and Jump. The first two is standard. Foveate means to increase the level of resolution about any chosen point. Jump means that we want to view an arbitrary new position of the image.
- The Client Program uses 4 threads: **Input**, **Manager**, **Data**, **Display**.
- There is a natural flow of information that suggests a linear ordering on these four threads: (1) User Thread receives input commands (mouse and keyboard) from the user. The User Thread processes this information and submits the request to the Manager Thread. (2) The Manager Thread, perhaps using some scheduling algorithms, and information about current bandwidths, converts these into appropriate requests for data from the Server. (3) The data received from the Server is processed by the Data Thread, and stored in some data structure. (4) The Display Thread takes information and displays it.
- This linear flow of information is not the only one. For instance, the User Thread may sometimes bypass the Manager Thread and directly send information to the Display Thread. For instance, a "panning" motion may require instantaneous response from the Display Thread, in addition to any other data that we may need from the network.
- The heart of the system is the Manager Thread which controls the flow of information going through the network. Communication between thread is accomplished by queues: there is a Manager Queue

where the Input Thread places requests for the Manager. There is a Display Queue where the Input Thread and Data Thread places information for the Display Thread. There is a Data Queue where the Manager informs the Data Manager about information that may arrive over the network.

- To minimize the bandwidth, we will try to guarantee that no redundant data is ever sent. This might seem to require sophisticated encoding techniques. E.g., if I request data for a given rectangular window, I may only need to send data corresponding to some subregion of this window, since the remaining data had previously been sent. But this seems to require very complicated encoding of this "subregion".
- One simple solution is based on the idea of "masks". Both the Server and Client maintains a mask of what information has been sent. Based on this, it is possible to avoid any duplication of data without complicated encoding.
- The foveate command on the Manager Queue has the following form:

$$FOVEATE(x, y, w, h, L_0) \tag{1}$$

where (x, y) is the gaze-point, (w, h) is size of the fovea, and L_0 is the foveal resolution.

- Assuming the wavelet based approach in earlier lectures, the network manager will transform this into a sequence of requests to the server, of the form

$$FOVEATE(ID, x, y, w, y, \ell, size).$$

The data for this request comes from the transformed matrix M_ℓ at level ℓ. Here $size$ is the number of bytes of data requested and ℓ is the resolution level ranging over $[L_{min}...L_{max}]$. (NOTE: $size$ is needed so that the data processing thread knows when the all the data have arrived over the network.)

- The range of levels is guaranteed to satisfy the bound

$$0 \le L_0 \le L_{min} \le L_{max} \le MAX\_LEVEL$$

(recall that the image has resolutions from 0 to MAX_LEVEL). Normally, $L_{min} = L_0$, unless the network manager seeks to trim the request. Also, L_{max} is always set to the largest level such that $size$ is greater than 0.

ID is a unique identifier generated by the network manager for the request equation (1). Thus, there are actually $L_{max} - L_{min} + 1$ requests sent on the network with this ID. The data processing thread needs this ID to reassemble the updated data.

§4. Introduction to Threads

- What are threads?
  - They are like processes, but exists within a process.
  - Each thread has its own execution state (registers, stacks, etc)
  - All thread's in a process share the same code, memory space.
- How are threads difference from Processes?
  - Thread is a user entity (exists only in user space). This view is only an approximation.
  - Processes is a kernel entity
  - Kernel calls are not needed to create a Threads
  - It is OK to create 100 thr's, but 100 processes are expensive.

  - "Light weight processes" (LWP) is sometimes a synonym, but has other meanings in some OS. E.g., in Solaris, a process can have several LWP, and a LWP can serve several threads's.

- Varieties of Threads's
  - LWP
  - sun, winNT, OS2, etc
  - pthreads: POSIX standard

  Function calls to the POSIX library are prefixed by `pthread_`. E.g. `pthread_create()`.

- Advantages of Threads
  - Better performance (in uni and multiprocessor environment)
  - One binary for both uni or multi environment
  - Better throughput in uni (e.g, a thread with a blocking system call will not tie up progress)
  - Cheaper than processes (e.g. thread communication is cheaper than inter process communication)

- Issues of Threads
  - Synchronization: simplest example is using mutex.
  - Scheduling: local vs global scheduling; how much can the user control this?
  - Signals

- Thread States
  - active
  - runnable
  - sleeping
  - stopped
  - zhombie (waiting to be collected)

§5. Attribute Objects

- POSIX Threads uses a concept called **attribute objects**. Attribute objects are an object-oriented approach to to encapsulate properties. Their sole purpose is to store a collection of **attributes**, and their associated **values**. The set of attributes is fixed, but their values can be freely queried and modified.

- In this introductin, we just describe two kinds of attribute objects: **thread attribute objects** and **mutex attribute objects**. For instance, a thread attribute object has an attribute called **scope**. The value of this attribute can be "process-level" or "system-level". A thread with "process-level scope" means it is only known within its process; in contrast, "system-level scope" means it is known to the kernel.

- How do we use attribute objects? Suppose `X` is a particular thread attribute object. When we create a new thread, one of the arguments can be `X`. The new thread will have all the attributes encoded in `X`. If we subsequently modify the attribute values in `X`, this has no effect on attribute values in the thread.

- We can see the advantages of having attribute objects: if we create many threads with the same attribute values, we only need to initialize them with the same attribute object. If we subsequently want to change some attribute values, it is sufficient to just modify the object. We can also reuse the attribute object: if you want a new thread T which differ by just one attribute value, you can modify the attribute object and use it to create T.

- Thread attribute objects. These objects has the following 5 attributes: scope, detach state, stack address, stack size, and scheduling.
- Examples of attribute values: We had already explained the scope attribute. The scheduling attribute can be FIFO or Round Robin (default is FIFO). Round Robin means each thread has only an allocated amount of time to run before being put back on the queue.
- In pthreads, you initialize and kill such an object using:

```
pthread_attr_init(pthread_attr_t * attrObj);
pthread_attr_destroy(pthread_attr_t * attrObj);
```

- Mutex attribute objects. Currently, these only have scope attribute (with the same values as for thread attribute objects).

§6. POSIX Threads

- Before you create a thread, you need to write a function `myThreadFunc`, which will be executed as the code of the thread:

```
void * myThreadFunc (void *rt) {
    int i;
    for (i - 0; i< 20; i++)
       printf("round    pthread_exit(NULL);
}
```

  This trivial example executes a for-loop 20 times. Note that the final instruction is a call to `pthread_exit()`.
- Under POSIX, the method to create, and to join (i.e., wait for a child thread to end) is as follows:

```
pthread_t myThr;                                      // declare thread object "myThr"
pthread_create(& myThr, NULL, myThreadFunc, NULL);   // create thread with default attribute values
pthread_join(myThr, NULL);                            // parent waits on the child thread to exit
```

  The "join value" (returned by the child thread) is discarded by the parent in this example. Otherwise, instead of NULL, it would be stored at the address indicated by the second argument of `pthread_join()`.

§7. Synchronization

- The simplest type of syncronization is mutual exclusion (MUTEX).
- In mutex applications, some portion of a thread code are designated as a **critical section**. This means that it is accessing some variables that is shared with one or more threads. Thus, the other threads will have a corresponding critical section. It is clear that no two processes must be in their critical section at the same time.

- A collection of critical sections (distributed over two or more threads) that must be mutually exclusive in this way can be protected by a single **mutex variable**. If `mx` is a mutex variable, the protocol is that you need to "lock" `mx` before entering the critical section, and "unlock" `mx` upon exiting from the critical section.
- In POSIX, you create, use and destroy mutex variables as follows:

```
pthread_mutex_t mx;                  // declare mutex variable "mx"
pthread_mutex_init( & mx, NULL);     // NULL is the default mutex attribute object
...
pthread_mutex_lock(& mx);            // lock on mx
...  critical section ...
pthread_mutex_unlock(& mx);          // unlock on mx
...
pthread_mutex_destroy(& mx);         // Destroy "mx"
```

§8. APPENDIX A: Non-Blocking Reads

- TCP/IP is a stream oriented service. The two basic operations (on either end of the socket) is a read() and write() function. Each write specifies a chuck of data to be sent. Each read specifies a buffer of some fixed size to receive data. On UNIX, read/write operations on sockets are logically equivalent to read/write on files or pipes.
- What does "stream oriented service" mean? Suppose the server performs a sequence of n successive writes on a socket connection: let $W_1, W_2, \ldots, W_n$ be the data that is written. On the other end, the client performs a sequence of successive reads on the same socket connection: let $R_1, R_2, \ldots, R_m$ be the data that is read. The only guarantee is that

  $$W_1 W_2 \cdots W_n = R_1 R_2 \cdots R_m.$$

  The boundaries between the chunks of data between successive writes are not guaranteed to be preserved as the boundaries between the chunkes of data that is read on the other end. In fact, $n \neq m$ in general.
- There are 2 kinds of read: blocking and non-blocking. In either case, if there is error, this can be detected. Otherwise, you are told how many bytes of data is actually read. You have to provide your own logic to know if there is more data coming or not.
- Unfortunately, specifying blocking or non-blocking is not part of the Berkeley sockets API. Here is how you do this in UNIX:

  You call a function `fcntl(int fileDes, int cmd, int arg)` where `fileDes` is the file descriptor (which in this case is a socket but can also be a file or a pipe). The commands `cmd` of interst here are `F_SETFL` or `F_GETFL`. The `arg` may specify non-blocking or other arguments.
- Here is a program to set the non-blocking flag for socket `fd`.

```
void nonblock (int fd) {
     int flags;
     flags = fcntl(fd, F_GETFL);
     flags |= O_NONBLOCK;                              ll
     if (fcntl(fd, F_SETFL, flags) < 0) { error(); }
}
```

- Suppose you have one thread that has to read from several possible sockets. Instead of trying each one in turn, a more efficient way is to use a function `select(int fd, fd_set *)` that allows you to find only those sockets that have data waiting to be read.

§9. APPENDIX B: Thread Program Example

- Thread Program directory: http://cs.nyu.edu/ yap/classes/visual/01f/lect/l10/prog/thread/
- thread.c program

THE END

References

[1] E. Chang, C. Yap, and T.-J. Yen. Realtime visualization of large images over a thinwire. In *IEEE Visualization '97 (Late Breaking Hot Topics)*, pages 45–48, 1997. See also CD proceedings of conference.

[2] E.-C. Chang. *Foveation Techniques and Scheduling Issues in Thinwire Visualization.* PhD thesis, Department of Computer Science, New York University, May 1998.