Tiled Display OpenGL Widget-based Application Framework

Justin Binns Mike Papka
Mathematics and Computer Science Division, Argonne National Laboratory
Argonne, IL, 60439, USA
fl-info@mcs.anl.gov

General Overview

The goal is to produce an application framework that allows for interchangeable 'widgets' with common interfaces to be included in a larger distributed tiled display application. Such a framework will include an event management system, a global position/scale state management system, a 'distributed' memory mechanism (for synchronized memory regions across all tiled display nodes), OpenGL context handling and localization, and a protocol-independent mechanism for defining external user interfaces.

In order to reach the above goals in reasonable time, some simplifying assumptions are made:

1) A single OpenGL context (with associated static coordinate system) is provided, within which all widgets must reside. This universal context and coordinate system serves as the 'desktop' or common locality mechanism (allowing consistent placement of widgets relative to each other, coordinated buffer swap, and potentially geometry and/or color correction code transparent to individual widget operation).
2) Only OpenGL drawing mechanisms will be used and only such mechanisms as do not corrupt the OpenGL context for use by other widgets. This implies that should a widget manipulate the OpenGL context during the course of its drawing process, it will return the OpenGL context to an unmodified state prior to exiting said draw process.
3) Widgets are well-behaved and well-coded (no security context or protection for code or memory regions between widgets is provided).
4) All message handling can be accomplished on the single 'master' node.
5) The localized 'slave' nodes (one per display) are capable of performing their function entirely based on read-only access to global data as well as widget-specific 'distributed' memory regions, with the additional localization information provided by the overarching Window Manager. Generally, this means that the slave nodes only draw, perhaps doing some minimal local calculation, based on information provided in a read-only manner through the global and per-widget distributed memory regions.
6) No time-critical operations are required for function. This is a weak requirement – optimization or careful construction of the event handling and synchronization subsystems may allow for selected time-critical operations, but in the general case (and to simplify first implementation) this will not be supported.

In addition to the above simplifying assumptions, standard message-based assumptions apply, such as timely message handlers, and well-formed messages.

Component Summary

A variety of components make up the system described above. These components fall into two broad classes – server-side components and client-side components. Server-side components run on a single, coordinating 'server' node. These components are responsible for all event handling, primary application processing, user interactions, synchronization, and general functioning of the application. The client-side components are instanced on each of the display nodes. They are responsible for taking the information provided by the server-side components and producing images when asked to.

The server-side components can be divided into two categories – infrastructure components and application components. Infrastructure components, including the memory management sub-system, the event handling subsystem, and the Window Manager, make up the critical infrastructure necessary to the operation of the system. Application components take advantage of the services provided by the infrastructure components to provide application functionality. In general, these Application components will be called 'Widgets'.

The Infrastructure components of the server-side system consist of the event management system, the memory management system, and the Window Manager. The event management system is the structure by which user interfaces drive the function of an application. Events enter the system through a User Event Manager, and are either dispatched (in the case of queries, which require responses to the user), or placed in the incoming event queue, which is managed by the Event Queue Manager. The Window Manager is responsible for the main event loop, which includes removing events from the queue and dispatching them to the various Widgets in the system (as well as handling some itself for Widget and global context management). Widgets, including the Window Manager, use the memory management system to store information that may be required either by client-side components (the memory management system periodically synchronizes the memory regions in use with the clients) or by request handlers (those functions that handle user interface interactions requiring responses, which are handled in real-time). The memory management system is important both for its shared-memory-like structure and for providing locks and access methods that ensure sequential access to memory regions in a multi-threaded environment.

The Widgets are the application bits of the system. They provide functionality that is of interest to the application designer and/or the user. Often, this functionality may be simple, but when combined with other widgets, and an appropriate client, a powerful system can develop. Generally, a server-side Widget will be made up of an event/query handler, which is responsible for performing appropriate actions and producing accurate responses when events and/or queries are dispatched to it, and a server component, which is responsible for specific action during the event loop, including any local computation necessary prior to the draw stage.

The client-side components can likewise be divided into the same two categories– infrastructure components and application components. Infrastructure components, including the client-side memory management system and the client-side Window Manager, make up the client-side portion of the infrastructure that their server-side counterparts support. The memory management system is responsible for synchronization of the various memory regions, and for providing the same kinds of access controls and methods to client-side components as to server-side components (making the memory management system appear to be a single system instead of a server-side and client-side model). The client-side Window Manager is responsible for receiving and acting on instructions from the server-side Window Manager, such as the initiation of memory synchronization, or of a draw phase. The client-side Window Manager also sets up the common OpenGL context prior to draw, applies any last-step changes after draw (such as geometry and/or color correction), and performs synchronization of buffer-swap to guarantee in-step processing of the entire distributed application.

The client-side application components, or client-side Widgets, are responsible for using the data provided by their server-side counterparts, in conjunction with localization information provided by the client-side Window Manager, to appropriately modify the screen and/or operating environment on that client. The intention is that, aside from the actual drawing, the computation necessary in the client-side Widgets should be minimized. It is important to note that it is the responsibility of every Widget to respect the shared nature of the OpenGL context. Each widget that makes use of the OpenGL context MUST return the context to its received state prior to termination of the draw cycle (i.e. Widgets MUST NOT corrupt the context).

Window Manager Execution Cycle

Much of the design of the system is based around the execution cycle of the Window Manager. We are primarily concerned with the server-side Window Manager, as the client-side Window Manager Execution Cycle is considerably simpler. The server-side cycle proceeds as follows:

1) Pre-cycle Update – the Window Manager updates any internal non-event-based state that needs to be updated before the execution cycle begins. This may include incrementing a cycle counter, clearing a set of status registers, or generally preparing for the execution cycle. In the current sample implementation of the architecture, nothing is done at this stage.
2) Pre-Event Widget Updates – the Window Manager calls the ‘updatePre()’ function on each of the widgets, in order from most recent focus to least recent focus. This update allows each widget to perform similar preparation as (1) above allows for the Window Manager.
3) Event Processing – the Window Manager dispatches all pending events, handing each event off to the widget with most recent focus, and continuing to pass each event down as long as the event is returned. If the event is not returned, it is considered consumed. If an event is returned by the last widget on the stack, the Window Manager attempts to dispatch it internally, then it is discarded (and considered consumed).
4) Post-Event Widget Updates – the Window Manager calls the ‘updatePost()’ function on each of the widgets, in order from most recent focus to least recent focus. This update allows each widget to perform calculations and updates based on those messages which have been dispatched during this execution cycle.
5) Post-Event Update – the Window Manager performs post-event updates on itself. As a final part of this, the Window Manager triggers a memory system

synchronization, which synchronizes any memory region that has changed during this execution cycle. The Window Manager then checks the state of the flag that indicates a graphic update is required (settable by any widget during (3) or (4) above), and if it is set, initiates a draw.

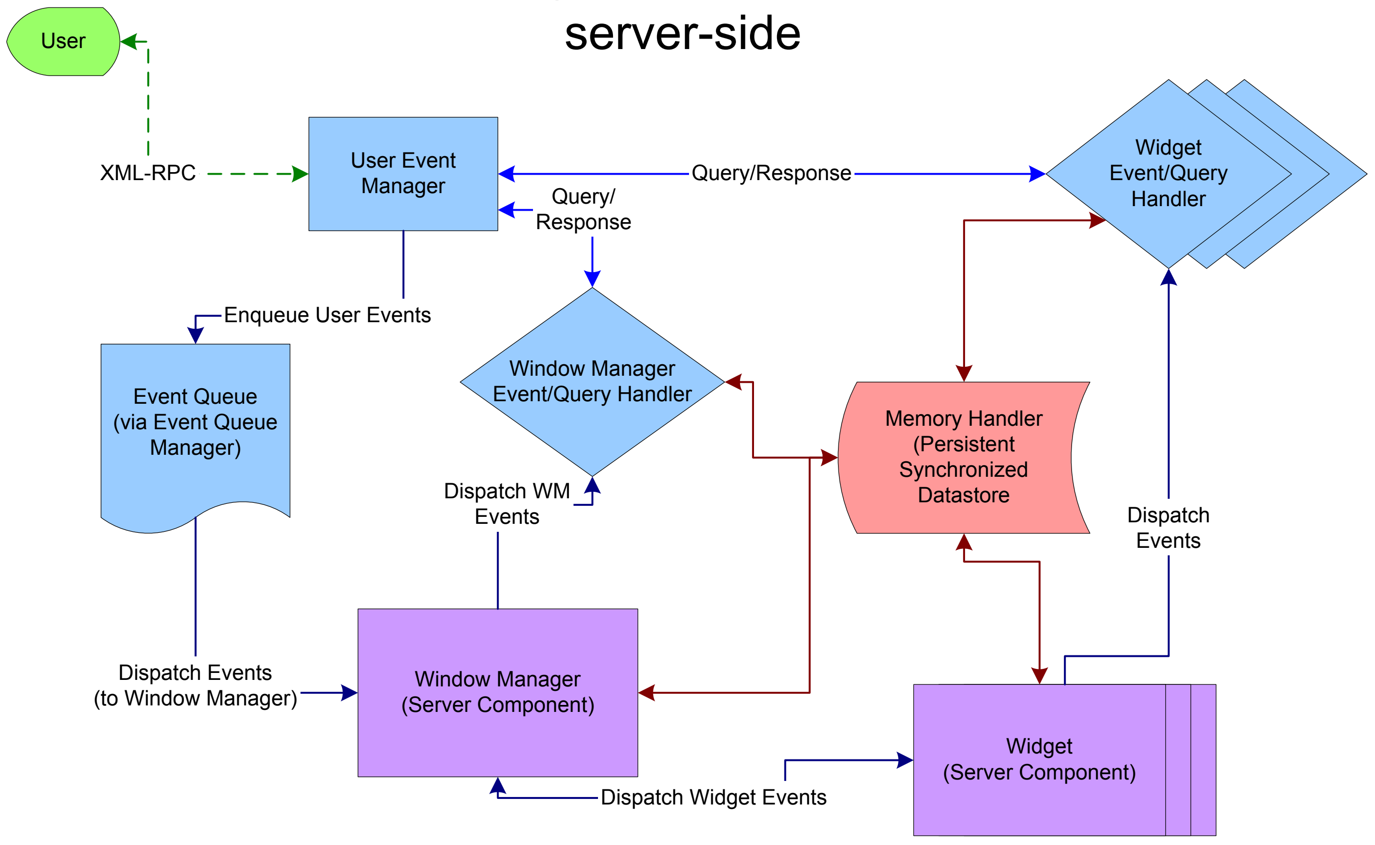
6) Event Queue Swap – the Window Manager swaps the empty Event Queue used in this execution cycle for the queue that was being populated during execution. This double-buffering of the event queues allows for smoother client interaction, reduced likelihood of deadlock, and batch processing of events (which is especially important with slow-executing widgets or widgets with long draw cycles).

For completeness, the client-side cycle should be discussed, though it is primarily reactionary, and dependent on messages from the server-side cycle. On a synchronization message, the memory system synchronization is performed. On a draw message, each widget is called to draw itself, from the 'bottom up' (or from the widget with least recent focus to the widget with current focus). On a quit message, each widget is asked to clean itself up and exit cleanly, after which the Window Manager does likewise.

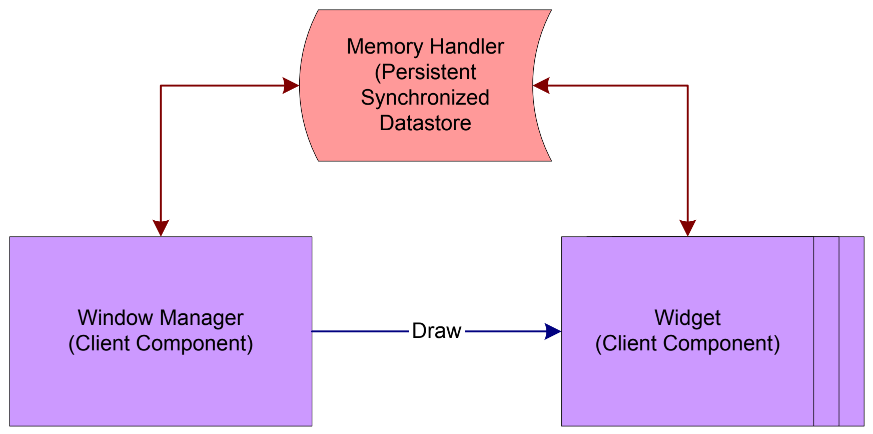
The execution cycle described above for the server-side Window Manager exposes several key concepts within the system. First, the entire system is driven from the master. This is important, as it allows all user interface functionality and complex management tasks to be centralized, reducing the likelihood of drift or loss of synchronization that might occur if we attempted to process complex tasks on each local node separately. Second, the memory synchronization is performed only on those regions that are changed. The memory handling system has a system of locks allowing simultaneous access to data, but also guaranteeing dedicated access when changes are being made. These locks allow for the memory system to itself be aware of which regions are changed, and only synchronize those regions, thereby reducing network load and synchronization delay. Third, the concept of focus is important, as it defines the order in which various widgets get access to messages. Many behaviors are allowed by the system, including, potentially, an semi-automated 'click-to-focus' policy, which would make it the responsibility of each widget to pass messages on that do not pertain to their own managed areas. The vast majority of events are widget-specific, however, leaving implementation of such policies up to the client.

The last thing the above execution cycle demonstrates is the double-buffered nature of the event queues. This allows for a queue of parsed events to be built up and processed as a batch. Such a system reduces system lag caused by the event parsers (i.e. the XMLRPC server) having to wait for an opportunity to invoke an event. Instead, the event is parsed as quickly as possible and simply dropped in the current background queue. That queue is then swapped into the foreground during the next execution cycle, and all queued events are processed in batch. The constraint this places on the system is that all events must be simple messages, no message-response format is allowed. Queries for state need to be supported, and are handled by the locking mechanisms in place within the memory handling system. Queries are serviced immediately, with the constraint that any user-accessible information must be maintained within the memory management system, to provide locking and protection from race conditions.

tdWidget Architecture
server-side


User
XML-RPC
User Event Manager
Query/Response
Query/ Response
Widget Event/Query Handler
Enqueue User Events
Event Queue (via Event Queue Manager)
Window Manager Event/Query Handler
Memory Handler (Persistent Synchronized Datastore
Dispatch WM Events
Dispatch Events
Dispatch Events (to Window Manager)
Window Manager (Server Component)
Widget (Server Component)
Dispatch Widget Events

tdWidget Architecture
client-side (per display)


Memory Handler (Persistent Synchronized Datastore
Window Manager (Client Component)
Draw
Widget (Client Component)