A Trip Up the Looking Glass Event Pipeline

Deron Johnson
Version 1.0
Thu Dec 8 06:26:06 PST 2005

1 Introduction

This document describes the processing that Project Looking Glass (LG) performs on input events, from the time they are generated by the input kernel driver to the time when they are delivered to applications. Because all user-visible objects on the LG display are 3D objects, even 2D windows, which are textures mapped onto 3D objects, it is necessary for LG to perform a variety of processing on input events that are not necessary in a conventional 2D window system.

This document presents the stages of LG input event processing from an event's eye view. We will start at the point where the event data is read from the input device kernel driver and follow it upward has it is handed off to successively higher layers of software. But first we will present a brief overview.

In this document we will not go into very much detail on the existing X server event processing. Instead we will focus primarily on the LG additions to the X event processing pipeline. A certain amount of familiarity with Xorg server internals is assumed. Readers not familiar with how the X server handles events should consult references [1,2]. It is recommended that the reader follow along in the LG X Server and LG Display Server source code while reading this document. This is not strictly necessary, but many concepts may be clearer when this document is used in conjunction with the the source code.

For more information on the architecture of Project Looking Glass and it's 3D client API refer to [3,4,5].

Note: readers who are only interested in the changes LG has made to the X server should refer to Section 26 for a list of the sections in this document that specifically deal with the LG X server changes.

2 LG Display System Overview

The LG display controller consists of two processes: a modified Xorg Server (XS) and a new Display Server (DS). The DS owns the screen; everything that is displayed on the screen is drawn by the DS. When an X11 application maps a window a corresponding *window avatar* (a 3D object) is created and added to the Display Server's scene graph. This scene graph is a data structure which lives in the address space of the DS and which describes all of the 3D objects visible on the screen. Typically, a window avatar is a rectangular slab (strictly speaking, a parallelepiped). The contents of the window are captured by the XS (via the Composite extension) and these contents are sent to the DS via shared memory. The DS download the window contents image into a texture in the 3D graphics device. The texture is mapped onto the window avatar, thus making the window contents visible on the screen. As the X11 app performs further rendering on the X window, the modified window contents images are sent to the DS and downloaded into the texture. In this way continuous changes to the window contents become continually visible on the screen.

Just as the DS and XS cooperate in making window contents visible, the DS and XS also cooperate in processing input events. In order to give an X11 app the illusion that it is running in a conventional 2D window system, events that are generated when the mouse cursor is over a particular window avatar are translated into the coordinate space of that window. The event is then sent upstream where all of the normal processing of an X11 event is performed upon it.

In LG, in addition to containing the avatars of X11 windows, the scene graph can also contain 3D objects which are not associated with any X11 window, but rather, which have been created by a new type of windowing application: a LG3D application. This is a type of application that knows that it is running within a 3D window system and which creates and manages 3D user interface objects using the LG Client API. If an input event is generated when the cursor is over one of the objects of an LG3D app, a new type of event (a *3D event*) is generated and is distributed to special event listeners registered by the application. These 3D events are not distributed to LG3D applications via the normal X event distribution mechanism. Instead they are distributed via mechanisms provided by the LG Client API.

In any window system which is displaying an assortment of objects and which is associating input events with these objects, it is necessary to provide proper synchronization between changes in the display and the input event stream. For example, if the user clicks a mouse button on a menu item to bring up a text window it is often desirable to allow the user to type ahead keystrokes even before the text window is displayed and for these events to be sent to the text window when it eventually appears on the screen. Also, it is often desirable for a scroll bar to continue to receive mouse events as long as the mouse button is still held down. This allows the application to be tolerant if the cursor accidentally "drifts" outside the window while scrolling. (It is almost impossible for users to move a mouse in a perfectly vertical path, and if a scroll bar is narrow it would be frustrating for the user to have the scrolling stop if the cursor moved outside of the window by a smalll amount).

To this end, the X11 window system provides a sophisticated (or, as some think, grossly overcomplicated) set of mechanisms for controlling input device synchronization and delivery. These mechanisms allow window managers and applications to properly synchronize display changes with event delivery, to freeze the event queue at various times, and to "snoop" events which are being delivered to applications, among other things. It is also necessary to provide at least some of these mechanisms in a 3D window system. For example, if you are scrolling a 2D app by dragging a scroll bar elevator and the cursor just happens to drift over onto an object of 3D app, we do not want the scrolling to stop. In other words, 2D and 3D events must be uniformly treated by the event system; 3D events must receive the same processing and sychronization that 2D events receive. If the event queue is frozen, both 2D and 3D events alike must be frozen.

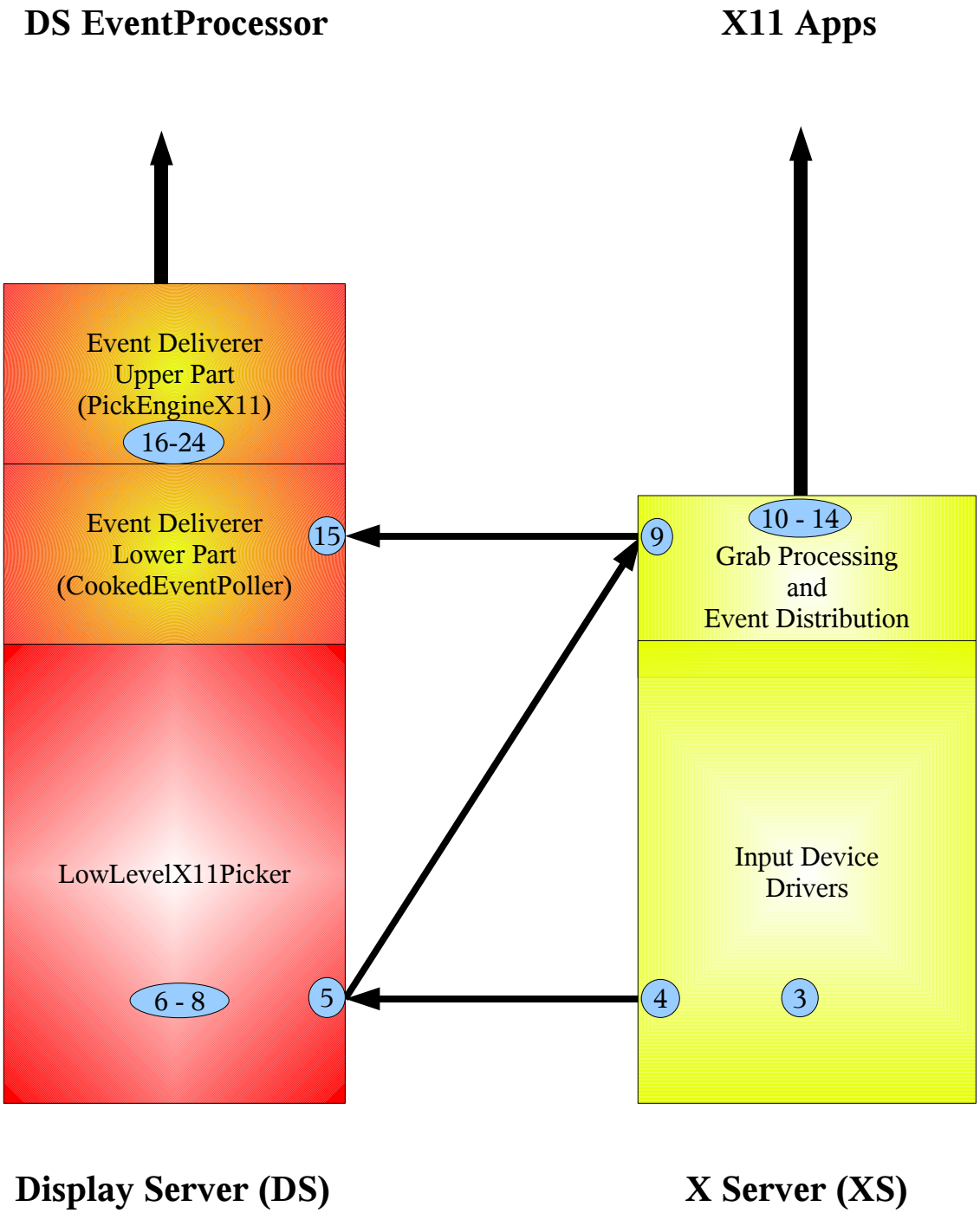
A significant goal of the LG input system (or *event pipeline*) is to provide consistent treatment between 2D and 3D events. In order to achieve this, 3D events are subject to much of the same processing as 2D events. Initially, there is a stage where it is not known whether the event is 2D or 3D. The event coordinates are used to perform a *pick* operation on the scene graph. A ray is fired from the eye point extending into the scene graph and passing through the event coordinates. If the object *hit* (i.e. intersected by the ray) is an avatar for an X11 window the event is 2D. If the object picked belongs to an LG3D app the event is 3D. But, at this point in the pipeline, the event type is merely saved with event and sent upwards through the rest of the pipeline.

Event synchronization control is provided by XS via an Xlib API called the *grab interface* (see chapter 12 of [2]). This API allows an X11 client to freeze (i.e. stop processing of) the event queue of a particular type of device and to restart it. It also allows clients to *grab* the event stream of a particular device and force all events to be sent to this client for a period of time. LG does not at this time provide a similar interface for LG3D apps, although the fundamental support is built into the system. We decided not to make this interface public at this time because it's not clear if this is the architectually correct kind of synchronization we want for 3D apps. But the underlying grab mechanisms are available should we decide to export them.

Events are read from the kernel driver by XS and are sent to the DS. The DS has the scene graph data in hits address space and so it must be the one to perform the pick. After the DS performs the pick, it records in the event whether the event is 2D or 3D and then sends the event back to the XS server. The event flows upward through the normal X event processing until the time comes to deliver the event to a client. At that point, if the event is 2D it is sent to the X11 apps that have registered interest in the event. But if the event is 3D it is sent back to the DS, where it undergoes further processing and is eventually delivered to LG3D applications that have registered interest in the event.

This event pipeline architecture is called the *Z Model* because the event path follows a Z shaped path in the block diagram, as can be seen in the following figure. This diagram also indicates which sections in this document describe the various parts of the architectural block diagram. (Specifically, the pertinent section numbers are provided in the small circles and ovals).

Z Model Event Path



Although it looks overly complicated on the surface, several considerations lead to the Z Model being adopted as the architecture for the LG event pipeline. It was decided to have the XS read from the input drivers because all of the necessary software had already been written and the overhead of sending the events from the XS to the DS (the lower part of the Z path) is negligible. It was decided to perform the bulk of X11 event and grab processing in XS because the code which performs this processing is extremely complicated and moving it outside of XS into the DS could introduce bugs and added complexity. Again, the overhead of sending the picked event from the DS to the XS is negligible. Thus, although additional overhead is added by ping-ponging events between the XS and DS, we have in practice found the overhead to be minimal and the Z Model to require the fewest code changes to the X Server.

Note: LG does not currently support the XINPUT extension, which supports other types of input devices besides a keyboard and a mouse, such as buttons, dials, etc. It is intended that this support will be added in future versions.

3 XS: Device Event Generation

With the overview concluded we will now proceed to follow an event on its journey up the LG event pipeline. Our journey starts in `dix/dispatch.c:Dispatch`.

(Note: unless otherwise specified all file paths for files mentioned in the XS sections of this document are relative to the top of the X server source tree, which is located in `xc/programs/Xserver` in the Xorg CVS tree. The CVS branch `lg3d-dev-0-7-1` contains the latest LG source code).

`Dispatch` continually loops. On each iteration of the loop `os/WaitFor.c:WaitForSomething` is called. This routine performs a `poll` (or `select`) system call on a list of file descriptors. This list includes the file descriptors of the mouse and keyboard devices. When input data is available to be read from these devices, the poll will return with a mask indicating which device(s) have data to be read. The wakeup handlers are then called. `hw/xfree86/common/xf86Init.c:InitOutput` registers the routine `hw/xfree86/common/xf86Events.c:xf86Wakeup` as a wakeup handler at server init time. This is one of the routines called after the `poll` returns.

For each input device which has data to be read, `xf86Wakeup` calls the device's `read_input` routine. For the mouse, this routine is `hw/xfree86/input/mouse/mouse.c:MouseReadInput`. This routine reads the data by calling `hw/xfree86/common/xisb.c:XisbRead` and constructs a mouse event and then calls `mouse.c:MousePostEvent`, which in turn calls `mouse.c:MouseDoPostEvent`. In this routine, if the event is a button event `hw/xfree86/common/xf86Xinput.c:xf86PostButtonEvent` is called to place the button event into the device event queue (`xf86EventQueue`). If the event is a motion event `hw/xfree86/common/xf86Xinput.c:xf86PostMotionEvent` is called to place the motion event into the device event queue.

For the keyboard, on Linux, `xf86Wakeup` calls `hw/xfree86/os-support/linux/std_kbdEv.c:xf86KbdEvents`. This routine reads data

from the keyboard device file descriptor and calls `hw/xfree86/common/xf86Events.c:xf86PostKbdEvent` to place the keyboard event into the device event queue.

Back in the dispatch loop in `dispatch.c`, in addition to calling `WaitForSomething` on each iteration, the contents of two global pointer variables (`*checkForInput[0]` and `*checkForInput[1]`) are continually compared to see if they are different. When these values are different it means that there are events in the device event queue. (This is because `hw/xfree86/common/xf86input.c:xf86eqInit` calls `dispatch.c:SetInputCheck` to set these pointers to point to the head and tail pointers of the device event queue).

When the head and tail pointers of the device event queue differ then it contains events. In this case, `hw/xfree86/common/xf86Events.c:ProcessInputEvents` is called. This routine calls `hw/xfree86/common/xf86Xinput.c:xf86eqProcessInputEvent`s (because the compile flag `XINPUT` is defined).

For mouse events, `xf86eqProcessInputEvent`scalls `xkb/xkbAccessX.c:ProcessPointerEvent`. (This routine is configured in `dix/devices.c:_RegisterPointerDevice` because the flag `XKB` is defined). `ProcessPointerEvent` calls `dix/events.c/CoreProcessPointerEvent`.

For keyboard events, `xf86eqProcessInputEvents` calls `xkb/xkbPrKeyEv.c:ProcessKeyboardEvent` which, eventually, calls `xkb/xkbPrKeyEv.c:XkbProcessKeyboardEvent`.

4 XS: Sending Device Events to the DS

Up until this point, all of the code described has been standard Xorg code. This is the point where the first LG-specific code appears.

Throughout the XS, most LG-specific code is conditioned by the boolean variable `lgeDisplayServerIsAlive`. This indicates that the server is in *LG mode*. Only when this variable is true does LG-specific event processing occur. This variable is set if there is a DS process running. The DS puts the XS into LG mode by invoking an LGE extension request. If there is no DS, the XS reverts to normal X event processing mode and no LG-specific event processing occurs. `lgePickerClient` indicates the DS client connection that performs picking. `CoreProcessPointerEvent` and `ProcessKeyboardEvent` make sure that the DS client is still alive before sending events to it.

`CoreProcessPointerEvent` (and `XkbProcessKeyboardEvent`) are called in two places. One source of calls is from `ProcessInputEvents` (as described above). The other source of calls is from the LGE extension. When the DS has finished performing picking for a device event it sends the event back to the XS via an LGE extension request. This request will call `CoreProcessPointerEvent` (or `XkbProcessKeyboardEvent`). Because these routines need to do different processing depending on where they are called from there is a boolean variable `lgeEventComesFromDS`. This variable is false when the calls are being made from

`ProcessInputEvents` and true when the calls are being made from LGE.

If the call is coming from `ProcessInputEvents` (i.e. `!lgeEventComesFromDS` is true) `CoreProcessPointerEvent` fetches the pseudo-root-window (PRW[1]) in which cursor currently resides (note that LG is capable of supporting multiple screens). The event's `root` field is set to this window. (`XkbProcessKeyboardEvent` does the same thing). Then the event is then sent to the DS. This is the lower horizontal segment of the Z path.

(One area that deserves further exploration is whether the server should be grabbed (i.e. requests should be blocked) while events are being sent to the DS for Picking. This was suggested by Keith Packard. This would prevent clients from changing the window tree while picking is occuring. For example, if the DS picks an X window that is in the process of being deleted from the window tree it will end up assigning a nonexistent XID to the event. This can cause confusion later in the event pipeline. In practice, however, we have not found this to be a problem. And grabbing the X server in this way could cause animating programs to become jerky. So at the present we do not implement this scheme. Instead, we just discard events with windows that no longer exist in the tree. But it would be worthwhile exploring this issue further to see whether animation jerkiness actually occurs).

5 DS: Event Entry into the DS

(All paths mentioned in sections about the DS are relative to *lgroot*`/src/classes/org/jdesktop/lg3d/displayserver/fws/x11` unless otherwise specified. *lgroot* is the root of the lg3d-core directory in the java.net CVS tree. The latest code is in the trunk of the CVS tree. For information on how to access this CVS refer to `http://lg3d-core.dev.java.net/lg3d-developers-guide.html`).

In the DS, the `DeviceEventSourceSession` thread continually tries to read device events from the XS, blocking if none is available.

Each incoming event is assigned a *pick sequenc*e number. This is a continually incrementing number (with wrapping at `DeviceEventSource.LG3D_PICK_SEQ_MAX`) which serves to uniquely identify events in subsequent event processing. We will see how this is used later in Sections 7 and 20. The `root` field of the event is used to determine in which `Canvas3D` the pointer resides. Also, the coordinates of button and keyboard events are set to the coordinates of the last motion event (the XS only assigns coordinates to motion events and the picking code requires all events to have valid coordinates). Finally, the event is appended to the device event queue. At this time, if there are consecutive motion events in the queue the earlier ones are discarded and only the latest motion event remains in the queue. (This *motion compression* step is critical for good interactive performance).

6 DS: Picking

1 The PRW is a window that is a child of the actual root window. It has the same shape and size as the root window and always occupies the full screen. The DS performs all of its drawing on this window instead of the actual root window because not all systems allow 3D rendering on the root window).

The picking operation is performed by an instance of the singleton class `LowLevelX11Picker`. This is a Java3D behavior which wakes up once per rendering frame and checks to see if there are events in the device event queue. For each motion or button event, a picking operation is performed (key events inherit the picking information from the previous motion or button event). This section describes what occurs during picking operations.

Here is what is involved in picking:

First, the event's `Canvas3D` is used to find the corresponding Java3D `PickCanvas`. This is a class which Java3D uses to perform picking on the scene graph. Next, the event coordinates are given to the `PickCanvas`. Previously, the `PickCanvas` has been informed of the location of the view point. With these two pieces of information, the `PickCanvas` constructs a ray which extends from the view point into the scene, passing through the event coordinates, which are assumed to reside in the Z=0 plane. The picker then calls `PickCanvas.pickAllSorted`. This method returns a list of all scene graph nodes which are intersected by this ray. Note that the DS uses fine grained picking. That is to say, it intersects the ray with the actual geometry of the scene graph nodes, rather than merely their bounding boxes. The first node in the list returned by `pickAllSorted` is called the *hit node*.

There are three possible outcomes of the picking call:

1. The hit node could be null.

   This is called a *pick miss*. In this case, the event window is set to the PRW, the event coordinates are set to the screen absolute coordinates and the event is sent back to the X server. Because the event window is the PRW, the X server treats this event as a new type of event: a 3D event. 3D events receive special processing in the X server.

2. The hit node is an avatar of an X11 window.

   In this case the method `findIntersectionPoint` is called. This method translates the x, y coordinates of the intersection of the pick ray and the avatar into window relative x, y coordinates. These coordinates are stuffed into the event along with the window ID of the avatar's X11 window and the event is then sent back to the X server. To the XS, this event will appear to be a normal 2D input event generated over a normal flat X window.

3. The hit node is an object belonging to an LG3D application.

   Several processing steps are performed in this case. First, the event window is set to the PRW and the event coordinates are set to the screen absolute coordinates. Next, the event is tested to see whether it **might** activate a grab (more on this in Section 8). Then, the pick information which has been accumulated for the event so far is saved and is associated with the event. Finally, the event is sent back to the X server. Because the event window is the PRW the X server will treat this event as a 3D event.

Note: the `LowLevelX11Picker` is also responsible for moving the cursor in response to mouse motion events and for changing the cursor visual representation depending on representation required

for the current hit node.

7 DS: 3D Event Information

As mentioned above, after picking as been performed on an event it is sent back to XS for further processing. If the event is a 2D event, it will eventually be sent to some conventional X11 client. But if the event is a 3D event, it will eventually be sent back to the DS. The DS will perform further processing and send it to LG3D applications. In order to perform this further processing and route the event to the proper destination, various information about the pick results is needed. This information is called *3D event information*. This information is calculated by the `LowLevelX11Picker` and is used later in the DS part of the event pipeline.

So this information must remain associated with the event for the remainder of the pipeline. In order to achieve this, one option would be to extend the size of the X event structure to hold the additional data. However, X events are constrained to be 32 bytes in length and the 3D information is far larger than this limit. So instead we adopt a simpler expedient: we put all of the 3D information for the event into an instance of class `EventInfo3D` and put a DS-internal reference to this object in an unused field in the X event structure. As the event flows through the XS event processing machinery, this reference will follow along with it. This reference is called a *pick sequence number* and is guaranteed to be unique[1].

The `LowLevelX11Picker` saves the following information in the `EventInfo3D` objects:

- The pick sequence number.
- A list of objects which provide information about what 3D objects have been intersected by the pick ray[2].
- Whether the event might trigger a grab.

The `EventInfo3D` objects are stored in a queue. The `LowLevelX11Picker` adds `EventInfo3D` objects to the end of the queue. When the 3D events return to the DS the `EventInfo3D`'s will be taken off the front of the queue and will be matched up with the events. Since the events stay in order as they are processed by the X server it is straightforward for the DS to match up an event with its corresponding `EventInfo3D`. Refer to Section 20 for further details.

8 DS: Grab Trigger Pre-Test

Another operation that the `LowLevelX11Picker` performs when the hit node is a 3D object is to test

---

1This guarantee is based on the assumption that DS and the XS can together process enough events to keep the number of in-flight events below 65535.

2 Note that the `PickInfo` list holds intersection information for each and every 3D object intersected by the pick ray. The current implementation of LG only utilizes the object with the intersection closest to the view point. But future GUI techniques, such as 3D drag-and drop, may use information about the other intersections.

to see whether the event can potentially trigger a grab. Note that we say "potentially." This is because only the XS knows whether the event will actually trigger a grab. For example, if a 2D grab is already active a button event over a 3D object cannot trigger a 3D grab; only one grab per device can be active at a time. But what the DS does is to pass along with the event information about whether the event can potentially trigger a grab. This type of grab is called a *passive grab* because it requires an event to activate it.

If the event type is one that can potentially trigger a passive grab (such as a button[1] or key event) LowLevelX11Picker first checks the *passive grab table* to see if one of its grabs might be triggered by the event. The passive grab database is a collection of grabs registered by LG3D apps.

(Note: we currently do not publically export the interface to register grabs in the passive grab table. This is because it is not yet clear whether this X-centric synchronization model is appropriate for 3D applications; we do not yet have enough experience with these. But the mechanism is integrated into the LG event path so that we can experiment in this area)

Since the check for a grab trigger from the passive grab table will always fail, the next check that the `LowLevelX11Picker` performs is to see if a *default button grab* triggers. In X, a default button grab is an implicit grab that is predefined for all windows: if the user presses a mouse button and holds it down a pointer grab is initiated. This means that all pointer events (both motion and button events) are sent to clients interested in the window over which the button press occurred, regardless of where the pointer actually is on the screen. Pointer events continue to be sent this window until the user terminates the grab by releasing the button. (Also note that all other mouse buttons must be in the released state; this is an X11 semantic).

Just as X supports an implicit passive grab for its 2D X windows, LG supports the concept of an implicit passive grab for its 3D objects. In `LowLevelX11Picker.testForPassiveGrabStateChange`, a call is made to `PassiveGrabState.checkForPassiveGrabRequest` to query the passive grab table. As mentioned above, this always returns null. So next, `testForPassiveGrabStateChange` calls `PassiveGrabState.determineDefaultButtonGrabRequest` if the event is a button. If the event is a button press, a 3D grab might happen and the DS must tell this to the XS.

The DS uses a special protocol to inform the XS that this event might cause a 3D grab to trigger. Specifically, it sets the value of the event's `child` field to be a certain value, `REQUEST_PASSIVE_GRAB_TRIGGERED`. Also, if the event is a button release that might terminate an existing 3D grab, the value `REQUEST_PASSIVE_GRAB_TERMINATED` is placed into the event's child field. (Note that the `child` field is uninitialized at this point so it is okay to "hijack" it).

We will see how these codes are used further on in this document.

9 XS: Receiving the Picked Event

1Although mouse wheel events look like button events to the DS, they are explicitly disallowed from triggering grabs.

The DS sends processed events back to XS via the `XlgeSendEvent` request. This is the diagonal segment of the Z path. This request calls `XkbProcessKeyboardEvent` or `CoreProcessPointerEvent`, depending on the event type. But first it sets a special global flag (`lgeEventComesFromDS`) to change the behavior of these routines. The first time these routines were called the events were sent to the DS Picker. But now these routines must pass the events further up the event pipeline.

10 XS: Keyboard Event Processing

`XkbProcessKeyboardEvent` performs various XKB processing operations (which we won't detail here) and eventually calls `dix/events.c/CoreProcessKeyboardEvent` (see various places in `xkb/xkbActions.c`).

Next, if the XEVIE (accessibility support) extension is active, the event is sent to the Accessibility Manager process. This process will eventually return a stream of one or more (possibly modified) events back to XS and `CoreProcessKeyboardEvent` will eventually be called for each key event in this stream.

After the XEVIE check, `CoreProcessKeyboardEvent` performs a variety of key-related operations. The most interesting of these is the to call `CheckDeviceGrabs`. If there is no grab already active and a `KeyPress` arrives, `CheckDeviceGrabs` will be called to see if this `KeyPress` triggers a grab. If the event is a 2D event, it will scan the list of passive key grabs which have been registered with XS for the event window. If the event is a 3D event, it will use the grab pre-test information in the `child` field of the event, which was placed there by the DS Picker (see Section 8). This field will indicate whether the a 3D grab should be triggered (if possible).

If a keyboard grab is already active when the `KeyPress` arrives, `CheckDeviceGrabs` will not be called. This is because a passive grab cannot override a grab which is already active. For 3D events, this means that the grab pre-test information in the event will be ignored.

If the event is a `KeyRelease` and a keyboard grab is active, this routine checks to see if this terminates the grab. We will describe this further in Section 13.2.

11 XS: Button Event Processing

`CoreProcessPointerEvent` is used to process pointer events returning from the DS. First, if XEVIE is active, the event makes a round trip to the Accessibility Manager and back (just as with key events this can generate multiple events from a single pointer event).

Next, the `DeviceEventCallback` routine is called (if defined).

Next, if the event is a `ButtonPress` and no pointer grab is already active `CheckDeviceGrabs` is called to see if this `ButtonPress` triggers a grab. If the event is a 2D event, it will scan the list of passive button grabs which have been registered with XS for the event window. If the event is a 3D

event, it will use the grab pre-test information in the `child` field of the event, which was placed there by the DS Picker (see Section 8). This field will indicate whether the a 3D grab should be triggered (if possible).

Similarly to key processing, if a pointer grab is already active when the `ButtonPress` arrives, `CheckDeviceGrabs` will not be called. For 3D events, this means that the grab pre-test information in the event will be ignored.

Grab triggering and termination will described further in Section 13.

The foregoing has described how `CoreProcessPointerEvent` handles button events returning from the DS. If the event is a motion event different processing is performed. This will be described in the next section.

12 XS: Motion Event Processing

If the event is a motion event, `CoreProcessPointerEvent` calls `CheckMotion`. This routine has been modified to do some special LG processing.

In normal Xorg code, `CheckMotion` updates the global sprite coordinates and keeps track of what screen the pointer is in. It also performs cursor confinement if necessary[1].

The next thing the unmodified `CheckMotion` routine did was to determine the window the event was over, by calling the routine `XYToWindow`. In the LG-modified routine, a bit more than this happens.

First, if the event is null (which happens when the window tree has changed) the normal `XYToWindow` routine is called. But if the window of the event is a PRW, the event is considered to be a 3D event and there is no need to recalculate the window field of the event.

But if neither of these cases applies, the event is considered to be a normal 2D X event. In this case, the DS has already figured out the top-level window for us and has placed it in the window field of the event. We can use this as a starting point from which to search downward through the window tree to find the lowest level window in which the cursor resides. For this, we call a new LG routine `XYToSubWindow`[2].

13 XS: Grab Processing

This section describes the grab processing performed by XS on both 2D and 3D events.

---

1 Any attempt by an X11 app to confine the cursor while in LG mode is ignored because developers of modern apps have figured out that this is a very bad user interface practice.

2 `XYToSubWindow` is in a file called `dix/xytosubwin.c` and it needs to call the routine `events.c:PointInBorderSize`. `PointInBorderSize` has file (static) scope in `events.c` so LG has modified it to have global (external) scope.

13.1 Grab Triggering

In `CoreProcessKeyboardEvent` and `CoreProcessPointerEvent`, if no grab is already active `CheckDeviceGrabs` is called to see if a key or button event triggers a passive grab. 2D and 3D events are handled differently.

13.1.1 2D Events

2D events are handled in the way XS normally checks for grab triggers. First, if there is a focus window, the window tree is searched upward toward the root window for the first window which has a passive grab registered for the event type. If one is found, that grab is triggered (i.e. activated).

If no appropriate grab is found in the previous search, another search is performed starting in the window in which the cursor resides (aka the *sprite window*). This search proceeds upward toward the root looking for the first window which has a passive grab registered for the event type. If one is found, that grab is triggered.

A passive grab triggered by a 2D event is called a *2D grab*.

Note: the triggering `KeyPress` or `ButtonPress` event is always sent to the client which registered the passive grab.

13.1.2 3D Events

Grab triggering for 3D events is handled differently than for 2D events. The DS has already determined whether the event can trigger a grab. At this point, we know that no other grab is active, so the 3D event triggers a grab. This grab is called a *3D grab*.

If a 3D grab is triggered `Activate3DPassiveGrab` is called. This routine constructs a `GrabRec` object with the appropriate fields and calls the `ActivateGrab` routine of the input device. It then calls `FixupEventFromWindow` to transform the event coordinates into the coordinate space of the event window and then sends the event to the Event Deliverer client in the DS (this client is registered with the XS at start up time). Finally, `Activate3DPassiveGrab` performs some code which deals with synchronous grab handling. (Synchronous grab handling is an extremely complicated aspect of the X grab mechanism and we will not deal with it in this document).

`ActivatePassive3DGrab` also updates the `child` field of the 3D event (which is not used for anything else). This field is set to `DELIVERY_GRAB_STATE_PASSIVE_GRAB_ENABLED`. This will tell the DS that a 3D passive grab has been triggered. (This will be described further in Section 13.3).

Note: the triggering `KeyPress` or `ButtonPress` event is always sent to the DS Event Deliverer client.

13.2 Grab Termination

The foregoing sections have described how grabs are triggered. We will now discuss how grabs are terminated (i.e. deactivated).

In `CoreProcessKeyboardEvent`, if a keyboard grab is active and a `KeyRelease` event arrives we check to see if the grab resulted from a passive grab[1] and if the `KeyPress` which triggered the grab was the same key as the `KeyRelease` event. If so, the key grab is terminated. Next, the `KeyRelease` event is sent to the grabbing client (which is the DS Event Deliverer in the case of a 3D grab) and finally the keyboard device's `DeactivateGrab` routine is called to deactivate the grab.

`CoreProcessPointerEvent` handles pointer grab terminations in a similar way, except that it is a `ButtonRelease` that terminates pointer grabs.

13.3 Informing the DS of Grab State Changes

So far we have described how 2D and 3D grabs are triggered and terminated, and how motion events are handled. But a careful reader of the `events.c` code will notice that there is some other new LG-specific code which we have not yet discussed. In several places in the code there are calls to a routine called `informDSOfDeviceGrabStateChange`. In this section we will describe the purpose of these calls.

Because the DS `LowLevelX11Picker` only performs a pre-test for 3D grab activation the DS doesn't know whether an event that **might** trigger a 3D grab really **will** trigger a 3D grab. The only component that knows when a 3D grab is triggered is the XS. So the XS must inform the DS when the state of a 3D grab for a particular type of device triggers or terminates. That is two say, the XS must inform the DS when the grab state changes.

A simple mechanism is used for the XS to inform the DS of such changes. The `child` field of 3D events is not being used to convey any other information. We simply send the information about the grab state change to the DS in the `child` field of the next outward bound 3D event. Any type of event will do, even motion events, which ordinarily have nothing to do with grab triggering and termination.

Note: The astute reader will wonder why a grab state change will ever occur when there isn't a correponding key or button event so that the state change needs to be sent to the DS in a motion event. The answer to this is that there is another type of grab, the *active grab*, which we will decribe in the next section.

The routine `informDSOfDeviceGrabStateChange` is called whenever the grab state can potentially change. Specifically, it is called from the routines

1 There is another type of grab called an *active grab* which will be discussed further on.

`CoreProcessKeyboardEvent`, `CoreProcessPointerEvent` and `Activate3DPassiveGrab`. But before we describe in detail what this routine does, we must discuss another type of grab which the XS supports, the active grab.

Note: the DS only cares about state changes of grabs caused by 3D events. It needs to know this so it can properly route the 3D events that are sent to it. If a 2D grab triggers events will just stop coming to the DS for the lifetime of the grab. So there will be no events for the DS to forward along to clients.

13.4 Handling Active Grabs

In the foregoing we have been discussing type of grab called a *passive grab*, which only triggers and terminates when a certain type of event arrives from an input device. The XS also supports another type of grab called an *active grab*. This is a grab which is triggered by an explicit request from some client. This request arrives in the XS asynchronous to the event stream. The LG-modified event pipeline has been carefully designed to work properly with both passive and active grabs. An active grab is similar to a passive grab: while the grab is active all events are sent to the grabbing client. The only difference between an active grab and a passive grab is the way in which the grab is triggered.

An active grab of the keyboard is triggered via the `GrabKeyboard` request. It is terminated by the `UngrabKeyboard` request. An active grab of the pointer is triggered via the `GrabPointer` request. It is terminated by the `UngrabPointer` request.

In addition to notifying the DS whenever a passive grab triggers or terminates, the XS must also notify the DS whenever an active grab triggers or terminates. LG-specific code has been added to the four aforementioned routines. If any one of these routines is called, the global flag `lg3dNotifyActivePointerGrabStateChange` is set to true if the client is the DS. Note that it is not necessary to send the active grab state change notification to the DS immediately, in asynchronous way; grabs only determine how events are processed, so it is sufficient to notify the DS when the next 3D event is to be sent to the DS. Also, it is not necessary to notify the DS of 2D grab changes (active or passive); the events will simply stop flowing to the DS and it doesn't need to know.

When `lg3dNotifyActivePointerGrabStateChange` is true it means some sort of 3D grab state change has occurred and the DS must be notified. Before sending any type of 3D event to the DS, the XS checks this flag to see if there has been a 3D grab state change which has not yet been communicated to the DS. If so, the `child` field of the 3D event is updated accordingly. The logic for how this field is updated is contained in the routine `informDSOfDeviceGrabStateChange`.

`informDSOfDeviceGrabStateChange` behaves as follows. If a 3D grab has recently been activated and the DS has not yet been informed of the change, the `child` field of the 3D event is set to `DELIVERY_GRAB_STATE_ACTIVE_GRAB_ENABLED`. (Note: that this routine will only be called for active grab changes. The information that a passive grab has activated is conveyed in `Activate3DPassiveGrab`; see Section

13.1.2). If, on the other hand, a 3D grab has been terminated and the DS has not yet been notified, the `child` field of the 3D event is set to `DELIVERY_GRAB_STATE_GRAB_TERMINATED`. And if there has been no change of interest to the DS, the `child` field is set to `DELIVERY_GRAB_STATE_NO_CHANGE`.

We will see how the DS uses this information in Section 21.

The above logic depends on the XS knowing what it last told the DS. To this end, whenever we update the `child` field of a 3D event, we save the value in the global variable `lg3dDeliveryGrabStateLastSent`.

14 XS: Event Distribution to Interested Clients

When `CoreProcessKeyboardEvent` and `CoreProcessPointerEvent` have finished processing the event, they call one of three XS *high-level* event delivery routines to deliver the event to clients.

14.1 High-Level Delivery Routines

This section describes the LG changes that have been made to the high-level event delivery routines. These are routines that are called directly by `CoreProcessKeyboardEvent` and `CoreProcessPointerEvent`, and others.

These three high-level routines call lower level routines to perform some of their work and to eventually send the event to interested clients. Some of these lower level routines needed to be changed in order for LG to work properly. These changes are described in Section 14.2.

14.1.1 DeliverGrabbedEvent

This is called for both key and pointer events to deliver the event to the grabbing client.

LG has not changed the behavior of this routine. It functions as it always did.

14.1.2 DeliverFocusedEvent

This is called for key events to deliver the event to clients who have expressed interest in events from the current focus window, or some window upward in the window tree.

LG has not changed the behavior of this routine. It functions as it always did.

14.1.3 DeliverDeviceEvents

This is called for pointer events to deliver the event to clients who have expressed interest in events from the current sprite window, or some window upward in the window tree.

Ordinarily, `DeliverDeviceEvents` delivers its argument events to the current sprite window. The problem is that sometimes LG window avatars are in the process of being animated and the XS is not informed of their new position until the animation stops, so XS's notion of the current window position, and hence the current sprite window, might be stale. If a button event occurs over a window avatar while it is being animated, the event might end up getting sent to the wrong window.
Note that this is only a problem for button events—key events are not sent to the current sprite window and motion events have already had the sprite window updated in `CheckMotion`.

To handle the case of button events, LG has modified the event pipeline to call a special routine called `lg3dDeliverDeviceEvents`. This routine is called instead of `DeliverDeviceEvents` at all call sites. The modified routine handles 2D button events specially: instead of sending these events to the current sprite window, they are sent to the window which was determined by the `XYToSubWindow` call. That is to say, they must be sent to a window which is a subwindow of the avatar that the DS picked.

In addition to the above change, `DeliverDeviceEvents` has been modified in another way. Ordinarily, if no client has expressed event interest for the target window (i.e. the current sprite window for a pointer event or the current focus window for a key event), the XS will check the parent of the target window to see if some client has expressed event interest for this window. If no client has expressed interest in the parent window, the parent of the parent window is checked. Ordinarily this checking propagates up the window tree until it hits the root window. But in LG, all windows are descendents of the PRW. So if a 2D event occurs in a 2D window and no client has expressed event interest upward through the window tree, `DeliverDeviceEvents` will eventually come to the PRW and will want to deliver the event to the DS (because the DS expresses interest in all types of events on the PRW). But sending a 2D event to the DS will wreak havoc, because there is no associated 3D Event Information for the event; no such information was saved by the `LowLevelX11Picker` because the event was determined to be a 2D event. To avoid this problem, `DeliverDeviceEvents` has been altered to stop the upward propogation at top-level X windows (i.e. direct children of the PRW).

14.2 Other Delivery Routines

This section describes lower level routines that are called by the high-level event delivery routines which were described in the previous section.

14.2.1 TryClientEvents

This routine is one of the main low-level routines which is used to send events to clients. It ordinarily sets the event sequence number field to the current sequence number of the client. But 3D events already have significant information in this field, namely, the pick sequence numbers which are used by the DS to reassociate the 3D event information with the event. So this routine has been modified to not override the event sequence number

field for 3D events.

14.2.2 FixupEventFromWindow

This routine is called to update an event's coordinates based on the destination window. Other event fields are updated as well. Ordinarily this routine calculates the `root` and `child` field of the events, sets the event window to the destination window which has been chosen by XS, and calculates the event's window relative coordinates by transforming by the event's screen absolute coordinates into the coordinate space of the destination window.

The next two subsections describe how LG changes this routine.

14.2.2.1 Fixing up 2D Events

Key event fields no longer need to be recalculated; all the fields of key events are already correct at this point. For pointer events, if the destination window is the same as the window currently set in the event, then no recalculation of fields is necessary--the information is already correct. (This might happen, for example, if the cursor is over a top-level window which has no subwindows.

If the destination window is different from the event window, the event's window relative coordinates (which were calculated by the DS and which are relative to the top-level window) need to be transformed into the coordinate space of the destination window. This happens when the DS has set the event window to a top-level window and `XYToSubwindow` has determined the destination window to be a proper subwindow of that top-level window.

Note: in order to perform this coordinate transformation, the screen absolute coordinates of the top-level window need to be determined. If the top-level window has been destroyed since the pick was performed, these coordinates will not be available. So in this case simply we skip the transformation. Since the top-level window has disappeared, all of its subwindows will have disappeared and the event will be undeliverable anyway.

Note: there is one possible problem: if a window is reparented out of a top-level window into another window after the pick has been performed but before `FixupEventFromWindow` is called, the transformation will be incorrect. This is an issue that needs to be addressed but it is currently a low priority because applications typically do not reparent windows in this way.

It should also be pointed out that even if the event is a 3D event, and the DS has determined that it should be sent back to the DS after its trip through the XS, it is possible that a 2D grab may be active and therefore the target window is the normal X11 window of the grabbing client. The modified `FixupEventFromWindow` handles this properly.

14.2.2.2 Fixing up 3D Events

For a 3D event, which is supposed to be sent back to the DS, the only event field which need to be calculated is the `root` field. But if the event is a 2D event which is being sent to the DS because of a 3D grab, the window relative coordinates of the event need to be set to the coordinates of the 2D event relative to the PRW. These coordinates are simply the screen absolute coordinates of the 2D event.

14.2.3 DeliverEventsToWindow

This routine is called by in several places in XS. It is called by `DeliverDeviceEvents`, `DeliverFocusedEvent`, and `EnterLeaveEvent` (to deliver XS-generated window `EnterNotify` and `LeaveNotify` events). It is also used to deliver other XS-generated events such as focus, property, and window change events, and also to deliver client-generated events.

This routine implements the default button grab check for 2D events. If a grab is not already active and the event type is `ButtonPress`, a grab is triggered.

Unfortunately, mouse wheel events are represented as `ButtonPress` followed by a `ButtonRelease` events in the XS. This means that, in ordinary circumstances, moving the mouse wheel will cause a series of grab triggers and terminations. This appears to be a bug in the XS. So it has been fixed when LG is active. This has been reported as Xorg bug 5327.

14.3 Event Delivery Summary

Eventually an event will percolate through all of the XS event delivery routines and `WriteEventsToClient` will be called to actually write the event to all interested clients. There are several cases:

- A 2D event whose pick operation hit an X window avatar is sent to all X11 clients which have expressed interest in receiving that type of event for that window.
- If a 2D grab is active, all events are sent to the grabbing client and are assigned the grab window, until the grab is released. All of these events are considered to be 2D events.
- A 3D event whose pick operation hit an LG3D application 3D object, or hit nothing, is sent to the DS Event Deliverer client.
- If a 3D grab is active, all events are sent to the DS Event Deliverer client until the 3D grab is released.

From here on out, 2D events are distributed in the standard way by the XS to clients. The rest of this document will continue to follow the path of 3D events.

15 DS: Events Return to the DS

This section deals with how the DS processes 3D events after they have been processed by the XS. They have finally escaped the hairball of XS event processing and are back in the DS. (This is the upper horizontal segment of the Z path).

The `CookedEventPoller` class implements the lower part of the DS Event Distributor. `CookedEventPoller.run` is a thread which continually loops reading events from the XS. The events received by `LowLevelX11Picker` were raw (unprocessed) device events. But the events received by the `CookedEventPoller` have received a substantial amount of processing by XS, hence the term *cooked* event.

Cooked events which are received from the XS are converted into equivalent `AWTEvent`s, which is the type of event class that Java apps know how to deal with. A single cooked event may generate one or more `AWTEvent`s.

Note: these `AWTEvent`s are special in that they also contain the pick sequence number. Special subclasses of `AWTEvent`, namely `SequencedKeyEvent`, `SequencedMouseEvent`, and `SequencedMouseWheelEvent` are used for this purpose.

For the most part, there is a one-to-one mapping between the bits of information in an X11 event and the bits in an AWT event. However, there is some special processing which happens.

15.1 Motion Events

If any of the mouse button events are pressed while the motion was being made, the event type is set to `MouseEvent.MOUSE_DRAGGED`. Otherwise the event type is set to `MouseEvent.MOUSE_MOVED`.

15.2 Mouse Wheel Events

Mouse wheel events, which arrive from the XS appearing as `ButtonPress` events of the mouse buttons 4 and 5 (depending on the wheel movement direction), are converted into events of type `MouseEvent.MOUSE_WHEEL`.

15.3 Button Events

A check is made for multiple button clicks. Pressing a button starts a *multi-click* detector: the button number, time, location, and window of the `ButtonPress` is recorded. Subsequent `ButtonPress` and `ButtonRelease` events that match the button number, location[1] and window values and which have a time interval which is within the multi-click timeout (a configurable value) are considered to be a part of the multi-click event sequence. Each event has a

1 The location match has a 3 pixel tolerance.

multi-click counter field which indicates its position in the multi-click sequence. In addition, after the final `ButtonRelease` of a multi-click sequence, a special `MOUSE_CLICKED` event is generated (as per Java AWT event semantics).

15.4 Key Events

The key event processing performed by `CookedEventPoller` is extremely complex. This is because the mapping of key codes to X *keysyms* is extremely complex. The processing is performed by `nativeHandleKeyEvent`, a C native routine. The X key event is passed to this routine and one more key events may be generated and sent along the event pipeline. The actual processing performed by this routine is far too complex to be described in this document.

`nativeHandleKeyEvent` generates key events by invoking the `CookedEventPoller.enqueueKeyEvent` method. This method converts the X event into an equivalent `AWTEvent` and adds it to a global list called `keyEventList`. These events are sent along the event pipeline.

16 DS: Upper Part of the Event Deliverer

Once `CookedEventPoller` has converted an incoming X event into one or more `AWTEvent`s, `CookedEventPoller.processEvent3D` is called to hand off the event to the upper part of the DS Event Deliverer[1]. This module is responsible for performing 3D grab processing, determining the destination scene graph object(s) for the event, and forwarding the event along to the LG EventProcessor. From there the event will be delivered to LG clients.

The upper part of the DS Event Deliverer is implemented by the class `PickEngineX11`[2]. This class inherits significant functionality from the `PickEngine` class.

`CookedEventPoller.process3D` (eventually) adds the events generated by the `CookedEventPoller` to the end of `PickEngineX11`'s input queue. (`PickEngineX11` has double-buffered input queues: events are added to one queue while the events in the other queue are being processed, then the roles of the queues are swapped).

NOTE: motion compression is performed on `PickEngineX11`'s input queue. This means that consecutive motion events are collapsed into one event. Motion compression on this queue is critical for good interactive performance in LG.

`PickEngineX11` is a Java3D behavior which is run once per rendering frame. Each time it is run, it processes all of the events in its input queue. It calls `processAWTEvent` to do this. This is a polymorphic method which takes either an AWT `KeyEvent` object or a `MouseEvent` object. We will first describe the `KeyEvent` case because it is the simplest. But first we need to describe LG *Event*

1 At the time of this writing, the event travels through an intermediate object of class `InputEventBroker`, but this intermediate is gratuitous and will removed in the future.
2 `PickEngineX11` does perform picking, but only in infrequent cases. The name is historical and should probably be changed.

Sources.

17 DS: LG Event Sources

In LG, a scene graph object which is sensitive to events is called an *event source*. LG3D applications can register interest in certain types of events which are associated with certain 3D objects. These objects are considered to be event sources. LG3D applications can register interest in the events of specific event sources or any event source. From the application's point of view, the events "come from" the event source objects.

It is the Event Distributor's job to send its incoming events to appropriate event source objects. So, from the perspective of the Event Distributor, an event source is actually an event "destination", but we will always call them event sources to be consistent. The Event Distributor may send an event to multiple event sources.

18 DS: Distributing 3D Key Events

`PickEngineX11.processAWTEvent` for key events uses a list called `nodePathForKeyEvent`. This is calculated based on the *keyboard focus node* as specified by the LG Scene Manager. The Scene Manager can use any policy (e.g. click-to-type or follows-mouse) to determine this node[1]. If the Scene Manager doesn't specify a node, the default policy implemented by the Event Distributor is follows-mouse.

`nodePathForKeyEvent` is a list of nodes. The first node is the keyboard focus node, the second node is its parent, the third node is its grandparent, and so on, up to the root node of the scene graph. (This is similar to the `focus->trace` array of XS). `processAWTEvent` for key events iterates over this list and chooses the first node which is an event source for key events.

Finally, the event is converted into the LG key event type, `KeyEvent3D` and it is sent to the LG `EventProcessor` by calling `EventProcessor.postEvent`. The `EventProcessor` will be responsible for invoking various LG action methods in response to this event.

19 DS: Distributing 3D Mouse Events

The Event Distributor's processing of mouse events is far more involved than its processing of key events.

19.1 EventInfo3D Matching and Grab Processing

The basic approach of `PickEngineX11.processAWTEvent` for mouse events is to first reunite the event with its corresponding `EventInfo3D` object and then figure out what is the destination node. It does this by calling `determineEventDestInfo`. This information is stored in an instance of `PointerObjectX11`. This object stores everything that is known about a state of the pointer device. `pointerObjCur` is a variable which refers to the current state of

1 Java3D uses the term *node* to refer to a 3D object in the scene graph. We follow this terminology here.

the pointer. `pointerObjPrev` refers to the state of the pointer as of the previous event.

`determineEventDestInfo` also takes grabs into account. This will be described further in Section 21.

19.2 Calculate the Path from the Root Node

Next, the path from the root node of the scene graph down to the hit node is calculated (only pickable nodes are considered). This is called `PointerObjectX11.pathFromRoot`. (This is similar to the `spriteTrace` array in XS).

19.3 Deliver Event Based on Node Information

Then the event is delivered to the event processor based on the all of the information about the hit node that has been collected so far.

In the next sections, we will examine these steps in greater detail.

20 DS: EventInfo3D Matching

`PickEngineX11.determineEventDestInfo` is called to match the event to its corresponding `EventInfo3D` object and to determine the destination node of the event. Note that this may be different than the hit node of the event, especially if a 3D grab is active.

First of all, button clicked events must be handled specially. Button click events are always generated after a button release event. But button release events can terminate a grab. We always want to send the button clicked event to the node to which we sent the button release event. So we keep track of the destination node to which we last sent a button release event and send a mouse clicked event to that node.

Next, we check to see if there is an active 3D grab. `activeGrabState` is a variable which stores all of the information about any grab which may be active.

After that, we figure out whether the event might have a corresponding `EventInfo3D`. Button press events that activate a 3D passive grab will always have an `EventInfo3D`. But that may not be true for other events, depending on whether a 3D grab is active. If a 3D grab is not active then any events arriving in the Event Deliverer will have a corresponding `EventInfo3D`. But if a 3D grab is active, some of the events arriving in the Event Distributor may actually be 2D events which were hijacked by the 3D grab, and these events will not have an `EventInfo3D`!

If the event might have an `EventInfo3D`, `determineEvinfo` is called. First, the pick sequence number of the event is determined. For most types of events, the pick sequence number is a positive integer. But window enter and leave events (and also key clicked events) have a pick sequence number of zero. This is because they were generated by later stages of the event processing pipeline and were never sent to the DS for picking. Therefore these types of events have no associated `EventInfo3D`.

So, in these cases, the `EventInfo3D` of the previous event is used instead[1].

For events whose pick sequence number is a positive integer, the list of events is searched for the `EventInfo3D` which has the corresponding pick sequence number, starting at the beginning of the list. As the search progresses, if we find any `EventInfo3D`s which have pick sequence numbers less than that of the current event we discard them. That is because the information in these objects is obsolete and is no longer needed. For example, this might happen if the DS picker sent a 3D event back to the XS but this event was hijacked by a 2D grab and sent to a conventional X11 application.

The `EventInfo3D` search is terminated either when the desired `EventInfo3D` is found. (If, before finding the desired object, an object were to be found which has a higher pick sequence number, it would be a fatal error condition; it would indicate a bug in the LG event pipeline code).

21 DS: 3D Grab Processing

After it has found the `EventInfo3D` which corresponds to the event (or determined that there isn't one), `determineEventDestInfo` performs grab processing.

First, the event is checked to see if it contains any information that the XS's grab state has changed.

If the `child` field of the event indicates whether a 3D passive grab has been activated, the specific information about the grab is determined from the `EventInfo3D` (it was calculated by the DS Picker). Then the `ActiveGrabState.grabEnable` method is called to put the pointer device into the grabbed state.

If the `child` field of the event indicates that a 3D grab has been terminated we simply record this information for later use.

(NOTE: the code supports a third possible state of the event `child` field, namely, that a 3D active grab has been enabled. The DS contains code to allow LG3D clients to actively grab input devices. But the interface to this code has not yet been publically exported to LG3D application developers, for the same reason as described in Section 2. So we will ignore this case here).

Back in `determineEventDestInfo`, if no 3D grab is active, we call `determineEventDestInfoNotGrabbed` to figure out the destination node of the event. The operation of this routine is further described in the next section. But if a 3D grab is active, we determine the destination node based on information contained in the EventInfo3D by calling `determineEventDestInfoGrabbed`. For a button press event we use the pick result that was closest to the eye (determined during DS picking). From this we calculate the intersection point (in object local coordinates) of the pick ray with the object's geometry. We also calculate the intersection point in virtual world coordinates. And we record the distance from the eye to the intersection point. We record all of this information in a `MouseEventNodeInfo` object (aka *node info*). This information is

1 When certain LG animations finish, a motion event with a special pick sequence number of -1 is generated to resynchronize the current pointer object to the current state of the scene graph. There is no corresponding `EventInfo3D` for this type of event. So a pick operation will be performed.

only calculated for button press events. All other events will use the node info of the button press event which triggered the grab.

After `determineEventDestInfo` has computed the destination node and its corresponding node info for a grabbed event, it is finally safe to check whether the event terminates the grab. Earlier we recorded whether the grab had been terminated. If this is the case we call `ActiveGrabState.grabDisable` to deactivate the grab.

22 DS: The Destination of Ungrabbed Events

The previous section described how the destination node is determined for events while there is an active grab. This section will describe how the destination node is determined when there is no active grab. `determineEventDestInfoNotGrabbed` is called to figure this out.

First this routine determines whether it can get the information it needs from the event's `EventInfo3D`, or whether it needs to perform a pick itself to determine this information. Any event which has come this far for which the Event Deliverer cannot find any valid `EventInfo3D` is going to need to have some generated for it. This will be the case for window enter and leave events which were generated by XS (or for key clicked events which were generated by `CookedEventPoller`).

`determineEventDestInfoNotGrabbed` performs a pick in much the same way as the `LowLevelX11Picker`[1]. From the resulting pick results we determine the node info (intersection point, eye distance, etc.) in the exact same way as in `determineEventDestInfoGrabbed`.

23 DS: Calculating Path-From-Root

Once the destination node and its associated node info has been determined, `determineEventDestInfo` then returns to `processAWTEvent` for mouse events. The next step in the event distribution process is to calculate the path from the root node of the scene graph to the destination node. This is a simple matter: we start at the destination node and traverse upward through the parent links and add the nodes we encounter to the **front** of the path node list. During this traversal we ignore nodes which are not pickable.

24 DS: Delivering Event Based on Node Info

After `determineEventDestInfo` has determined the node path, it calls `deliverEventUsingNodeInfo` to pass the event to the EventProcessor.

First of all, the event position is converted into virtual world coordinates. This will later be assigned to the `cursorPos` field of the event. (Note that we do not actually move the cursor here; we merely calculate its position).

Next, `deliverEnterExitEvents` is called. This routine sends the appropriate 3D object enter and

1 There is a Java3D bug which sometimes makes this pick fail. So in some cases, if we really, really need the information, we must try several times.

3D object exit events. If the current pointer object differs from the previous pointer object, all mouse event source scene graph nodes which the two objects do not share in common will be sent either an enter or exit event. The nodes in the path of the previous pointer object will be sent exit events and the nodes in the path of the current pointer object will be sent enter events.

Note: in this routine we also track the current keyboard focus object so we can implement a follow-mouse keyboard focus policy if the user hasn't explicitly specified a keyboard focus.

Finally, we deliver the event to the registered event sources for this type of event, based on the scene graph path, by calling `deliverMouseEventToSources`. This routine converts the `AWTEvents` into event objects which are subclasses of type `LgEvent` (`MouseMotionEvent3D`, `MouseButtonEvent3D`, etc.) These are the types of events which are used in the LG Client API. These events are then sent to all mouse event source scene graph nodes in the path from the scene graph root to the current pointer object. The sending is accomplished by calling `EventProcessor.postEvent`. This injects the event into the LG EventProcessor which will take care of delivering the event to interested listeners of the source objects.

At this point, the event has entered the realm of LG event handling and it is here that our journey ends.

25 Conclusion

An architecture and implementation for input redirection in a 3D window system has been presented. We have traced the path of an input device event from the time it is generated until the time it is distributed to either X11 clients or LG clients. Raw device events are read by XS and sent to DS. The DS uses picking to determine the 3D object associated with an event. If the object is a window avatar, the event is a 2D event, otherwise it is a 3D event. The event is sent back to to XS where it undergoes more event processing. For 2D motion events the lowest enclosing subwindow is calculated. 2D and 3D button and key events may trigger or terminate a grab. The DS is informed of any 3D grab triggers or terminations. 2D events are sent to conventional X11 applications and 3D events are sent back to the DS. For 3D events, the DS converts the X events into `AWTEvents`. Grab processing is performed on 3D key and button events, and all 3D events are delivered to the interested client listeners for various LG scene graph objects.

26 Appendix: XS Modification Summary

This section provides a summary of the sections in this document which describe the modifications that have been made to the Xorg server to support Looking Glass.

Section 4	XS: Sending Device Events to the DS
Section 9	XS: Receiving the Picked Event
Section 10	XS: Keyboard Event Processing
Section 11	XS: Button Event Processing
Section 12	XS: Motion Event Processing
Section 13	XS: Grab Processing
Section 14	XS: Event Distribution to Interested Clients

27 Acknowledgements

The following individuals provided invaluable assistance in the design of the LG event pipeline:

Paul Byrne
Amir Bukhari
Jim Gettys
James Gosling
Hideya Kawahara
Keith Packard

28 Bibliography

[1] *The X Window System Server* by Elias Israel and Erik Fortune, Digital Press.

[2] *X Window Sytem: the complete reference to Xlib, X Protocol, ICCCM, XLFD* by Robert W. Scheiffler, James Gettys; with Jim Flowers and David Rosenthal. 3rd ed. Digital Press, 1992. ISBN 1-55558-088-2.

[3] http://lg3d-core.dev.java.net
This is the home page for Project Looking Glass. A variety of information about LG is available at this site.

[4] http://lg3d-core.dev.java.net/files/documents/1834/5964/J1-2004-TS1586-final.pdf
This presentation provides a good overview of the overall goals and architecture of LG as well as explanatory high-level block diagrams.

[5] https://lg3d-core.dev.java.net/files/documents/1834/7596/LG3DAPIOverview-DRAFT.pdf
This document describes the LG Client API. Section 3.6 deals specifically with events.