Introducción a X-Windows[1]

(Manual de Programación de Xlib, O´Reilly, 1992, Capítulos 1, 2, y 3)

por

Dr. Alfredo Weitzenfeld
Departamento Académico de Computación
ITAM

[1] Este documento es una traducción y adaptación del documento original en inglés.

1. Introducción

X es un sistema de ventanas en red, pudiendo correr un programa en una computadora remota y desplegarse los resultados localmente.

1.1 Versiones de X

X se desarrolló conjuntamente por el Proyecto Athena en MIT y DEC, con contribuciones de otras compañías.
Fue diseñado por el grupo de Robert Scheifler en MIT, aunque inspirado por el paquete de ventanas “W” desarrollado por Paul Asente en Stanford.
La versión X10 R4 (Release 4) estuvo lista en 1986 siendo la primera versión en utilizarse comercialmente.
En Setiembre 1987 MIT produjo la primera versión de X11 R1.
En Marzo de 1988 se produjo X11R2.
En Febrero de 1989 se produjo X11R3.
En Enero de 1990 se produjo X11R4.
En Agosto de 1991 se produjo X11R5.
En 1995 se produjo X11R6, la última versión.

Con la versión X11R2 el control de X pasó en 1988 de MIT al X Consortium, una asociación de la mayoría de los fabricantes importantes de computadoras apoyando el estándar X.
Muchas casas de software al igual que universidades son miembros asociados.

La biblioteca de subrutinas de X11 es Xlib.

1.2 Conceptos del Sistema de X-Windows

El sistema de X Windows es complejo, aunque basado en algunos conceptos básicos que pueden comprenderse.

1.2.1 Despliegues y Pantallas (Displays and Screens)

X es un sistema para despliegues gráficos con **bitmaps** (en gráficos bitmaps o **rasters**, cada punto en la pantalla, o **pixel**, corresponde a uno o más bits en la memoria). Los programas modifican el despliegue simplemente escribiendo a la memoria de despliegue.
Se apoya despliegues a color, monocromo y escalas de gris (gray-scale).

Un concepto peculiar es que un ***despliegue*** (***display***) se define como una estación de trabajo consistiendo de un teclado, un dispositivo para apuntar tal como un ratón, y *una* o *más* ***pantallas*** (***screens***).
Múltiples pantallas pueden trabajar juntas, permitiendo que el movimiento del ratón cruce los bordes físicos de las pantallas.
Mientras las múltiples pantallas se controlen por un sólo usuario con un sólo teclado y dispositivo de apuntar, ástas definen un sólo despliegue.

1.2.2 El Modelo Servidor-Cliente

El programa que controla cada despliegue es el ***servidor*** (***server***).
Esta utilización del término servidor es diferente de cuando se piensa que una máquina en la red es un servidor, en lugar de un programa local controlando el despliegue local.
Aquí se refiere a que el despliegue local es accesible a otros sistemas en la red, y para esos sistemas, el código ejecuta remotamente actúa como un servidor de despliegue.

El despliegue actúa como intermediario entre los programas de usuario (llamados ***clientes*** o ***aplicaciones***) ejecutando en sistemas locales o remotos y los recursos del sistema local.
El servidor básico es responsable de las siguientes tareas:

- Permitir acceso de múltiples clientes al despliegue.
- Interpretar mensajes de red de los clientes.
- Pasar entradas de usuario a los clientes enviando mensajes de red.
- Hace dibujo de dos dimensiones. Los gráficos los ejecuta el servidor de despliegue y no el cliente.
- Mantener estructura de datos complejas, incluyendo ventanas, cursores, fonts, y “contextos gráficos”, como ***recursos*** (***resources***) que pueden ser compartidos entre clientes y referidos simplemente por IDs de recursos.

1.2.3 Administración de Ventana (Window Management)

La organización y apariencia de la pantalla y el estilo de interacción de usuario con el sistema son definidas por el ***administrador de ventana*** (***window manager***).

Este programa esta también escrito con Xlib, excepto que se le da permiso especial para controlar la política de organización de las ventanas en la pantalla, incluyendo la ley de traslape, *click-to-type*, etc.

twm es el manejador estándar de X-Windows, aunque existen muchos otros.
olwm es el manejador de Open Look de Sun
mwm es el manejador de Motif de OSF (Open Software Foundation).

1.2.4 Eventos

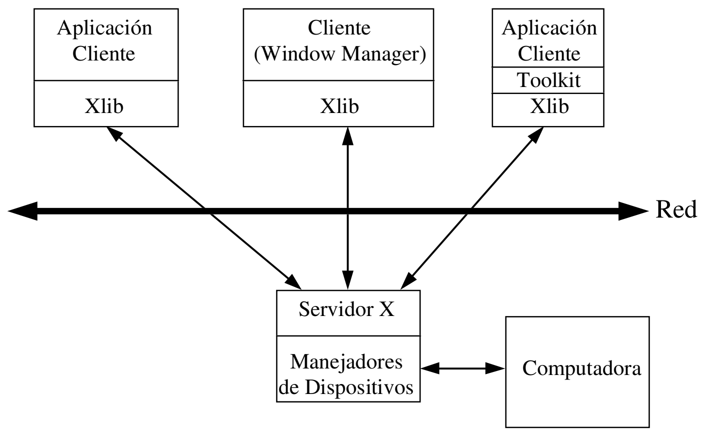
Como cualquier sistema de ventanas manejado por el ratón, un cliente de X debe poder responder a los muchos diferentes ***eventos***, como presión de una tecla, click del ratón, movimiento del ratón, e inclusive interacción con otros programas (e.g., si una porción oculta de una ventana se expone cuando otra ventana traslapada se mueve, cierra, o cambia de tamaño, el cliente debe redibujarla).
Eventos pueden ser de muchos tipos y ocurrir en cualquier momento y en cualquier orden.
Estos se ponen en una cola en el orden en que ocurren para ser procesadas por el cliente en ese orden.
La necesidad de manejar eventos es diferente a programas tradicionales sin ventanas, donde se utiliza funciones en C para leer caracteres y se hace un polling de las entradas.
En constraste, en un sistema con ventanas se utilizan funciones de espera de eventos.

1.2.5 Extensiones a X

Extensiones abiertas en X incluye extensiones gráficas para 3D, llamada PEX, con sus dos APIs (Application Programming Interface) llamados PHIGS y PEXlib.

Un vendedor del servidor no requiere proveer apoyo a todas las extensiones estándares. Por lo tanto, antes de utilizar una extensión, se debe preguntar al servidor si la extensión es apoyada.

1.3 Arquitectura de Software para el Sistema X-Windows



Un servidor de despliegue es un programa que corre en cada sistema que apoya el despliegue gráfico, teclado y ratón.
Existen servidores de X para virtualmente todas las estaciones de trabajo, además de versiones para PCs.

Las aplicaciones se comunican con el servidor por medio de llamadas a la biblioteca de bajo nivel de rutinas en C, llamada *Xlib*, la cual provee funciones para conectarse a un servidor de despliegue particular, crear ventanas, dibujar gráficas, responder a eventos, etc.
Las llamadas de Xlib se traducen en solicitudes de protocolo enviadas a través de TCP/IP al servidor local u otro servidor remoto.

Toolkits implementan un conjunto de objetos, como menús y botones (referidos como *widgets*) y permiten a la aplicación manejar estos objetos para facilitar la construcción de sistemas de interfaces.

2. Conceptos de X

2.1 Xlib

2.1.1 El Protocolo X

El protocolo X especifica como se compone cada paquete de información que se transfiere entre el servidor y Xlib en ambas direcciones.

Inclusive cuando el servidor y Xlib se ejecutan en la misma máquina, se utiliza el protocolo para comunicación, utilizando un canal interno en lugar de la red externa.
Hay cuatro tipos de paquetes que se transfieren por medio del protocolo:

- **Solicitudes (Requests)**
- **Respuestas (Replies)**
- **Eventos**
- **Errores**

Una *solicitud* del protocolo se genera por Xlib y se envía al servidor, y puede llevar una gran variedad de información, tal como una especificación para dibujar una línea o cambiar el valor del color en una celda (cell) en el mapa de colores (colormap) o una consulta sobre el tamaño actual de la ventana.
La mayoría de las rutinas de Xlib generan solicitudes del protocolo.
Las excepciones son rutinas que sólo afectan estructuras de datos locales a Xlib y no afectan al servidor.

Una *respuesta* del protocolo se envía del servidor a Xlib en respuesta a ciertas solicitudes.
No todas las solicitudes se responden por medio de respuestas, sólo las que solicitan información.
Solicitudes que especifican dibujar, por ejemplo, no generan respuestas.
Cuando Xlib recibe una respuesta, guarda los datos solicitados en los argumentos o valor devuelto de la rutina de Xlib que generó la solicitud.
Una rutina de Xlib que requiere una respuesta se llama una *solicitud de viaje-redondo* (*round-trip request*). Estos se deben minimizar en los clientes ya que empeoran el rendimiento cuando hay retardos de la red.

Un *evento* se envía del servidor a Xlib y contiene información sobre una acción de un dispositivo o sobre un efecto secundario de una solicitud previa.
Los datos contenidos en los eventos varia.
Los eventos se mantienen en una cola en Xlib y pueden ser leídos uno a la vez por el cliente.

Un *error* le dice al cliente que una solicitud previa no ha sido válida.
Un error es como un evento, pero se maneja un poco diferente dentro de Xlib.
Errores no pueden ser leídos por llamadas a Xlib que leen eventos, en su lugar, los errores se envían a una rutina de Xlib para el manejo de errores.
El manejador de omisión simplemente imprime una mensaje y sale, pudiendo cambiarse el manejador por una rutina del cliente.

2.1.2 Buffering

Xlib guarda las solicitudes en lugar de enviarlas al servidor inmediatamente, para que el programa del cliente pueda continuar ejecutando en lugar de esperar obtener acceso a la red después de cada llamada a Xlib.

Xlib envía el buffer lleno de solicitudes al servidor bajo tres condiciones.

- Llamadas de Xlib que reciben información del servidor requieren una respuesta antes de que el programa pueda continuar, i.e. una rutina con **Query**, **Fetch** o **Get** en el nombre.
- La más común es cuando una aplicación llama una rutina de Xlib para esperar un evento sin que exista un evento correspondiente en la cola de Xlib.
- El cliente quisiera poder explicitamente enviar el buffer manualmente (flush) en situaciones diferentes a las dos anteriores, por ejemplo, en una animación donde el despliegue cambia constantemente, i.e. las rutinas **XFlush()** o **XSync()**. (No se usa mucho esto, ya que existen suficientes solicitudes de los dos tipos anteriores, haciendo el "flush" frecuente.)

Cuando la aplicación comienza, todas las solicitudes iniciales se ponen en la cola de Xlib.
Entonces la aplicación entra en el ciclo de eventos (event loop) y llama **XNextEvent().**
Como nada ha sido enviado todavía al servidor, no existen ventanas y por lo tanto no se han generado aún eventos.
Por lo cual **XNextEvent()** causa que todas las solicitudes se envíen al servidor, desplegando la aplicación.
Mientras tanto la aplicación está aun esperando la primera entrada del usuario.
Cuando el usuario mueve el ratón o presiona un botón o tecla, el servidor envía el evento a Xlib de forma inmediata.
Una vez que la aplicación vuelve en el ciclo a **XNextEvent()** y no hay más eventos en la cola, las solicitudes de la cola se envían al servidor y el proceso se repite.

Existe una cola de eventos en Xlib para cada servidor al cual la aplicación está conectada.
Así mismo, se mantiene una cola de salida de solicitudes para cada servidor al cual la aplicación este conectada.

2.1.3 Recursos (Resources)

X utiliza varias técnicas para reducir el tráfico en la red.
Una técnica importante es que el servidor mantenga abstracciones complejas, tal como ventanas o fonts, y que el cliente asigne un número entero como ID para cada una de las abstracciones, llamada *recurso*.
Un recurso puede ser una ventana, pixmap, colormap, cursos, font, o contexto gráfico.

Cuando se ejecuta una operación en una ventana (o cualquier otro recurso), el ID de la ventana se usa en un argumento a la rutina de Xlib, en lugar de enviar la estructura completa. (No se puede utilizar apuntadores a estructuras, ya que el cliente y el servidor pueden estar en máquinas diferentes.)
El único problema es que el cliente debe consultar al servidor cuando necesita información sobre recursos, lo que no es muy a menudo.

Si un cliente conoce el ID de un recurso, el cliente puede manipular el recurso inclusive si otro cliente ha creado el recurso.
Esto significa que más de un cliente puede dibujar en la misma ventana, aunque a menudo esto no es deseable.

Esta es la forma como los administradores de ventanas se implementan, pudiendo mover y cambiar los tamaños de las ventanas de las aplicaciones ya que conocen los IDs.

Existe otra utilización del término “recurso” en X que tiene que ver con el administrador de recursos.
En este contexto, un recurso es una especificación de preferencias de usuario que controla elementos ajustables (customizable) por el usuario ajustados a la aplicación.

2.1.4 Propiedades y Atomos

Una *propiedad* es un paquete de información asociado con una ventana, accesible a todos los clientes ejecutando bajo un servidor.
Las propiedades se usan por los clientes para guardar información que otros clientes puedan necesitar o quieran conocer y para leer esa información de otros clientes.

Las propiedades tienen un nombre (una cadena) y un ID numérico llamado un átomo.
Un *átomo* es un ID que identifica de forma única una propiedad particular.
Nombres de propiedades son tipicamente en letras mayúsculas, con palabras separadas por líneas de subrayar, como “WM_COLORMAP_WINDOW”.
Los átomos se usan para referirse a propiedades en llamadas a rutinas para no tener que enviar cadenas de longitud variable en la red.
Una aplicación recibe el átomo para una propiedad llamando **XInternAtom(***nombre-propiedad***)**.
Cada aplicación recibirá el mismo átomo para la misma propiedad si está conectado al mismo servidor (que no ha sido restablecido).

Algunos átomos, llamados *átomos predefinidos*, se definen cuando el servidor se inicializa.
Estos átomos son constantes simbólicas comenzando con **XA_**, e.g. **XA_WM_HINTS**, (también corresponden a propiedades).
Estos átomos identifican propiedades cuyo contenido tiene un significado especial conocido por convención por todos los clientes.
Las propiedades como tales no tienen contenido hasta que un cliente o administrador de ventanas los asigne.

Un grupo de clientes relacionados o una extensión pueden definir otras propiedades y átomos que tendrán un significado conocido a todos los clientes en el grupo o usando la extensión.

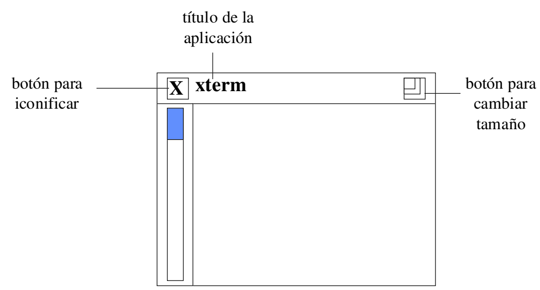
Para identificar una propiedad particular de una ventana se requiere el átomo y la ventana particular.
Sólo el tipo **Atom** se utiliza en el código del cliente, las propiedades se manejan internamente como datos por el servidor.

Uno de los usos más importantes de propiedades es para comunicar información de las aplicaciones al administrador de ventanas y vice-versa.

2.1.5 Administrador de Ventanas (Window Manager)

El administrador de ventanas es simplemente otro cliente escrito con Xlib, pero con responsabilidades especiales.
Atiende a demandas para recursos físicos de despliegue, incluyendo espacio de pantalla y colormap.
Tiene por lo general una interfaz que permite al usuario mover ventanas en la pantalla, cambiar su tamaño, e iniciar nuevas aplicaciones.

La mayoría de los administradores de ventanas decoran las ventanas principales de todas las aplicaciones con una barra de etiqueta y varias herramientas para iconificar y cambiar los tamaños de la aplicación.



Mucha de la comunicación entre clientes y el administrador de ventanas y viceversa ocurre por medio de propiedades (el resto por medio de eventos).
Muchas de las propiedades se conocen como ***hints*** (***insinuaciones***) ya que no necesariamente serán consideradas por el administrador de ventana, inclusive si uno está ejecutando.
Una aplicación debe estar preparada en caso de que el administrador de ventana decida ignorar, modificar, o considerar las preferencias indicadas por medio de los “hints” del administrador de ventanas.
Las propiedades por sí mismas no tienen contenidos válidos hasta que las aplicaciones o el administrador de ventanas las asigne.
Algunas de las características de Xlib (y el protocolo X) existen sólo para dar al administrador de ventanas el mecanismo para hacer cumplir su autoridad.

Una de esas características es ***redirección de subestructura***, refiriéndose al tamaño, posición y orden de traslape de los hijos de una ventana.
Redirección significa que solicitudes por la aplicación para cambiar las configuraciones de las ventanas son enviadas al administrador de ventanas para su aprobación antes de enviarse al servidor.
Redirección de subestructura permite que el administrador de ventanas intercepte cualquier solicitud de una aplicación para modificar la *configuración de la ventana*, o sea, cambiar el tamaño, posición, ancho del borde, orden de superposición de las ventanas de alto nivel en la pantalla.
Cualquier solicitud por una aplicación para cambiar la configuración de su ventana de alto nivel será cancelada, y en su lugar, un evento será enviado al administrador de ventanas indicando los argumentos utilizados en la solicitud de reconfiguración.

El administrador de ventana puede entonces decidir que aspectos de la reconfiguración aceptará hacer.
Para ventanas temporarias como menús pop-up y cajas de diálogos, la redirección de subestructura se puede desactivar usando un atributo de la ventana.

La redirección de subestructuras tiene dos implicaciones importantes:
- La aplicación no puede suponer que la configuración que especifica para una ventana realmente se reflejará en la ventana en la pantalla; por lo cual antes de dibujar en la pantalla, se debe determinar su configuración.
- Como toma tiempo la redirección, la aplicación no puede suponer que la ventana está lista inmediatamente para dibujar en ella, debiendo esperar un evento significando que ésta ya es visible.

La mayoría de los administradores de ventanas también pueden crear y destruir aplicaciones, conocido como *administración de sesión*, teniendo también la habilidad de iniciar cualquier cliente y controlar sus argumentos de línea de comando y guardar el estado del grupo completo de clientes para luego poder reinicializarlos en su misma ubicación en la pantalla.
Cada aplicación incluye su línea de comando como un “hint” para que el administrador de ventana o sesión tenga suficiente información para regresar al estado actual.

2.2 Que son X-Windows ?

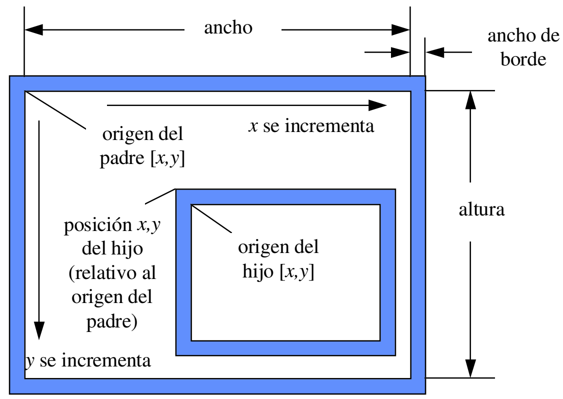
Un servidor X controla una pantalla de bitmaps, dividida logicamente en ventanas rectangulares, cada uno con su propia actividad.

Las propias ventanas están hechas de por si de muchas capas de ventanas de varios tamaños, scrollbar, titlebar, botones de comando, y otros elementos.

2.2.1 Características de la Ventana

Cada ventana tiene una ventana *padre* (*parent*), que se asigna cuando la ventana se crea.
Cada ventana esta contenida dentro de los limites de su padre.
La ventana no puede desplegar salida fuera de su propia área y no puede recibir entradas del teclado o ratón, mientras este apunte fuera del área de la ventana (a menos que *grab* o *keyboard focus* esté especificado).
Cada ventana pertenece a una jerarquía definida por sus hijos, su padre, el padre del padre, etc.
La primera de todas las ventana, la que no tiene padre, se llama la ventana raíz y ocupa la pantalla completa, siendo creada por el servidor X al inicializarse.

Cada ventana tiene su propio sistema de coordenadas, siendo el origen de la ventana la esquina izquierda de arriba, y las coordenadas x e y se incrementan hacia la derecha y hacia abajo.



Para resumir, la configuración de la ventana incluye:

- La altura y ancho de la ventana en pixeles, sin incluir el borde.
- El borde de la ventana. Puede variar en ancho; cero significa borde invisible.
- La posición particular de la ventana en la pantalla, especificada por *x* e *y* en pixeles, medido desde el origen del padre (la esquina superior izquierda, dentro del borde) a la esquina superior izquierda del hijo, fuera de su borde.
- El orden particular de superposición entre las ventanas con el mismo padre.

El ancho, altura, y posición se conocen en conjunto como la *geometría* de la ventana.
Las aplicaciones a menudo permiten a los usuarios especificar la geometría y ancho de borde de la ventana como un argumento de la línea de comando o por medio de mecanismos de omisión.

Una ventana tiene características referidas como ***profundidad*** (***depth***) y ***visual***, las cuales juntas determinan sus características de color.
La profundidad es el número de bits existentes para cada pixel para representar color (o escalas de gris).
El visual representa la forma en que los valores del pixel se traducen para producir salida a color o monocromo en el monitor.

Una ventana tiene una ***clase*** de tipo **InputOutput** o **InputOnly**.
No existe una ventana de tipo output-only ya que todas las ventanas requieren eventos de entrada.

Una ventana tiene un conjunto de *atributos*, los cuales controlan muchos aspectos de la apariencia y respuesta de la ventana:

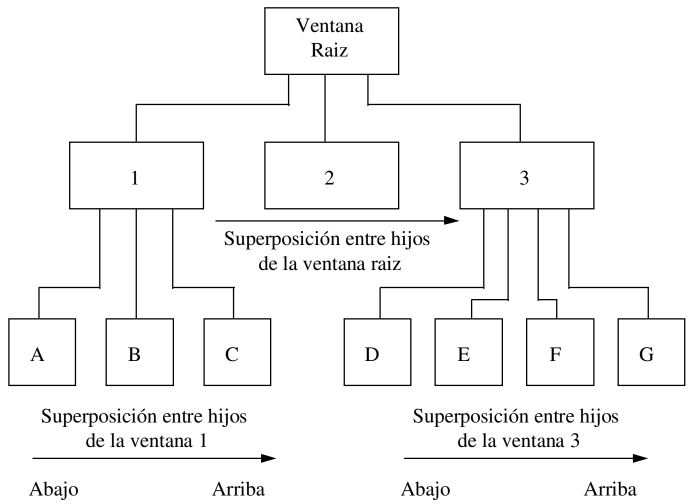
- Qué color o patrón se utiliza para el borde y fondo de la ventana ?
- Cómo se reubican contenidos parciales de las ventanas cuando se cambia su tamaño ?
- Cuándo se guardan automaticamente los contenidos de la ventana según se cubren y exponen ?
- Qué tipo de eventos se reciben, y que tipos se descartan sin ser pasados a las ventanas ancestras ?
- Debería poder desplegar, mover, o cambiar el tamaño de la ventana sin notificar al administrador de ventana ?
- Qué mapa de colores se usa para interpretar los valores de los pixeles dibujados en la ventana ?
- Qué cursor debe ser desplegado cuando el apuntador está en la ventana ?

2.2.2 Jerarquía de Ventana

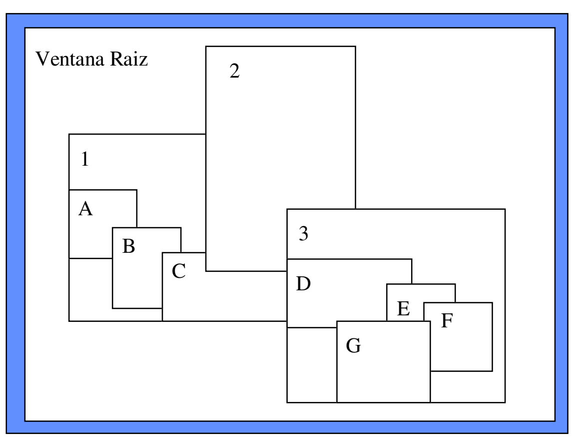
Las ventanas se arreglan en una jerarquía como un árbol genealógico, excepto que sólo se requiere un padre para crear una ventana hijo.
Existe una jerarquía separada para cada pantalla.
Arriba esta la ventana *raíz*, que llena la pantalla completa y se crea cuando se inicializa el servidor.
Las primeras ventanas a ser creadas por cada cliente son hijos de la ventana raíz.
Se utilizan las llamadas **XCreateWindow()** o **XCreateSimpleWindow()**.

Los hijos de la ventana raíz son especiales, ya que son las ventanas de alto nivel para cada aplicación y las administra el administrador de ventanas.

Como ejemplo, se muestra la siguiente jerarquía de ventanas de forma esquemática:



La superposición en la pantalla se muestra a continuación:



Un hijo puede posicionarse parcialmente o completamente fuera de su ventana padre, pero la salida al hijo se despliega y la entrada se recibe solamente en el área donde el hijo se traslapa con el padre.

Si una ventana se mueve de tal manera que se extendería más allá del borde del padre, se haría un clip, y se mostraría solamente la parte que traslapa con el padre.

Algunos términos en uso:
- *Subventana* - es sinónimo con ventana hijo.
- *Hermanos* (siblings) - son las ventanas creadas con el mismo padre.
- *Descendientes* - son los hijos de la ventana junto con sus siguientes generaciones.
- *Ancestros* - son el padre de una ventana y todos sus antepasados.

2.2.3 Superposición de Ventanas (Window Stacking Order)

Cuando una ventana se traslapa con uno de sus hermanos, la que está arriba oscurece parte de la otra.
El ordenamiento determina qué ventana aparece arriba.
Este orden puede cambiarse con varias rutinas para subir, bajar, o circular ventanas entre sus hermanos.
Estas rutinas sólo afectan un grupo de hermanos y sus descendientes pero no sus ancestros.

Las ventanas hijo siempre se mantienen en frente de su padre.
Cuando una ventana con hijos se mueve en el ordenamiento, todos sus hijos se mueven con él.

2.2.4 Mapeo y Visibilidad

Una ventana recién cerrada no aparece inmediatamente en la pantalla.
Es una entidad abstracta donde aun no puede dibujarse.
Mapeo marca una ventana como elegible para despliegue.
Si no esta oscurecida por hermanos o hermanos de ancestros, puede estar visible, y sólo entonces se puede dibujar.

XMapWindow() mapea una ventana en su posición actual en su ordenamiento, mientras que **XMapRaised()** ubica a la ventana en la cima del ordenamiento de sus hermanos antes de mapearse.
Para una nueva ventana nunca antes mapeada, estas dos llamadas son equivalentes, ya que la posición inicial de ordenamiento es arriba.

Se debe mapear una ventana para poder verla, pero esto no es suficiente:
1. La ventana debe mapearse con **XMapWindow()** o rutinas similares.
2. Todos los ancestros de la ventana deben estar mapeados.
3. La ventana no debe estar oscurecida por hermanos o ancestros visibles. El ordenamiento puede manipularse con **XCirculateSubwindows()**, **XConfigureWindow()**, y **XRestackWindows()**.
4. El buffer de solicitudes debe haberse vaciado por una rutina que recibe eventos, con una llamada a **XFlush()**, o con una función que solicita información del servidor.
5. El mapeo inicial de una ventana de alto nivel es un caso especial, ya que la visibilidad de la ventana puede retardarse por el administrador de ventanas por razones de redirección de

subestructuras. Por razones complicadas el cliente debe esperar el primer evento **Expose** antes de asumir que su ventana está visible y se le puede dibujar.

Una consecuencia importante de esto, es que desmapeando una ventana, con **XUnmapWindow()**, se borra la ventana y todos sus descendientes de la pantalla.
X permite controlar la ubicación y visibilidad de un cliente completo hecho de una jerarquía de ventanas, simplemente manipulando la ventana de alto nivel.

La configuración de ventana y atributos de ventana se mantienen cuando se desmapea una ventana, aunque se pudiera perder el dibujo.
Las operaciones gráficas sobre una ventana que no está visible o que no esta mapeada no tienen efecto.
Las gráficas visibles en una ventana serán borradas cuando esa ventana se oscurece y luego se expone, por lo cual es importante que el cliente redibuje el contenido de la ventana si se necesita.

En algunos servidores existe una *almacenamiento de respaldo* (*backing store*) que guarda el contenido de la ventana para automaticamente redibujarlos cuando la ventana es nuevamente visible después de haber sido opacada.
Este característica es muy cara en término de recursos computacionales y debería utilizarse sólo para ventanas cuyo contenido es difícil de recrear.

Mapeo se logra con las rutinas **XMapWindow()**, o **XMapSubwindows()**.
Desmapeo se logra con las rutinas **XUnmapWindow()**, o **XUnmapSubwindows()**.

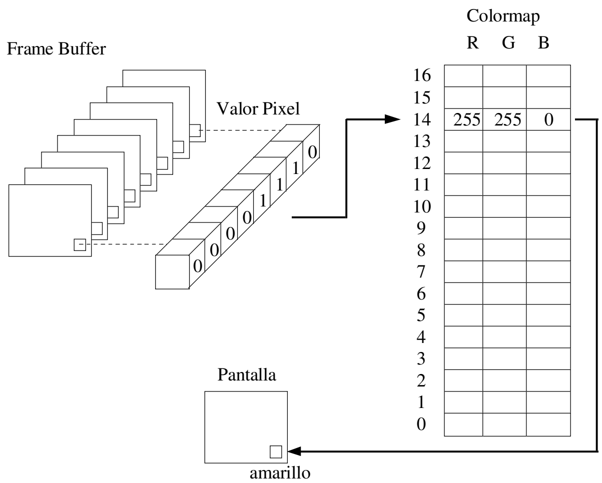
2.3 Introducción a Gráficas en X

2.3.1 Pixeles y Colores

En los despliegues más sencillos de blanco y negro existe sólo un bit por pixel, y el estado del bit determina si el pixel será blanco o negro.
En sistemas a color o monocromos con escalas de gris, existen múltiples bits por pixel.

El estado de los múltiples bits asignados a cada pixel no controla directamente la intensidad del color o escala de gris para ese pixel.
En su lugar, se usan como índices en una tabla (look-up table) llamada ***colormap*** (***mapa de colores***), como se muestra en la siguiente figura.

En un despliegue a color, un pixel consiste de fósforos rojo, verde y azul, cada uno sensible a rayos de electrones separados; la intensidad relativa inmediata de estos tres colores le hace parecer al ojo que es un sólo color.
Por lo tanto, el colormap contiene un arreglo de tripletas de rojo, verde y azul (RGB).
Por ejemplo, si el valor de los bits para un pixel particular (*pixel value*) es 14, los valores de RGB para el miembro 14 del colormap serán desplegados en esa ubicación en la pantalla.



Cada elemento del colormap se llama ***colorcell***, cada una de las cuales traduce un valor de pixel en un conjunto especifico de valores de rojo, verde y azul.
Todos los despliegues con bitmap tienen al menos un colormap en hardware, aunque las pantallas monocromas de un sólo plano, pueden consistir de sólo dos colorcells.
En la mayoría de los casos, todos los clientes comparten el colormap único asignando sólo el número de colorcell que necesitan y compartiéndolo.
Cuando los clientes tienen requisitos especiales, X permite tener colorcells privados o crear colormaps virtuales que luego se intercambian con el colormap de hardware cuando se requiera. Esta es la razón que el colormap es un atributo de la ventana.

2.3.2 Pixeles y Planos

El número de bits por pixel también se refiere como el número de ***planos*** en el despliegue gráfico. Sistemas de blanco y negro tienen un sólo plano, mientras despliegues a color tienen de 4 a 28 planos y los despliegues a escala de gris tienen por lo general de 2 a 4 planos.
X11 apoya hasta 32 planos.

El número de colores posibles o tonos de gris que pueden desplegarse *simultaneamente* (para un sólo colormap) en la pantalla es 2^n, donde n es el número de planos en el despliegue.

Todos los cálculos gráficos se hacen en valores de pixel antes de traducirse a RGB.
Los valores *fuentes* (*source*) de pixel especificados en la solicitud de dibujo y los valores de la *vieja destinación* de pixel se combinan de acuerdo a la máscara del plano, máscara de clip, y función lógica (o sea el ***GC* - *Contexto Gráfico***) para llegar a los valores de *destinación* final de pixel.

Los macros **BlackPixel()** y **WhitePixel()** devuelven los valores de pixel que mapean el negro y blanco usando el colormap de omisión para la pantalla.
Estos macros están garantizados para uso en programas monocromos para trabajar en despliegues monocromos, escala de gris y a color.
Para hardware a color, los colores de blanco y negro pueden no ser realmente blanco y negro, pero se garantiza que contrastaran entre ellos.

2.3.3 Pixmaps y Drawables (Dibujables)

Una ventana no es la única destinación válida para dibujar.
Pixmaps también son destinaciones válidas para solicitudes gráficas.
Un ***pixmap*** es un bloque de memoria fuera de la pantalla en el servidor.
Las ventanas y pixmaps se conocen en conjunto como ***drawables*** (***dibujables***).

Un pixmap es un arreglo de valores de pixel, teniendo una profundidad similar a cualquier ventana.
Sin embargo, no tiene una posición relativa a algún otra ventana o pixmap, y no tiene atributos de ventana, tales como colormaps.
Todas esas cosas afectan un pixmap sólo cuando se copia a una ventana, siendo el pixmap visible sólo luego de copiarse a la ventana.

Existen varias rutinas para crear pixmaps, siendo la más sencilla **XCrearePixmap()**, la cual crea una pixmap vacío con contenido indefinido.
Se debe siempre borrar un pixmap creado con **XCrearePixmap()** antes de poder usarlo, para asegurar que no contenga basura.

Algunas rutinas operan sólo en pixmaps o sólo en ventanas.
Estas rutinas especifican como argumento **Pixmap** o **Window**.
Si ambas están permitidas, el argumento a la rutina de Xlib será especificada como **Drawable**.
Todas las rutinas para dibujar especifican el tipo de argumento **Drawable**.

Un pixmap no es susceptible a estar cubierto por otras ventanas.
Ventanas por otro lado, sólo se pueden dibujar de forma útil cuando son visibles, ya que su contenido no se mantiene cuando esta tapado o desmapeado (a menos que se haya asignado el almacenamiento de respaldo).

Para ser copiardo a una ventana con **XCopyArea()**, un pixmap debe tener la misma profundidad que la ventana.
Una vez copiado, el colormap asociado con la ventana se utiliza para traducir los valores de pixel del pixmap a colores visibles.
Después de copiar, dibujos adicionales en el pixmap no aparecerán en la ventana.
Un sólo plano de un pixmap de cualquier profundidad puede copiarse a cualquier ventana con **XCopyPlane()**.

El copiado de pixmaps a ventanas puede tener una baja muy grande en rendimiento.
Quizás más importante, la memoria fuera de pantalla en el servidor usada para pixmaps puede estar limitada en cantidad.
Por lo tanto, programas que usan muchos pixmaps pueden que no trabajen en servidores PC y terminales X.

Un pixmap de profundidad 1 se conoce como un ***bitmap***, aunque no existe tipos o recursos separados llamados Bitmap.
Un bitmap es un arreglo de bits en dos dimensiones usados para muchos propósitos incluyendo definiciones de cursores, fonts, y templates para figuras de dos colores.
Cada bit representa un sólo valor de pixel que se asigna (set - 1) o desasigna (unset - 0).
Dependiendo en el tipo visual, estos valores de pixel se pueden interpretar como dos colores o simplemente como blanco y negro.

2.3.4 Dibujando y el Contexto Gráfico

Como toda biblioteca gráfica, X provee rutinas especiales, llamadas ***primitivas gráficas***, para dibujar puntos, líneas, rectángulos, polígonos, arcos, texto, etc.
Pero en X, una primitiva gráfica particular no contiene toda la información necesaria para dibujar un gráfico particular.
Un server llamado el ***contexto gráfico*** (***GC***) especifica las variables restantes, tales como el ancho de línea, colores, y patrones de relleno.
El ID de un GC se especifica como un argumento a la rutina de dibujo y modifica la apariencia de todo lo que se dibuja en ese dibujable.

El GC debe ser creado por el cliente antes de cualquier dibujo.
El GC creado se guarda en el servidor, para que la información que contiene no tenga que enviarse con cada primitiva gráfica, sólo su ID se envía.
Más de un GC puede crearse, pudiendo definirse diferentes valores.

2.3.5 Tiles and Stipples

Se pueden también utilizar pixmaps para dibujar patrones en un área, como para el fondo de una ventana o en un GC, conocidos como tiles o stipples.

Un ***tile*** es un pixmap con la misma profundidad que el dibujable donde se utilizará.
El tile es tipicamente de 16x16 pixeles pero puede ser de otros tamaños.
(Algunos tamaños se pueden dibujar más rápido según el hardware, ver **XQueryBestTile()**.)
Está compuesto tipicamente de sólo dos valores diferentes de pixeles, ya que es el tipo más fácil de crear, pero también se permiten múltiples valores de pixel.

Un ***stipple*** es un pixmap con profundidad 1, y se utiliza junto a valores de pixeles de primer plano y a veces con pixeles de fondo de forma similar a un tile.

2.4 Más sobre Características de Ventanas

2.4.1 Atributos de Ventana

Los atributos de ventana consisten de información sobre cómo la ventana debe verse y actuar.
Cada ventana tiene un conjunto separado de atributos, que pueden asignarse con **XChangeWindowAttributes()**, o en algunos casos con rutinas que cambian atributos individuales.

Los atributos controlan los siguientes aspectos de la ventana:

Background Puede ser un color sólido, un tiled pixmap, o transparente.

Border Puede ser un color sólido o un tiled pixmap.

Bit Gravity Determina cómo se preserva el contenido parcial de una ventana cuando ésta cambia de tamaño.

Window Gravity Determina cómo ventanas hijos se reubican cuando una ventana cambia de tamaño.

Backing Store Provee insinuaciones (hints) sobre cuando el contenido de una ventana debe guardarse automaticamente mientras la ventana se desmapea u oscurece, qué planos de despliegue deben guardarse, y qué valores de pixeles se utilizarán cuando se recupera planos no guardados. No todos los servidores tienen backing (determinar con **DoesBackingStore**()).

Saving Under Provee hints sobre si el área de la pantalla debajo de una ventana, tal como un menú pop-up, debe salvarse para no tener que redibujarse cuando la ventana se remueve. No todos los servidores lo apoyan (determinar con **DoesSaveUnders**()).

Eventos Indica cuales eventos serán recibidos y cuales eventos no deben enviarse a ventanas ancestras.

Substructure Redirect Override
Determina si se le debe permitir a la ventana ser mapeada en la pantalla sin intervención del administrador de ventanas. Esto se hace usualmente para menús y otras ventanas que frecuentemente se mapean y luego casi inmediatamente se desmapean.

Colormap Determina qué colormap virtual debe utilizarse para la ventana.

Cursor Determina cual cursor debe desplegarse cuando el apuntador esta en la ventana.

Algunos de los atributos de ventanas que no tienen efecto:
No se afecta el tamaño o posición de una ventana, su padre, o su ancho de borde; ésto es parte de la configuración de la ventana.
No se afecta la profundidad, clase, o visual de una ventana; éstas están permanente asignadas cuando se crea la ventana.
Los atributos no determinan como se interpretan las solicitudes gráficas; éste es el trabajo del contexto gráfico.

2.4.2 Configuración de Ventana

La configuración de una ventana consiste de su posición, ancho y altura, ancho de borde, y superposición.
Estos factores se manejan de forma diferente de los atributos de ventana (aunque internamente se guardan como estructuras de **XWindowAttributes**) por una razón importante: cambios a la configuración de una ventana de alto nivel (un hijo de la ventana raíz) debe hacerse en cooperación con el administrador de ventanas.

2.4.3 Clase: Ventanas tipo InputOutput e InputOnly

La diferencia principal entre la dos clases es que una ventana **InputOnly** no puede utilizarse como dibujable.
Por lo tanto, este tipo de ventanas tiene un conjunto más limitado de atributos de ventana, no tiene borde y tiene un fondo transparente, y no puede tener ventanas hijo de tipo **InputOutput**.

Ventanas tipo **InputOnly** hacen un área invisible en la pantalla donde la entrada tiene un propósito diferente pero el despliegue no cambia, asignándose por lo general un cursor diferente para su distinción.
Este tipo de ventana rara vez se usa, y luego de asignado durante la creación de la ventana, ya no puede cambiarse.

2.4.4 Profundidad y Visual

La profundidad y visual de una ventana se asignan durante su creación y no pueden luego cambiarse.
La profundidad es el número de planos que se utilizan para representar color o escalas de gris en la ventana, siendo también el número de bits por pixel.
La profundidad máxima permisible para ventanas **InputOutput** es el número de planos apoyado por la pantalla a la cual esta asociada.

La profundidad de una pantalla **InputOnly** es siempre 0.

El visual toma en cuenta las diferencias entre varios tipos de hardware de despliegue para determinar cómo los valores de pixeles se traducen en colores visibles dentro de una ventana particular.
Una pantalla puede apoyar uno o varios tipos de visuales.
Una estructura **XVisualInfo** contiene toda la información sobre un visual particular.
Un miembro de **XVisualInfo** es la clase visual, que tiene uno de los valores **DirectColor**, **GrayScale**, **PseudoColor**, **StaticColor**, o **TrueColor**.
Estos valores especifican las características de los colormaps que pueden utilizarse con la ventana, si el colormap es sólo-lectura o lectura/escritura, color o monocromo, dividido en tres colores primarios o compuestos.
Otros miembros especifican el rango válido de valores de pixeles; cuantos bits del pixel están asignados a rojo, verde y azul; y otras variables.

La profundidad y visual se heredan del padre cuando una ventana se crea con **XCreateSimpleWindow()**.

2.4.5 Iconos

Un ***icono*** es una pequeña ventana que indica que una ventana principal más grande existe pero actualmente no esta mapeada en la pantalla.

La mayoría de los administradores de ventanas permiten a un usuario *iconificar* una aplicación para que no estorbe pero sin destruirla.
Una aplicación iconificada continúa ejecutándose, al menos que específicamente se haya pedido que no.
Cuando se requiere entrada, el programa puede esperar hasta que la ventana se desiconifique o aceptar entrada en el icono.

Por medio de hints del administrador de ventanas, una aplicación pasa el nombre del icono y pixmap a ser desplegados en la ventana del icono.
La aplicación puede crear su propia ventana icono y pasar su ID al administrador de ventanas.

2.4.6 Características Especiales de la Ventana Raíz

La ventana raíz tiene características un poco diferentes de las demás ventanas.

La ventana raíz es de tipo **InputOutput**, estando siempre mapeada.
Su tamaño no puede cambiarse, y su esquina superior izquierda es siempre la esquina superior izquierda de la pantalla, donde las coordenadas globales son (0,0).
La ventana raíz tiene un ancho de borde igual a cero.

Los atributos de omisión de la ventana raíz incluyen un pixmap de fondo con líneas cruzadas diagonales, el colormap de omisión, y un cursor de omisión que es una X grande.
Todos éstos pueden cambiarse.
El atributo de máscara de eventos puede cambiarse, pero por omisión, ningún evento que ocurre en la ventana raíz se envían a ningún cliente.
Ninguno de los demás atributos de ventana se aplican a la ventana raíz.

La ventana raíz nunca se iconifica ya que, entre otras razones, no puede desmapearse.

2.5 Introducción a Eventos

2.5.1 Qué es un evento ?

Un evento es un paquete de información que genera el servidor cuando ciertas acciones ocurren y se ponen en una cola para uso posterior por el cliente.
La cola de eventos se puede leer más tarde en cualquier orden, aunque normalmente se leen y procesan en el orden en que han ocurrido.

Algunos tipos de eventos:
- Presionar una tecla (**KeyPress**).
- Presionar y soltar el botón del ratón (**ButtonPress**, **ButtonRelease**).
- Mapeo o desmapeo de la ventana (**MapNotify**, **UnmapNotify**).
- Ratón cruzando el borde de la ventana (**EnterNotify**, **LeaveNotify**).

Estos tipos de eventos por lo general se usan para entrada de usuario y para controlar interfaces de usuario.

Un segundo grupo de eventos reportan efectos secundarios de operaciones de ventanas.

Los siguientes son algunos eventos de este tipo:

- Cuando una ventana se hace visible luego de haber estado oscurecida, recibe el evento **Expose**.
- Cuando la gravedad de la ventana (un atributo de ventana que controla la posición de una ventana cuando su padre cambia de tamaño) toma efecto, se genera el evento **GravityNotify**.
- Un cliente puede solicitar que todas las entradas del teclado se envíen a una ventana particular sin importar la posición del apuntador; esto se conoce como una ventana de *foco de teclado* (*keyboard focus*). Cambiar el foco de teclado de una ventana a otra causa eventos **FocusIn** y **FocusOut**, indicando al cliente si éste puede esperar eventos adicionales de teclados.
- Cambiar el mapeo entre las teclas del teclado y los códigos que generan causa que se envíe un evento **MappingNotify** a todos los clientes.
- El administrador de ventanas a veces puede cambiar el padre de una ventana para agregar un marco a las ventanas en la pantalla. Esta acción genera un evento **ReparentNotify**.
- Se genera un evento **PropertyNotify** cuando un cliente cambia una propiedad en una ventana.

Un tercer propósito de los eventos es permitir a varios clientes comunicarse entre sí y con el administrador de ventanas.

- Los eventos **SelectionClear**, **SelectionNotify**, y **SelectionRequest** se usan para comunicarse entre un cliente que le permite a un usuario seleccionar una sección de texto (u otra información) y un cliente que le permite a un usuario poner la información en su ventana. Algunos de estos eventos se envían con **XSendEvent**.

2.5.2 Selección y Propagación de Eventos

Un cliente debe seleccionar los tipos de eventos que desea que el servidor envíe para cada ventana.
La selección se hace llamando a **XSelectInput()**, la cual asigna el atributo de ventana **event_Mask**.
También se puede hacer la selección asignando el atributo de ventana **event_Mask** con la rutina más general **XChangeWindowAttribute()**, o como parte del llamado **XCreateWindow**().

Por ejemplo, un scrollbar puede requerir eventos del botón del ratón pero no del teclado, mientras que la ventana principal de una aplicación puede requerir eventos del teclado pero no del ratón. Se seleccionaría diferentes tipos de eventos para cada una de estas ventanas.

Los eventos de teclado y apuntador se generan en la ventana más pequeña que rodea al apuntador.
Un evento de estos tipos (solamente) se propaga hacia arriba a través de la jerarquía de ventanas hasta que se encuentre en una ventana el tipo de evento en los atributos **event_Mask** o **do_not_propagate_Mask**.
Si primero se encuentra el evento en **event_Mask** (o ambos en la misma ventana), entonces el evento se envía como si hubiera ocurrido en esa ventana.

Si primero se encuentra en **do_not_propagate_Mask**, entonces el evento nunca se envía.
El ID de la ventana que finalmente recibió el evento se pone en el componente **window** de la estructura de evento.

El atributo **do_not_propagate_Mask** sólo puede asignarse con **XChangeWindowAttribute()** o **XCreateWindow()**.
Eventos que no sean de teclado o apuntador no se propagan, ocurren en la ventana donde fueron seleccionados cuando la acción apropiada ocurrió.

Para la mayoría de tipos de eventos, una copia de un evento puede enviarse a más de un cliente si cada cliente ha seleccionado ese tipo de evento en la ventana.
Cada cliente tiene su propia máscara de eventos para cada ventana.
El cliente que creó la ventana no tiene que hacer nada para cooperar.
El segundo cliente que quiere recibir un evento de una ventana que no creó, simplemente necesita encontrar el ID de la ventana y luego seleccionar los tipos de eventos deseados con **XSelectInput()** en esa ventana.
Un evento duplicado se envía a cada ventana, y estos eventos se propagan de forma independiente hacia arriba en la jerarquía en las dos aplicaciones.
(Esto raramente se hace en la realidad.)

2.5.3 La Cola de Eventos

Cada cliente tiene su propia cola de eventos la cual recibe los eventos seleccionados en el orden en que los envía el servidor.
El cliente puede remover cada evento en cualquier momento y procesarlo de acuerdo a su tipo e información adicional en cada estructura de evento.
El cliente también puede leer eventos en la cola sin removerlos, remover uno y luego regresarlo, o limpiar la cola borrando todos los eventos.
Eventos también pueden ser creados por un programa y enviados al administrador de ventanas u otros programas.

2.5.4 Estructura de Eventos

Expose es uno de los tipos de eventos más importantes.
Se genera cuando un área de la ventana se hace visible en la pantalla e indica que el cliente debe redibujar el área.
Esto ocurre cuando la ventana se mapea, mueve, cambia de tamaño, o deiconifica o cuando una ventana que la cubre se desmapea.
Eventos de exponer son comunes y pueden ocurrir en cualquier momento, ya que pueden ser causados por acciones de otros clientes.

```
typedef struct {
      int type;
      unsigned long serial;/* última solicitud procesada por servidor */
      Bool send_event; /* True si vino de una solicitud de SendEvent */
      Display *display; /* Despliegue de donde se leyo el evento */
      Window window;
      int x,y;
```

```
      int width, height;
      int count; /* Si no es cero, más eventos de expose siguen */
}XExposeEvent;
```

El tipo de evento se reporta en cada estructura de evento.
En la estructura **XExposeEvent**, el campo ***type*** sería la constante simbólica **Expose**.
La ventana a la cual el evento se propagó se reporta en el campo ***window***, presente en la mayoría de tipos de eventos.

2.5.5 El Ciclo de Eventos (Event Loop)

Como los eventos pueden llegar en cualquier momento, la estructura del código para manejar los eventos está predeterminada.
Cada programa contiene un ciclo de eventos donde cada evento se recibe y se procesa.
Normalmente este ciclo se implementa como un ciclo **while** infinito, comenzando con una rutina para recibir el evento, seguida por una línea de **switch** que se ramifica según el tipo de evento.
Dentro de cada rama para un tipo de evento, pueden haber ramificaciones adicionales correspondientes a la ventana donde el evento ocurrió u otros campos en la estructura de eventos.

El ciclo casi siempre incluirá eventos de exponer.
Cuando una ventana se mapea por primera vez, la primera función del programa es leer el evento de exponer que se generó al mapear la ventana.
Luego el programa puede dibujar el contenido de la ventana.
Esta también debe ser la forma de respuesta a eventos de exposición que lleguen más adelante.

Otro tipo de evento, **ConfigureNotify**, debe manejarse si el administrador de ventana modificó el tamaño de la aplicación antes de mapearla y en caso de que el usuario cambie el tamaño de la ventana más adelante.

2.6 Como programar con Xlib

2.6.1 Diseñando una Aplicación en X

Desde el punto de vista del usuario, una aplicación bajo un sistema de ventanas casi siempre creará una ventana en la pantalla de tamaño apropiado, determinará una posición para algún texto y/o gráfico dentro de la ventana, dibujará en la ventana, y aceptará entrada del teclado y/o del apuntador, cambiando la pantalla de forma correspondiente.

Existen algunas diferencias entre los diferentes sistemas gráficos por lo cual se debe determinar los requisitos de diseño para la aplicación, en este caso para X.

2.6.1.1 Requisitos de Diseño

Los siguientes cuatro párrafos describen lo que aplicaciones en X deben hacer para operar apropiadamente y que no son obvias.

1. X permite que las estaciones de trabajo se conecten en una red donde un nodo puede ejecutar programas en X y desplegarlos en cualquier otro nodo, dado permisos adecuados. Esto significa que el programa debe poder aceptar especificaciones del usuario de cual despliegue utilizar (esto equivale a establecer la conexión entre el cliente y el servidor apropiado). Este requisito no requiere virtualmente ninguna programación adicional.
2. La aplicación debe ser responsable en su utilización de los recursos limitados de despliegue, particularmente espacio de pantalla y colormaps, ya que pueden haber muchas aplicaciones corriendo de forma concurrente en la misma pantalla, compartiendo esos recursos limitados. El cliente encargado de manejar estos recursos limitados es el administrador de ventanas, Existen ciertos requisitos para la comunicación entre cada aplicación y el administrador de ventanas para asegurar que necesidades que compiten pueden tratarse de forma equitativa y ayudar a mantener una interfaz consistente. Estos requisitos no son difíciles de lograr para aplicaciones sencillas, pero se hacen más complejas en aplicaciones más grandes.
3. Otros clientes pueden moverse sobre nuestro cliente y luego retirarse, requiriendo que nuestro cliente redibuje su ventana o ventanas. X no puede mantener los contenidos de un número ilimitado de ventanas que se traslapan, y es ineficiente tratar de mantener inclusive pocas. Se le avisará a nuestro cliente cuando se requiere redibujar y en que áreas. Este requisito no es difícil de lograr, pero requiere programación de forma que se mantenga el “estado” actual de cada ventana para poder se redibujada.
4. El usuario puede cambiar el tamaño de la aplicación, por lo cual ésta debiera ser capaz de recalcular sus dimensiones y ubicación de las subventanas y gráficos para que se ajusten en la ventana dada.

Para los programas más sencillos, estos requisitos son fáciles de satisfacer.

2.6.1.2 Interfaz de Usuario

El primer paso en el diseño de una aplicación será determinar cuales serán sus características.
El siguiente paso probablemente será determinar cómo el usuario invocará estas características.
Por lo cual se requiere diseñar una interfaz de usuario.

X fue diseñado para que sea independiente de una política, y por lo tanto no viene con una interfaz de usuario estándar como muchos otros sistemas de ventanas.
Se tiene que escribir explicitamente todas las partes de la interfaz de usuario, a menos que se decida utilizar algunos de los toolkits existentes.
Utilizar un toolkit hace que la construcción de la interfaz de usuario sea más sencilla y se recomienda mucho, ya que de lo contrario, se deberá escribir menús, botones de comando, cajas de dialogo, etc., y determinar cómo serán utilizados.

Los elementos principales que interactúan en el diseño de una interfaz de usuario son la jerarquía de ventanas y la selección y procesamiento de eventos, particularmente los eventos del apuntador y teclado.
Como estos eventos de dispositivos se propagan a través de la jerarquía, dependiendo de si se han sido activados para cada ventana, la jerarquía y la selección juntas determinan cómo los eventos se recibirán.

Para cada acción de usuario, debe haber un camino (posiblemente único, posiblemente común para varias acciones diferentes de usuario) a través del código de manejo de eventos que obtenga algún tipo de respuesta para el usuario, sea por cambio visible, un mensaje, o un beep.
Por lo tanto, el trabajo del ciclo de eventos es distinguir todas las posibles acciones del usuario e invocar el código apropiado para su manejo.
En el ciclo principal de eventos, cada instrucción de caso para un tipo de evento debe tener otro switch, dependiendo de la ventana que recibió el evento, antes de llamar la función que ejecuta la acción que el usuario solicitó.
El tipo de eventos y la ventana donde ocurrió son sólo dos de los miembros más comunes de la estructura de eventos, pueden haber instrucciones de switch adicionales basados en otros miembros, tales como que teclas o botones se presionaron.

Especialmente para programas más complejos, un diseño cuidadoso de la jerarquía de ventanas y selección de eventos puede simplificar el código y ahorrar horas de depuración.
Se recomienda trazar la jerarquía de ventanas y definir los tipos de eventos seleccionados por cada ventanas, y luego definir los eventos que se propagarán a las ventanas ancestras.
Esto ayuda a encontrar problemas antes de escribir el código.

2.6.2 Escribiendo una Aplicación en X

La forma más fácil de comenzar a escribir una aplicación en X es quizás copiar una aplicación existente, lo más similar posible a la aplicación deseada o comenzar con un esqueleto de programa, como *basicwin*, que se describe en el siguiente capítulo.
Se puede seleccionar uno de la porción básica de la distribución estandar de MIT, ya que estos son los que probablemente sigan de forma más precisa las convenciones actuales.

2.6.2.1 Recursos y Adaptación (Customizability) de Usuario

Una aplicación no debe restringir en el código todas las posibles opciones bajo X, tales como colores y fonts.
Debe permitir al usuario especificar los colores para todas las ventanas, los fonts, el despliegue y pantalla a utilizar, el tamaño inicial y posición de la aplicación, y un número grande de otras opciones estándares y específicas a la aplicación.

Una aplicación debe proveer opciones de línea de comando, pero existen demasiadas variables para apoyarlas a todas como opciones de línea de comando.
Existen en X **recursos** para especificar opciones.
El usuario pone las opciones deseadas en un archivo utilizando un formato particular y ejecuta la aplicación de X llamada ***xrdb*** especificando este archivo como argumento de línea de comando.
xrdb pone una propiedad en la ventana raíz cuyos valores son el contenido de este archivo.
Las aplicaciones utilizan un conjunto de rutinas de Xlib llamadas administrador de recursos para devolver una asignación para cada variable requerida.
La rutina **XGetDefault()** simplifica este proceso para la aplicación.
Si el usuario no ha llamado ***xrdb*** para asignar la propiedad en la ventana raíz, **XGetDefaults()** lee un archivo llamado ***.Xdefaults*** en el directorio raíz del usuario.

La propia aplicación debiera contener valores de omisión para cada variable en caso de que ninguno de estos recursos contenga un valor correspondiente.

Este uso del término recurso es completamente diferente del termino *recurso del servidor*, el cual se refiere a ventanas y contextos gráficos.

Una especificación de recurso es un par llave/valor, que se aplica solamente a una ventana particular dentro de una aplicación particular, a una aplicación completa, a una clase especial de aplicaciones tales como editores, o a todas las aplicaciones.
El algoritmo utilizado para encontrar el valor para una variable particular opera un poco diferente de un manejador normal de bases de datos.
Dada una especificación incompleta de una llave, utiliza un algoritmo para determinar cual de las llaves en la base de datos de recursos es la más cercana y devuelve el valor indicado.
Siempre devuelve un valor único.
Esto es diferente de un manejador de bases de datos normales el cual devolvería muchos valores si la llave fuera muy general.

2.6.2.2 Compilando y Ligando Programas en X

Para usar todas las funciones en la biblioteca X, se requiere incluir *<X11/Xlib.h>*, *<X11/Xutil.h>*, *<X11/keysym.h>*, y *<X11/Xresource.h>*.
Estos archivos definen estructuras de datos, macros y símbolos, y declaran los tipos de funciones.
También es una buena idea incluir *<X11/Xos.h>*, el cual incluye ciertos archivos de encabezados comunmente utilizados en programas en C que difieren en nombre o ubicación entre varios sistemas operativos, especialmente System V y BSD.
Para compilar y ligar un programa con todas las bibliotecas existentes de Xlib, incluyendo una tabla de símbolos para un depurador simbólico, se debe usar:

```
cc -g -o archivosalida archivoentrada.c -lX11
```

La opción *-lX11* especifica ligar con la biblioteca estándar de X, Xlib.

Otra bibliotecas incluyen: Xmu, servicios varios y Xext, extensiones (que requiere software adicional del lado del servidor para poder funcionar).
Estas aun no han sido estándares adoptados por el consorcio X pero existen ampliamente.
Si se utiliza *-lXmu*, debe ponerse antes de Xlib en la instrucción de compilación.
Todas las bibliotecas de toolkits basadas en Xlib, tales como Xt, también deben aparecer antes de *-lX11* en la instrucción de compilación.
-lXext puede aparecer antes o después de *-lX11* ya que no utiliza Xlib.

Si se obtiene errores al compilar sobre archivos de encabezados en *<X11/Xos.h>* que no se encuentra, probablemente se este en System V que no define el símbolo **SYSV**.
Para resolver este problema simplemente agregar *-DSYSV* a la instrucción de compilación.

Es bueno utilizar *make*(1) para administrar las instrucciones de compilación.

Incluso mejor es utilizar *make* en combinación con *imake*, el cual genera *makefiles* de forma portátil.
imake se incluye con la distribución X (ver *The X Resource*, Issue 2, Spring 1992).

2.6.2.3 Convenciones de Nombres

Existen un número de convenciones para nombres de funciones y constantes en Xlib.
Las más importantes son las siguientes:

- Los nombres de todas las funciones de Xlib comienzan con una **X** mayúscula. Palabras compuestas se construyen poniendo en mayúscula la primera letra de cada palabra, e.g. **XAllocColor()**.
- Los nombres de las estructuras de datos y tipos de estructuras más visibles al usuario comienzan con una **X**. Las únicas excepciones son **Depth**, **Display**, **GC**, **Screen**, **ScreenFormat**, y **Visual**. Apuntadores a estas seis estructuras son bastante comunes en programas, pero sus miembros no deben accesarse excepto a través de macros preestablecidos.
- Los nombres de todos los miembros de estructuras de datos utilizan minúsculas. Nombres compuestos se enlazan con una línea de subrayar.
- Los nombres de macros no comienzan con una X. Para distinguir macros de símbolos definidos por el usuario , la primera letra de cada palabra en el macro se pone en mayúscula. (Existe una excepción por razones históricas, y es la biblioteca de *quarks*, estructura para representar cadenas, donde los nombres comienzan con **Xrm**.)
- Los nombres de constantes simbólicas definidas en archivos de encabezados en X (**#defined**) usan letras mezcladas, con la primera letra de cada palabra capitalizada, pero no comienzan con X. Letras minúsculas se reservan para variables y todo en mayúsculas para símbolos de usuario, de acuerdo a convenciones existentes. La única excepción son nombres de átomos predefinidos los cuales usan sólo mayúsculas, con líneas de subrayar separando las palabras. Los átomos comienzan con **XA_** para distinguirlos de símbolos de usuario.

Se debe escoger nombres de constantes y rutinas que no se confundan con las de Xlib.
Nombres de funciones deberían comenzar siempre con una minúscula.
Constantes deberían ser todas en mayúscula.
Variables con minúsculas, con líneas de subrayar separando palabras.

2.6.2.4 Usando Estructuras, Símbolos, y Máscaras

Apuntadores a estructuras es la forma más importante de especificar datos que se envían y devuelven a rutinas de Xlib.
Si la rutina devuelve datos, el valor devuelto será un apuntador a la estructura de datos, a menos que la rutina devuelva más de una estructura, en cuyo caso una o todas las estructuras serán argumentos.

Cuando se asigna las características de un recurso del servidor, tal como un conjunto de atributos de ventana, un contexto gráfico, las células en el colormap, o características de hardware (tal como presión de una tecla), una estructura y una máscara se especifican como argumentos.

La *máscara* especifica qué valores en la estructura especificada debe leerse cuando se actualiza el valor del recurso.
Un bit en la máscara se asigna a cada miembro de la estructura, y una constante especial se define en los archivos de encabezados de Xlib para representar ese miembro cuando se construye la máscara.
Cada una de las constantes de la máscara tiene un bit asignado.
El argumento de la máscara se hace combinando cualquier número de constantes de la máscara con el operador OR (|).
Por ejemplo, la constante **CWBackgroundPixmap** se usa para indicar que el miembro **background_pixmap** de la estructura especificada de los atributos de ventana se debe leer y cambiar el miembro correspondiente en el recurso.

La otra utilización importante de las constantes definidas en Xlib (además de máscaras) es como valores para los propios miembros de las estructuras.
Ellas indican cual de un número de alternativas es verdadera.
Por ejemplo, varios de los miembros de la estructura pueden tener sólo valores **True** o **False**.
Otro ejemplo, el miembro type de cada estructura de eventos puede tener uno de 33 valores diferentes, cada uno representado por una constante definida diferente, como **Expose**.
Constantes definidas también se usan como valores de devolución.
Constantes definidas también se usan para átomos predefinidos.

2.6.2.5 Optimizando el Rendimiento

Se debe utilizar lo máximo posible funciones de Xlib que no requieren respuestas de protocolo, tales como rutinas que contengan **Fetch**, **Get**, o **Query**, para no frenar la aplicación por retrasos de la red para los mensajes entre el servidor y cliente.

2.6.2.6 Portabilidad ANSI-C y POSIX

La versión X11R5 se ajusta con los estándares ANSI-C y POSIX, y es portátil en una gran variedad de plataformas.
Muchos sistemas no implementan estos estándares, o no lo hacen completamente, por lo cual X define archivos de encabezado nuevos que intentan ocultar las diferencias entre sistemas., los archivos son *<X11/Xfuncproto.h>*, *<X11/Xfuncs.h>*, *<X11/Xosdefs.h>*, y *<X11/Xos.h>*.
Ninguno de estos archivos es parte oficial del estándar R5, aunque es útil utilizarlas.

2.6.3 Depurando una Aplicación X

Fuera de las herramientas de depuración existentes para lenguajes como C, X incluye algunas otras herramientas.

- *xwininfo* despliega información sobre una ventana, incluyendo su ID y nombre, IDs y nombres de sus padre e hijos, todos los atributos de la ventana, y los hints asignados por el administrador de ventanas para esa ventana.
- *xprop* despliega el nombre, tipo, y valor de cada propiedad asignada en la ventana.

Si la aplicación genera errores de protocolo, es más fácil encontrar el error si se inhabilita el buffer de solicitudes por medio de **XSynchronize()** llamado inmediatamente después de la llamada para conectarse al servidor (**XOpenDisplay**()).

Uno de los aspectos más difíciles de depurar es el manejo de eventos.
Por tal razón se recomienda incluir **printf** al comienzo de cada ramificación de manejo de eventos durante el desarrollo.
La impresión se puede incluir dentro de un **#ifdef DEBUG**, **#endif**.

2.6.3.1 Errores

Hay tres niveles de manejo de errores para programas usando Xlib.

1. El primer nivel se implementa por el usuario monitoreando las devoluciones de las rutinas que crean recursos del servidor.
2. El segundo nivel es de errores de protocolo, normalmente causado por errores de programación.
3. El tercer nivel es un error fatal del sistema tal como la caída de una máquina corriendo el servidor o una falla de la red.

Los últimos dos tipos se manejan por dos funciones separadas que puede asignar el cliente, pero por omisión, simplemente imprimen un mensaje y terminan el programa.

Como ejemplo del primer nivel de manejo de errores, el cliente debe siempre verificar que hubo una conexión exitosa con el servidor de despliegue con **XOpenDisplay()** antes de proceder.
Estos errores se deben imprimir con un mensaje a *stderr*.

Errores del protocolo X ocurren cuando los argumentos de una rutina no corresponden a los rangos aceptados o cuando IDs no corresponden a recursos existentes, etc.
Este tipo de errores se envía a **XErrorHandler**.
Errores fatales, tales como conexionas rotas con el servidor, son condiciones irrecuperables e invocan **XIOErrorHandler**.
Ambos manejadores de error por omisión despliegan un mensaje y luego salen.

Existe también un listado de los errores en la documentación.

Rutinas para manejo de error definidas por el usuario serán llamadas de los manejadores de error si se le pasa nombres de procedimientos a **XSetIOErrorHandler()** o **XSetErrorHandler()**.
Si se le pasa un apuntador de función nulo, el manejador de omisión será utilizado.

Si el usuario escribe sus propias rutinas de manejo de errores, se recomienda usar **XGetErrorText()** o **XGetErrorDatabaseText()** (este último utiliza el manejador de recursos para proveer mensajes de error del archivo *XErrorDB* ubicado por omisión en */usr/lib/X11*) para obtener una cadena describiendo el código de error.

Sólo la rutina de manejo de errores **XErrorHandler** (o la definida por el usuario) recibe eventos de error.
Las ventanas no pueden seleccionar o recibir estos eventos.

2.6.3.2 Estructura de XErrorEvent

La estructura de **XErrorEvent** se muestra a continuación:

```
typedef struct _XErrorEvent {
      int type;
      Display *display; /* Despliegue de dónde se leyó el evento */
      XID resourceid; /* ID del recurso */
      unsigned long serial; /* número de solicitud fallida */
      unsigned char error_code; /* código de error de solicitud fallida */
      unsigned char request_code; /* código mayor de solicitud fallida */
      unsigned char minor_code; /* código menor de solicitud fallida */
} XErrorEvent;
```

La siguiente lista describe cada miembro de la estructura **XErrorEvent** en detalle:

- **serial** es el número de solicitudes enviadas sobre la conexión de la red desde que se abrió comenzando con 1. La diferencia entre **serial** y la última solicitud procesada según se reportó en el mensaje de error nos dice cuantas solicitudes retroceder en ese orden para encontrar la solicitud que causó el error.
- **request_code** es una representación de protocolo del nombre de la solicitud de protocolo que fallo.
- **error_code** puede ser algo como **BadWindow**.
- **minor_code** es cero a menos que la solicitud se parte de una extensión, donde el código indicaría la solicitud en la extensión que causó el error.
- **resource_id** indica uno de los recursos del servidor (ventana, colormap, etc.) que fue asociado con la solicitud que causó el error.

2.6.3.3 Sincronizando Errores

Como los eventos de error no se despliegan precisamente cuando ocurren, es bueno fijarse en la solicitud de protocolo al igual que el código de error para determinar en qué función ocurrió el error.

Es útil utilizar **XSynchronize()** para asegurarse que los errores de protocolo se despliegan de inmediato, aunque con esta instrucción el rendimiento de las gráficas se reduciría considerablemente.
Lo mismo ocurriría asignando la variable global **_Xdebug** a cualquier valor que no sea cero cuando se ejecuta un programa bajo un depurador.

2.6.3.4 Interrupciones de Software

Xlib no maneja interrupciones de software.

Por lo tanto, si se llama recursivamente dentro de Xlib desde un manejador de señales, el programa hará un “hang” o “crash”.
La forma correcta de manejar señales es nunca hacer llamadas a Xlib desde un manejador de señales.

3 Programa Básico de Ventanas

El programa tendrá las siguientes operaciones:

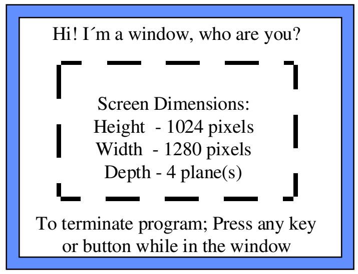
- Conectar el cliente al servidor X con **XOpenDisplay**(), y salir si ésto no se logra.
- Obtener información sobre la pantalla física, y usarla para calcular el tamaño deseado de la ventana.
- Crear una ventana con **XCreateSimpleWindow**().
- Asignar propiedades estándares para el administrador de ventanas.
- Seleccionar los tipos de eventos que necesita recibir.
- Cargar el font a utilizarse para imprimir texto.
- Crear un contexto gráfico para controlar la acción de solicitudes gráficas.
- Desplegar la ventana con **XMapWindow()**.
- Hacer ciclos en espera de eventos.
- Responder al evento **Expose** resultante de mapear la ventana (y cualquier otro evento **Expose** que pueda llegar más tarde), llamando rutinas para dibujar texto y gráficas. Si la ventana es demasiado pequeña para la función deseada, se desplegará un mensaje apropiado.
- Recibir eventos de **ConfigureNotify**, indicando que el administrador de ventanas ha modificado el tamaño de la ventana.
- Seguir manejando eventos hasta que llegue un evento **KeyPress** o **ButtonPress**, entonces cerrar la conexión de despliegue y salir.

El programa no hace ninguna de las siguientes operaciones, que son requeridas por un programa más completo en X:

- Permitir al usuario especificar opciones de línea de comando y leer recursos de la base de datos.
- Manejar colores.

3.1 Corriendo el Programa

El programa simplemente despliega una ventana con algún texto y gráficas dibujadas en ella.
Una de las cosas útiles que muestra es el tamaño y profundidad de la pantalla.
La salida del programa se muestra en la siguiente figura:



3.2 El Main de basicwin

El código se compone del programa “main” y varias subrutinas.
El “main” esta encargado de inicializar todo, excepto crear el contexto gráfico, cargar el font, y dibujar texto y gráficas.
Estas tareas se hacen en las rutinas **draw_graphics**, **draw_text**, **get_GC** y **load_font**.

3.2.1 Declaraciones y Archivos a Incluirse

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <X11/Xatom.h>

#include <stdio.h>

#include "../bitmaps/icon_bitmap"

#define BITMAPDEPTH 1
#define TOO_SMALL 0
#define BIG_ENOUGH 1

/* These are used as arguments to nearly every Xlib routine, so it saves  routine arguments to
declare them global.  If there were additional source files, they would be declared extern there.
*/

Display *display;
int screen_num;

static char *progname; /* name this program was invoked by */

void main(argc, argv)
int argc;
char **argv;
{
        Window win;
        unsigned int width, height;   /* window size */
        int x, y;      /* window position */
        unsigned int border_width = 4;       /* four pixels */
        unsigned int display_width, display_height;
        unsigned int icon_width, icon_height;
        char *window_name = "Basic Window Program";
        char *icon_name = "basicwin";
        Pixmap icon_pixmap;
        XSizeHints *size_hints;
        XIconSize *size_list;
        XWMHints *wm_hints;
        XClassHint *class_hints;
        XTextProperty windowName, iconName;
        int count;
        XEvent report;
        GC gc;
        XFontStruct *font_info;
        char *display_name = NULL;
        int window_size = BIG_ENOUGH; /* or TOO_SMALL to display contents */
```

Los tres archivos de encabezados <X11/Xlib.h>, <X11/Xutil.h>, y <X11/Xos.h> se necesitan virtualmente para todos los programas con Xlib.

El archivo `<X11/Xlib.h>` contiene declaraciones de tipos de estructuras usadas en funciones de Xlib.
El archivo `<X11/Xlib.h>` incluye de por si a `<X11/X.h>` el cual asigna varias constantes definidas.
El archivo `<X11/Xutil.h>` contiene más definiciones de estructuras y constantes para ciertos grupos de funciones en Xlib.
El archivo `<X11/Xos.h>` intenta hacer que los programas sean lo más portátiles posible incluyendo ciertos archivos que dependen del sistema operativo para el cual se compila el programa.
Este archivo de encabezados no es estándar y no es absolutamente necesario, pero es útil.

Los tipos **Window**, **Display**, **Pixmap**, **XSizeHints**, y **XEvent** usados en el programa se definen todos en `<X11/Xutil.h>`.

Window Un identificador (ID) entero único se devuelve de **XCreateWindow()** o **XCreateSimpleWindow()** para ser luego utilizado por el programa para referirse al recurso de la ventana creada.

Display Una estructura grande que contiene información sobre el servidor y las pantallas. Se llena sólo después de que el programa se conecta a un servidor llamando a **XopenDisplay()**.

Pixmap Un entero ID como **Window** pero para un recurso de pixmap. En este caso el pixmap es una figura para desplegar en el icono para la ventana.

XSizeHints Una estructura que se usa para proveer al administrador de ventanas con información sobre los tamaños preferidos e incrementos de tamaño para la ventana de alto nivel de la aplicación.

XEvent Una unión que guarda información sobre un evento. Puede interpretarse como uno de varios tipos individuales de estructura dependiendo del tipo de evento.

3.2.2 Conectando al Servidor

XOpenDisplay() conecta un programa de Xlib con el servidor:

```
char *display_name = NULL;
Display *display;
int screen_num;
Screen *screen_ptr;


/* connect to X server */
if ( (display=XOpenDisplay(display_name)) == NULL )
{
        (void) fprintf( stderr, "%s: cannot connect to X server %s\n",
                progname, XDisplayName(display_name));
        exit( -1 );
}

/* get screen size from display structure macro */
screen_num = DefaultScreen(display);
screen_ptr = DefaultScreenofDisplay(display);
```

El argumento ***display_name*** a **XOpenDisplay()** especifica a que servidor conectarse.
Este puede ser cualquier servidor en la red y puede especificarse en la línea de comandos en una aplicación más completa.

Cuando el usuario no especifica ***display_name*** esta debe ser igual a **NULL**, lo que causa que **XOpenDisplay()** conecte con el servidor especificado en la variable DISPLAY del ambiente UNIX.
Se puede ver el contenido actual de la variable de ambiente DISPLAY utilizando el comando:

```
echo $DISPLAY
```

Puede cambiarse con (C Shell)

```
setenv DISPLAY display_name
```

o (Bourne Shell)

```
DISPLAY=display_name; export DISPLAY
```

Se debe ser cuidadoso de asignar la variable DISPLAY cuando se entra en una máquina remota para asegurarse de que las aplicaciones que se ejecutan de esa terminal, su salida será desplegada en la pantalla donde se escribió el comando.

La variable de ambiente DISPLAY y el argumento *display_name* a **XOpenDisplay()** tienen ambos el mismo formato, *host:server.screen*, donde *host* se refiere al nombre de la máquina corriendo el servidor; *server* es el número de servidor en esa máquina (a veces se refiere a este como el número de despliegue); y *screen* el número de pantalla en esa máquina.

El número de servidor puede verse como el número de usuario en una máquina, siendo siempre cero en una máquina de un sólo usuario y diferente a cero solamente si una misma máquina tiene teclado, apuntador, y despliegue separados para más de un usuario, todos conectados directamente a la misma máquina (no por medio de la red), lo que hoy en día es bastante raro.

La parte de pantalla es opcional y sólo especifica que pantalla se devuelve por el macro **DefaultScreen()**.
Se puede usar cualquiera o todas las pantallas controladas por el servidor especificado.
Por ejemplo, **lamport:0.1** le pide al servidor donde se esta corriendo el programa, conectarse al servidor 0 en la máquina llamada Lamport y que la pantalla de omisión para ese servidor sea la 1. (Por lo general la mayoría de los servidores apoyan una sola pantalla.)

La rutina **XOpenDisplay()** devuelve un apuntador a una estructura de tipo **Display**.
Si la conexión fue exitosa, la estructura será rellenada con información sobre el servidor y cada una de sus pantallas.
Si el intento falla, **XOpenDisplay()** devolverá **NULL**.
El mensaje impreso cuando la conexión falla incluye el texto devuelto por la función **XDisplayName()**, el cual devuelve ***display_name*** o, si es nulo, la variable DISPLAY del ambiente UNIX.
XDisplayName() es necesario ya que sin ello no habría forma de decirle al usuario a que servidor se intentó conectarse.

El cliente puede no lograr conectarse a un servidor por varias razones.
Lo más probable, la variable display_name o la variable de ambiente DISPLAY no especifican un servidor válido que está conectado por medio de la red a la máquina donde se esta ejecutando el programa, o quizás la red esta fuera de servicio.
Otra posibilidad es que el servidor y cliente usan diferentes versiones del protocolo X, ya que la versión X10 no es compatible con la X11 y viceversa, aunque todos los Releases son compatibles para una misma versión.

La conexión también fallaría si la máquina donde se corre el cliente no está en la lista de accesos (*host acces list*) del servidor donde se trata de desplegar.
Esta lista de acceso es un mecanismo para permisos que es leído por el servidor para aceptar conexiones de máquinas incluidas en la lista.

Una variable global adicional es ***screen_num***, asignada al valor devuelto por el macro **DefaultScreen()**.
Es importante utilizar este macro y no definir la variable como cero directamente, para poder asignar el elemento de pantalla en la variable de ambiente DISPLAY.
El número de posibles pantallas se puede obtener con **ScreenCount()**.

3.2.3 Display Macros

Existe un gran número de macros, siendo utilizados en particular para obtener información de estructuras opacas, con miembros que el usuario no debe accesar directamente.

3.2.4 Obteniendo Información de la Ventana

Existen dos formas de obtener información sobre el tamaño de la pantalla:
- Se puede accesar los miembros de la estructura Display para obtener información de la ventana raíz. Este método sólo se puede aplicar a la ventana raíz.
- Se puede usar **XGetGeometry()** o **XGetWindowAttributes()** para obtener las dimensiones de la ventana. Estas llamadas se pueden hacer sobre cualquier ventana.

Para obtener las dimensiones en pixeles de una pantalla, se puede usar los macros **DisplayWidth()** y **DisplayHeight()**.
Los macros **DisplayWidthMM()** y **DisplayHeightMM()** devuelven los valores en milímetros.
La relación entre pixeles y milímetros se puede obtener con estos macros, lo que es importante para ciertos dibujos donde un círculo se pudiera ver como una elipse en pantallas con espacios entre pixeles diferentes para ambas direcciones (por lo general servidores sencillos para la PC).
La información se obtiene localmente de la estructura **Display**, por lo cual es muy eficiente.

La diferencia entre **XGetGeometry()** y **XGetWindowAttributes()** es que la última obtiene mayor información que la primera (realmente utiliza una llamada a la primera).
La información de ambas se obtiene del servidor por lo cual pueden ser ineficientes.
(Para otras ventanas que no sea la raíz, estas dos llamadas son la única forma de obtener la información.)

Utilización del macro:

```
Display *display;
int screen_num;
unsigned int display_width, display_height;


screen_num = DefaultScreen(display);

display_width = DisplayWidth(display, screen_num);
display_height = DisplayHeight(display, screen_num);
```

Utilizando **XGetGeometry**():

```
Display *display;
int screen_num;
Window root;
int x, y;                          /* window position */
unsigned int width, height;   /* window size */
unsigned int border_width;
unsigned int depth;                /* depth */

screen_num = DefaultScreen(display);

if (XGetGeometry(display,RootWindow(display,screen_num),&root,
               &x,&y,&width,&height,&border_width,&depth) == False )
       {
               (void) fprintf( stderr, "%s: can´t get root window geometry\n",
                              progname);
               exit( -1 );
       }
display_width = width;
display_height = height;
```

(El argumento **root** devuelve la ventana raíz en la cima de la jerarquía, lo que no es muy útil ya que esta es la ventana que estamos consultando.)

Utilizando **XGetWindowAttributes**():

```
Display *display;
int screen_num;
XWindowAttributes windowattr;

screen_num = DefaultScreen(display);

if (XGetWindowAttributes(display,RootWindow(display,screen_num),
               &windowattr) == 0 )
       {
               (void) fprintf( stderr, "%s: failed to get window attributes\n",
                              progname);
               exit( -1 );
       }
display_width = windowattr.width;
display_height = windowattr.height;
```

3.2.5 Creando Ventanas

La posición de una ventana relativa a su padre se determina cuando la ventana se crea, ya que las coordenadas se especifican como argumentos a la rutina que crea la ventana.

La aplicación *basicwin* tiene una sola ventana, y su creación es un caso especial ya que esta ventana es hija de la ventana raíz y por lo tanto está sujeta al administrador de ventanas.
Una aplicación puede sugerir una posición para esta ventana, pero es muy posible que sea ignorada.
La mayoría de los administradores de ventanas permiten al usuario posicionar la ventana según aparece en la pantalla, por lo cual la mayoría de las aplicaciones crean la primera ventana con la posición asignada a (0,0).

```
Window win;
unsigned int width, height;   /* window size */
int x, y;      /* window position */
unsigned int border_width = 4;       /* four pixels */


screen_num = DefaultScreen(display);
```

```
/* Note that in a real application, x and y would default to 0 but would be settable from the
command line or resource database.  */
x = y = 0;

/* size window with enough room for text */
width = display_width/3, height = display_height/4;

/* create opaque window */
win = XCreateSimpleWindow(display, RootWindow(display,screen_num),
              x, y, width, height, border_width, BlackPixel(display,
              screen_num), WhitePixel(display,screen_num));
```

Cada pantalla tiene su propia ventana raíz, y para crear la primera ventana de la aplicación en una pantalla particular, se usa la ventana raíz de esa pantalla como padre.
El ID de la ventana raíz en una pantalla particular se devuelve con el macro **RootWindow()**.
La primera generación de ventanas en una pantalla (conocidas como las ventanas de alto nivel) deben siempre usar este macro para especificar al padre.
XCreateSimpleWindow() crea una nueva ventana dado los argumentos para especificar el padre, tamaño, posición, ancho de borde, valor de pixel de borde, y valor de pixel de fondo.
Todos los demás atributos de la ventana se toman del padre, en este caso la ventana raíz.
Si quisiéramos especificar algún o todos los atributos en lugar de heredarlos de la ventana padre, utilizaríamos **XCreateWindow()** en lugar.

3.2.6 Estrategia de Color

Las aplicaciones no escogen valores de pixeles, escogen colores y se les devuelve valores de pixeles por medio de la rutina que se usa para asignar colores o reciben valores de pixeles por medio de los macros de despliegue **BlackPixel()** y **WhitePixel()**.

Este ejemplo es de una aplicación monocroma, pero trabajará igual en pantallas a color.
Se utiliza **WhitePixel()** para especificar el valor de pixel del fondo (en la llamada para crear la ventana) y se asigna el color de primer plano en el GC para que sea el valor contrastante devuelto por **BlackPixel()**.
El valor de pixel del borde se asigna también a **BlackPixel()**.
Los valores de pixeles del fondo y borde se asignan con los dos últimos argumentos de **XCreateSimpleWindow()**.
El valor del pixel de primer plano se asigna también con **get_GC()**.

BlackPixel() y **WhitePixel()** devuelven valores de pixeles que serán traducidos por el colormap en los colores que serán desplegados en la pantalla, los cuales serán colores contrastantes, no necesariamente blanco y negro.

3.2.7 Preparando un Pixmap para el Icono

Una aplicación debería crear un diseño de icono propio, para que si un administrador de ventanas esta corriendo y el usuario iconifica la aplicación, el icono se reconocerá perteneciendo a la aplicación particular.

Para crear un pixmap, el programa debe llevar a cabo dos pasos:

1. Debe encontrar qué tamaños de iconos son aceptables para el administrador de ventanas, y
2. Crear un pixmap del tamaño apropiado.

Como la mayoría de los administradores de ventana no especifican tamaños de iconos, no es necesario consultar tales tamaños, pero si lo fuera, se utilizaría la llamada **XGetIconSizes()** para determinar los tamaños.

```
void main(argc, argv)
int argc;
char **argv;
{
       Pixmap icon_pixmap;

       /* Get available icon sizes from Window manager */
       if (XGetIconSizes(display, RootWindow(display,screen_num),
                     &size_list, &count) == 0)
              (void) fprintf( stderr, "%s: Window manager didn't set icon sizes -
                      using default.\n", progname);
       else {  ;
/* A real application would search through size_list here to find an acceptable icon size, and
then create a pixmap of that size.  This requires that the application have data for several
sizes of icons. */
       }

       /* Create pixmap of depth 1 (bitmap) for icon */
       icon_pixmap = XCreateBitmapFromData(display, win, icon_bitmap_bits,
                     icon_bitmap_width, icon_bitmap_height);
```

Un diseño de icono puede crearse utilizando la aplicación *bitmap* del estándar X.
Se ejecuta *bitmap* con el nombre de un archivo y dimensiones en la línea de comando:

```
% bitmap icon_bitmap 40x40
```

Entonces se usa el apuntador para dibujar el bitmap.
bitmap crea una archivo ASCII que guarda la estructura.

3.2.8 Comunicándose con el Administrador de Ventanas

Antes de mapear la ventana (lo cual la despliega en la pantalla), una aplicación debe asignar propiedades estándares para decirle al administrador de ventanas por lo menos algunas cosas esenciales sobre la aplicación.

Una propiedad es un conjunto de información que se puede leer y escribir por cualquier cliente y normalmente se usa para comunicarse entre clientes.
Las propiedades estándares son parte de la convención para comunicación entre cada aplicación y el administrador de ventanas.
Una propiedad se asocia con una ventana particular.
Las propiedades estándares se asocian con la ventana de más alto nivel de la aplicación.
Es así como el servidor lleva el control sobre las propiedades estándares de las diferentes aplicaciones y las tiene listas para que el administrador de ventanas las lea.

Varias rutinas permiten a la aplicación leer estas propiedades, al igual que al administrador de ventanas.
La rutina diseñada para asignar las propiedades más importantes para las aplicaciones normales es **XSetWMProperties()**.

El conjunto mínino de propiedades que una aplicación debe asignar son las siguientes:

- Nombre de la ventana
- Nombre del icono
- Pixmap del icono
- Nombre de comando y argumentos (la línea de comandos)
- Número de argumentos
- Tamaños de ventana preferidas
- Modelo de foco de teclado

El código que asigna las propiedades estándares se muestran a continuación:

```
void main(argc, argv)
int argc;
char **argv;
{
        char *window_name = "Basic Window Program";
        char *icon_name = "basicwin";
        Pixmap icon_pixmap;
        XSizeHints *size_hints;
        XIconSize *size_list;
        XWMHints *wm_hints;
        XClassHint *class_hints;
        XTextProperty windowName, iconName;

        if (!(size_hints = XAllocSizeHints())) {
                fprintf(stderr, "%s: failure allocating memory\n", progname);
                 exit(0);
        }
        if (!(wm_hints = XAllocWMHints())) {
                fprintf(stderr, "%s: failure allocating memory\n", progname);
                        exit(0);
        }
        if (!(class_hints = XAllocClassHint())) {
                fprintf(stderr, "%s: failure allocating memory\n", progname);
                        exit(0);
        }


/* Set size hints for window manager.  The window manager may override these settings.  Note that
in a real application if size or position were set by the user the flags would be UPosition and
USize, and these would override the window manager's preferences for this window. */

/* x, y, width, and height hints are now taken from the actual settings of the window when
mapped. Note that PPosition and PSize must be specified anyway. */

        size_hints->flags = PPosition | PSize | PMinSize;
        size_hints->min_width = 300;
        size_hints->min_height = 200;

/* These calls store window_name and icon_name into XTextProperty structures and set their other
fields properly. */
        if (XStringListToTextProperty(&window_name, 1, &windowName) == 0) {
                (void) fprintf( stderr, "%s: structure allocation for
                        windowName failed.\n", progname);
                exit(-1);
        }

        if (XStringListToTextProperty(&icon_name, 1, &iconName) == 0) {
                (void) fprintf( stderr, "%s: structure allocation for iconName
                         failed.\n", progname);
                exit(-1);
        }

        wm_hints->initial_state = NormalState;
        wm_hints->input = True;
        wm_hints->icon_pixmap = icon_pixmap;
        wm_hints->flags = StateHint | IconPixmapHint | InputHint;

        class_hints->res_name = progname;
        class_hints->res_class = "Basicwin";

        XSetWMProperties(display, win, &windowName, &iconName,
                        argv, argc, size_hints, wm_hints,
```

```
                    class_hints);
```

Es importante ver que estas propiedades son sólo insinuaciones, o sea, información que puede o no usarse.
Puede que no haya un administrador de ventanas corriendo, o que simplemente el administrador de ventanas ignore algunos o todos los hints.
Por ejemplo, en el caso del hint del nombre de la ventana, algunos administradores de ventanas usarán esta información para desplegar una barrar de título (titlebar) sobre o al lado de cada ventana de alto nivel, mostrando el nombre de la aplicación.
Lo apropiado para la aplicación sería hacer que el nombre de la ventana sea el nombre de la aplicación.
Pero si la aplicación fuera un editor, trataría de asignar el nombre de la ventana al nombre del archivo actual.

Los hints del tamaño de ventana es una estructura que especifica el tamaño, posición, y relaciones de aspectos preferidos por la aplicación.

La estructura para `XSizeHints` es la siguiente:

```
typedef struct {
      long flags; /* Marca campo definido en esta estructura */
      int x,y;    /* Obsoleto desde R4 */
      int width, height;      /* Obsoleto desde R4 */
      int min_width, min_height;
      int max_width, max_height;
      int width_inc, height_inc;
      struct {
            int x;      /* Numerador */
            int y;      /* Denominador */
      } min_aspect, max_aspect;
      int base_width, base_height;  /* Nuevo en R4 */
      int win_gravity;              /* Nuevo en R4 */
} XSizeHints;
```

Todo esto organiza una prioridad para las diferentes asignaciones de la posición y tamaño de la ventana de alto nivel.
La prioridad más bajo es la aplicación (**PSize** y **PPosition**).
La próxima prioridad más alta la tiene el administrador de ventanas.
La prioridad más alta de todas la tiene el usuario (**USize** y **UPosition**).

3.2.9 Seleccionando los Tipos de Eventos Deseados

Este programa recibe eventos por tres razones:
- Para redibujarse en caso de exponerse.
- Para recalcular sus contenido cuando se cambia su tamaño.
- Para recibir una presión de botón o tecla indicando que el usuario terminó con la aplicación.

El programa debe seleccionar estos tres tipos de eventos explícitamente, ya que por omisión, no recibirá ninguno de estos tipos de entrada.

La máscara para selección de eventos se define de la siguiente manera:

```
/* Select event types wanted */
XSelectInput(display, win, ExposureMask | KeyPressMask |
                ButtonPressMask | StructureNotifyMask);
```

El argumento crucial de **XSelectInput()** es la máscara de eventos.
Cada símbolo usado aquí selecciona uno o más tipo de eventos.
Las constantes de la máscara de eventos se combinan con un OR de bits ya que asignan los bits de un sólo argumento.

ExposureMask selecciona eventos **Expose**, que ocurren cuando la ventana se despliega por primera vez y cuando se hace visible después de haber estado oscurecida, señalizando que la aplicación debe redibujarse.

X provee eventos separados para presionar y soltar las teclas del teclado y los botones del apuntador y símbolos separados para seleccionar cada uno de estos tipos de eventos. (**KeyPreeMask**, **KeyReleaseMask**, **ButtonPressMask**, **ButtonReleaseMask**.)

StructureNotifyMask selecciona una cantidad de tipos de eventos, especificamente **CirculateNotify**, **ConfigureNotify**, **DestroyNotify**, **GravityNotify**, **MapNotify**, **ReparentNotify**, y **UnmapNotify**.
La única que se necesita en esta aplicación es **ConfigureNotify**, que informa a la aplicación del nuevo tamaño de la ventana cuando se le haya cambiado el tamaño.
Sin embargo, no hay forma de seleccionar sólo este tipo de evento.
Sin este tipo de evento, en cada evento **Expose** la aplicación tendría que usar **XGetGeometry()** para saber el tamaño actual, lo cual requiere una respuesta del servidor sujeta a retardos de la red.

XSelectInput() realmente asigna el atributo event_Mask de la ventana.
Si se crea la ventana con **XCreateWindow**(), en lugar de **XCreateSimpleWindow()**, se puede seleccionar eventos simultaneamente con la asignación del atributo **event_Mask** en los últimos dos argumentos de la llamada.
Esto es un poco más eficiente que llamar **XSelectInput()** separadamente.
También se puede asignar este atributo con **XChangeWindowAttributes**().

3.2.10 Creando Recursos del Servidor

Recursos del servidor son conjuntos de información manejados por el servidor y referidos en la aplicación por medio de un número ID.
Artículos con los tipos **ColorMap**, **Cursor**, **Font**, **GC**, **Pixmap**, y **Window** son recursos del servidor.
Deben crearse una vez y el ID guardado en lugar de crear y destruirlos frecuentemente.

En este programa ya se han creado dos recursos, una ventana y el pixmap del icono.
Aun se requiere cargar un font para el texto y crear un contexto gráfico para dibujar texto y gráficos en la ventana.

```
load_font(&font_info);
getGC(win, &gc, font_info); /* create GC for text and drawing */
```

Estas operaciones se hacen en las rutinas **load_font** y **get_GC** llamadas antes de mapear la ventana.

3.2.11 Mapeando la Ventana

Finalmente se está listo para desplegar la ventana.
Previamente los hints del administrador de ventanas se asignaron para que este pueda manejar el mapeo apropiadamente, y los eventos deben seleccionarse para que el primer Expose le diga a la aplicación dibujar dentro de ella.

El mapeo se hace de la siguienete forma:

```
XMapWindow(display, win); /* Display window */
```

Se repiten nuevamente las condiciones para hacer visible una ventana:

1. La ventana debe mapearse con **XMapWindow()** o rutinas similares.
2. Todos los ancestros de la ventana deben estar mapeados. Esta condición siempre se satisface para los hijos de la ventana raíz, las ventanas de alto nivel para cada aplicación.
3. La ventana no debe estar oscurecida por hermanos o ancestros visibles, dependiendo del ordenamiento. Cuando se hace el primer mapeo, la ventana aparece sobre todos sus hermanos, lo que será sobre todas las demás ventanas si su padre es la ventana raíz.
4. El buffer de solicitudes debe haberse vaciado.
5. El mapeo inicial de una ventana de alto nivel es un caso especial, ya que la visibilidad de la ventana puede retardarse por el administrador de ventanas por razones de redirección de subestructuras. Por razones complicadas el cliente debe esperar el primer evento **Expose** antes de asumir que su ventana está visible y se le puede dibujar.

3.2.12 Vaciando el Buffer de Salida

XMapWindow() causa una solicitud al protocolo X que instruye al servidor desplegar la ventana en la pantalla.
Como todas las demás solicitudes del protocolo X, ésa se pone en la cola hasta que haya una rutina que lea eventos como **XNextEvent()**, una rutina que consulta con el servidor, o una rutina como **XFlush()** o **XSync()**.

La llamada **XNextEvent()** hace el vaciado del buffer suficientemente frecuente en las aplicaciones que toman entrada de usuario.
Las rutinas que consultan con el servidor deben llamarse con la menor frecuencia posible ya que reducen el rendimiento sobre la red.
El comando **XFlush()** instruye al servidor procesar todos las solicitudes en la cola de inmediato, lo cual es generalmente necesario cuando una aplicación requiere dibujar periodicamente inclusive sin entrada de usuario.

3.2.13 Asignando el Ciclo de Recolección de Eventos

Los programas en X están dirigidos por eventos, significando que luego de asignar todos los recursos del servidor y los hints del administrador de ventanas, el programa ejecutará las acciones posteriores sólo como respuesta a eventos.

El ciclo de recolección de eventos es la forma estándar de responder a eventos, ejecutando las acciones apropiadas dependiendo del tipo de eventos y la información contenida en la estructura de eventos.

El ciclo de eventos es normalmente un ciclo cerrado, donde uno de los tipos de eventos se utiliza para salir de la aplicación.
En algunas aplicaciones como el *xclock* el ciclo está completamente cerrado, y la única forma de terminar el programa es encontrar el ID del proceso y destruirlo desde un 'shell'.

La selección de que eventos se reciben se hicieron con la asignación del atributo **event_Mask**.
El ciclo de eventos debe asegurarse que manejará apropiadamente cada tipo de evento seleccionado.

```
	while (1)  { /* get events, use first to display text and graphics */
		XNextEvent(display, &report);
		switch  (report.type) {
		case Expose:
	/* unless this is the last contiguous expose,don't draw the window */
			if (report.xexpose.count != 0)
				break;

			if (window_size == TOO_SMALL) /* if window too small to use */
				TooSmall(win, gc, font_info);
			else { /* place text in window */
				draw_text(win, gc, font_info, width, height);
				/* place graphics in window, */
				draw_graphics(win, gc, width, height);
			}
			break;
		case ConfigureNotify:
/* window has been resized, change width and height to send to draw_text and draw_graphics in next Expose */
			width = report.xconfigure.width;
			height = report.xconfigure.height;
			if ((width < size_hints->min_width) ||
					(height < size_hints->min_height))
				window_size = TOO_SMALL;
			else
				window_size = BIG_ENOUGH;
			break;
		case ButtonPress: /* trickle down into KeyPress (no break) */
		case KeyPress:
			XUnloadFont(display, font_info->fid);
			XFreeGC(display, gc);
			XCloseDisplay(display);
			exit(1);
		default:
/* all events selected by StructureNotifyMask except ConfigureNotify are thrown away here, since nothing is done with them */
			break;
		} /* end switch */
	} /* end while */
```

Este es un ciclo infinito, con la instrucción **XNextEvent()** dentro del ciclo, obteniendo una estructura de eventos de la cola que Xlib mantiene para la aplicación poniendo el apuntador a ella en la variable ***report***.
XNextEvent() devuelve void, regresando sólo cuando hay un evento para devolver.

Los errores se manejan por medio de mecanismos de manejo de errores separados, no a través del valor de respuesta.
A continuación de **XNextEvent()** hay un switch que se ramifica dependiendo del tipo de evento. Existe un caso para cada uno de los cuatro tipos de eventos: **ButtonPress**, **ConfigureNotify**, **Expose** y **KeyPress**.

La rama de **ConfigureNotify** calculará los valores de las variables basados en los nuevos tamaños de ventanas, los cuales serán luego usados para calcular donde dibujar las cosas en la rama de **Expose** la siguiente vez que un evento **Expose** ocurre.
Un evento **ConfigureNotify** siempre está seguido por uno o más eventos **Expose**.

3.2.13.1 Pintando de Nuevo la Ventana

Eventos **Expose** ocurren cuando una ventana se hace visible en la pantalla, luego de haberse oscurecido o desmapeado, y por la razón de que X Windows no guarda normalmente el contenido de las regiones de ventanas oscurecidas por otras ventanas o no mapeadas.
El contenido de las ventanas debe redibujarse cuando se exponen.

El código para eventos **Expose** dibuja o redibuja los contenidos de las ventanas de la aplicación, el cual será hecho cuando la ventana se mapea por primera vez, y cuando una porción de la ventana se hace visible.

Una aplicación puede responder a eventos **Expose** refrescando sólo las partes de la ventana expuestas o refrescando la ventana completa.

```
typedef struct {
      int type;
      unsigned long serial; /* # de última solicitud procesada */
      Bool send_event; /* True si vino de una solicitud de SendEvent */
      Display *display; /* Despliegue de donde se leyo el evento */
      Window window;
      int x,y;
      int width, height;
      int count;  /* Si no es cero, se enviará eventos adicionales de
            Expose a continuación un número 'count' de veces */
} XExposeEvent;
```

Varios eventos **Expose** pueden ocurrir de una sola operación del administrador de ventanas, e.g. en el caso que una ventana oculte porciones de varias otras.

Se debe considerar dos componentes en el tiempo de redibujar:

- el tiempo que le toma a la aplicación procesar las instrucciones de redibujar, y
- el tiempo que le toma al servidor realmente redibujar.

En la mayoría de los servidores, el usuario debe esperar a que el servidor termine de dibujar antes de poder ir a otras acciones.
Por lo tanto, el tiempo tomado por el servidor es crítico, ya que se traduce directamente en espera del usuario.
Como el sistema corriendo clientes de X es normalmente multitareas, el tiempo tomado por la aplicación para minimizar el redibujo no es tan importante, ya que el usuario aun puede trabajar.

Existen dos enfoques para asistir al servidor en redibujar rápidamente regiones expuestas.
Una es evitar redibujar artículos en regiones que no hayan sido expuestas.
Haciendo esto en una aplicación requiere identificar cualquier artículo a dibujarse que no se extiende en ninguna de las áreas expuestas y eliminar estas solicitudes de dibujo.
Existen un conjunto de rutinas para ayudar a esto.

Las reglas generales para redibujar son:

- Si se dibuja la ventana rápido, la ventana completa puede dibujarse en respuesta al último evento **Expose** de una serie contigua; o sea dibujando sólo cuando **count** es cero. La definición de *rápido* variará de servidor en servidor, pero cualquier cosa que use los aspectos más complejos del GC, tal como líneas anchas, o que pueda tener cantidad de solicitudes de dibujo debería posiblemente considerarse como lento.
- Para ventanas que son lentas de dibujar, la aplicación debe evitar dibujar áreas que no se hayan expuesto. Si la aplicación puede saber que solicitudes lentas de dibujo se dibujarán sólo en áreas que no están expuestas y estos cálculos no toman demasiado tiempo, entonces estas solicitudes deben eliminarse.
- Para ventanas que son lentas de dibujar, el segundo mejor enfoque es asignar un “clip Mask” que permita al servidor eliminar solicitudes innecesarias. (Esto funcionara sólo si el servidor se ha diseñado para ello.) La aplicación puede combinar todas las áreas en una serie contigua de eventos de exposición en una sola máscara de clip y asignarla al GC. Este enfoque es sólo un poco más complejo que el anterior.

Como la aplicación de basicwin es sencilla se puede redibujar la ventana completa al recibir el último evento contiguo de **Expose** (como en la sección de código anterior para la rama **Expose**).
El segundo caso es difícil de aplicar y generalizar para los dibujos genéricos. (Una aplicación como una hoja de cálculo es más fácil aplicar ya que las ventanas están subdivididas logicamente.)
El tercer método, el más complejo, se muestra a continuación:

```
int window_size = BIG_ENOUGH; /* O TOO_SMALL */
Region  region; /* Integracion de rectangulos de todos los eventos de expo */
XRectangle rectangle; /* Poner rectangulos de expo aqui */

region = XCreateRegion(); /* Crear region para procesamiento de eventos de expo */


while (1)  { /* get events, use first to display text and graphics */
        XNextEvent(display, &report);
        switch  (report.type) {
        case Expose:
        /* unless this is the last contiguous expose,don't draw the window */
                if (window_size == TOO_SMALL) { /* if window too small to use */
                        TooSmall(win, gc, font_info);
                        break;
                }
                rectangle.x = (short) report.xexpose.x;
                rectangle.y = (short) report.xexpose.y;
                rectangle.width = (unsigned short) report.xexpose.width;
                rectangle.height = (unsigned short) report.xexpose.height;

                XUnionRectWithRegion(&rectangle, region, region);

                if (report.xexpose.count == 0) {
                        XSetRegion(display,gc,region); /* set clip region */
                        XDestroyRegion(region); /* clear region for next time */
                        region = XCreateRegion();

                        /* place text in window, */
                        draw_text(win, gc, font_info, width, height);
```

```
                /* place graphics in window, */
                draw_graphics(win, gc, width, height);
        }
        break;
```

Poder redibujar los contenidos de sus ventanas es importante para la mayoría de las aplicaciones, pero para algunas aplicaciones, puede ser muy difícil o imposible.
Hay otro método que puede ser usado en tal situación.
La aplicación puede dibujar en un pixmap y luego copiar el pixmap a la ventana cada vez que la ventana requiere redibujar.
De esa forma el contenido completo de la ventana estaría siempre listo para redibujarse según eventos **Expose**.
La desventaja de este enfoque es que el servidor puede que no tenga suficiente memoria para guardar muchos pixmaps en memoria (especialmente en despliegues a color) o puede que sea lento en copiar los pixmaps a la ventana.
Pero este seria un enfoque lógico para manejar exposiciones en aplicaciones que hacen doble buffering (una técnica de animación que oculta el proceso de dibujar, donde pixmaps se utilizan para dibujar y luego se copian cuando la imagen está completa; otra técnica es superposiciones (overlays)).
En estaciones de trabajo de gran rendimiento, puede que exista almacenamientos de respaldo para asistir el redibujo de las ventanas.
El único aspecto que no se apoyaría sería el cambio de tamaño de la ventana, que requeriría un redibujo completo.

3.2.14 Cuando se puede dibujar ?

No se puede dibujar inmediatamente después de una solicitud de **XMapWindow()**, sólo hasta después de recibir el primer evento de **Expose**.

3.2.15 Cuando Aparecerán los Dibujos ?

Los dibujos sólo aparecerán cuando se vacía el buffer de solicitudes y se envía al servidor.

3.2.16 Manejando Cambio de Tamaño de la Ventana

El evento **ConfigureNotify** le dice a la aplicación que cambió el tamaño de la ventana.
En este programa, se pasa información a las rutinas que dibujan, para que puedan ubicar las cosas apropiadamente.
También se verifica si el tamaño de la ventana es menor que del mínimo deseado.

```
        case ConfigureNotify:
/* window has been resized, change width and height to send to draw_text and draw_graphics in next Expose */
                width = report.xconfigure.width;
                height = report.xconfigure.height;
                if ((width < size_hints->min_width) ||
                                (height < size_hints->min_height))
                        window_size = TOO_SMALL;
                else
                        window_size = BIG_ENOUGH;
                break;
```

Cuando al ventana se mapea por primera vez, el evento **ConfigureNotify** aparece en la cola *antes* del primer evento de **Expose**.
Esto significa que el código trabaja inclusive si el administrador de ventanas modifica el tamaño de la ventana antes de permitir que se despliegue.
El **ConfigureNotify** inicial actualiza el tamaño de la ventana con la aplicación permitiendo que el siguiente evento **Expose** dibuje los contenidos.

Si no se escogiera el evento **ConfigureNotify** entonces el código **Expose** debería ser modificada para verificar los tamaños, teniendo que consultar con el servidor los tamaños cada vez que un evento **Expose** se envían, ya que los eventos describen sólo el área expuesta, no la ventana completa.

3.2.17 Saliendo del Programa

Este programa usa la presión de una tecla o botón para salir.
El programa debería liberar toda la memoria al salir, particularmente los recursos de X, y luego cerrar la conexión de despliegue con **XCloseDisplay()**.

```
case ButtonPress: /* trickle down into KeyPress (no break) */
case KeyPress:
        XUnloadFont(display, font_info->fid);
        XFreeGC(display, gc);
        XCloseDisplay(display);
        exit(1);
```

Es una buena practica utilizar **XCloseDisplay()** aunque la conexión al servidor se cierre automaticamente cuando un proceso sale.
De lo contrario, errores pendientes puede que no sean reportados.

3.2.18 Manejo de Errores

El manejo de errores para el programa es el siguiente:

- En la llamada a XOpenDisplay(), se verifica el valor de retorno.
- Para el resto de los errores, se depende en los mecanismos de omisión de manejo de errores: errores de protocolo causados por errores de programación, por el servidor que se quede sin memoria, o error del IO como pérdida de la conexión.

No todos los errores de protocolo causan que se invoque el manejador de errores.
Algunos errores como falla en abrir un font se indican por valores de tipo **Status**, siendo cero en caso de error y diferente a cero si tiene éxito.

3.2.19 Código Completo para basicwin

```
/*      basicwin.c      */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <X11/Xatom.h>

#include <stdio.h>

#include "../bitmaps/icon_bitmap"
```

```
#define BITMAPDEPTH 1
#define TOO_SMALL 0
#define BIG_ENOUGH 1

/* These are used as arguments to nearly every Xlib routine, so it saves  routine arguments to
declare them global.  If there were additional source files, they would be declared extern there.
*/

Display *display;
int screen_num;

static char *progname; /* name this program was invoked by */

void main(argc, argv)
int argc;
char **argv;
{
	Window win;
	unsigned int width, height;	/* window size */
	int x, y;	/* window position */
	unsigned int border_width = 4;	/* four pixels */
	unsigned int display_width, display_height;
	unsigned int icon_width, icon_height;
	char *window_name = "Basic Window Program";
	char *icon_name = "basicwin";
	Pixmap icon_pixmap;
	XSizeHints *size_hints;
	XIconSize *size_list;
	XWMHints *wm_hints;
	XClassHint *class_hints;
	XTextProperty windowName, iconName;
	int count;
	XEvent report;
	GC gc;
	XFontStruct *font_info;
	char *display_name = NULL;
	int window_size = BIG_ENOUGH; /* or TOO_SMALL to display contents */

	progname = argv[0];

	if (!(size_hints = XAllocSizeHints())) {
		fprintf(stderr, "%s: failure allocating memory\n", progname);
		 exit(0);
	}
	if (!(wm_hints = XAllocWMHints())) {
		fprintf(stderr, "%s: failure allocating memory\n", progname);
			exit(0);
	}
	if (!(class_hints = XAllocClassHint())) {
		fprintf(stderr, "%s: failure allocating memory\n", progname);
			exit(0);
	}

	/* connect to X server */
	if ( (display=XOpenDisplay(display_name)) == NULL )
	{
		(void) fprintf( stderr, "%s: cannot connect to X server %s\n",
				progname, XDisplayName(display_name));
		exit( -1 );
	}

	/* get screen size from display structure macro */
	screen_num = DefaultScreen(display);
	display_width = DisplayWidth(display, screen_num);
	display_height = DisplayHeight(display, screen_num);

/* Note that in a real application, x and y would default to 0 but would be settable from the
command line or resource database.  */
	x = y = 0;

	/* size window with enough room for text */
	width = display_width/3, height = display_height/4;

	/* create opaque window */
	win = XCreateSimpleWindow(display, RootWindow(display,screen_num),
			x, y, width, height, border_width, BlackPixel(display,
			screen_num), WhitePixel(display,screen_num));

	/* Get available icon sizes from Window manager */
	if (XGetIconSizes(display, RootWindow(display,screen_num),
			&size_list, &count) == 0)
```

```
		(void) fprintf( stderr, "%s: Window manager didn't set icon sizes -
			 using default.\n", progname);
	else {  ;
/* A real application would search through size_list here to find an acceptable icon size, and
then create a pixmap of that size.  This requires that the application have data for several
sizes of icons. */
	}

	/* Create pixmap of depth 1 (bitmap) for icon */
	icon_pixmap = XCreateBitmapFromData(display, win, icon_bitmap_bits,
			icon_bitmap_width, icon_bitmap_height);

/* Set size hints for window manager.  The window manager may override these settings.  Note that
in a real application if size or position were set by the user the flags would be UPosition and
USize, and these would override the window manager's preferences for this window. */

/* x, y, width, and height hints are now taken from the actual settings of the window when
mapped. Note that PPosition and PSize must be specified anyway. */

	size_hints->flags = PPosition | PSize | PMinSize;
	size_hints->min_width = 300;
	size_hints->min_height = 200;

/* These calls store window_name and icon_name into XTextProperty structures and set their other
fields properly. */
	if (XStringListToTextProperty(&window_name, 1, &windowName) == 0) {
		(void) fprintf( stderr, "%s: structure allocation for
			windowName failed.\n", progname);
		exit(-1);
	}

	if (XStringListToTextProperty(&icon_name, 1, &iconName) == 0) {
		(void) fprintf( stderr, "%s: structure allocation for iconName
			 failed.\n", progname);
		exit(-1);
	}

	wm_hints->initial_state = NormalState;
	wm_hints->input = True;
	wm_hints->icon_pixmap = icon_pixmap;
	wm_hints->flags = StateHint | IconPixmapHint | InputHint;

	class_hints->res_name = progname;
	class_hints->res_class = "Basicwin";

	XSetWMProperties(display, win, &windowName, &iconName,
			argv, argc, size_hints, wm_hints,
			class_hints);

	/* Select event types wanted */
	XSelectInput(display, win, ExposureMask | KeyPressMask |
			ButtonPressMask | StructureNotifyMask);

	load_font(&font_info);
	getGC(win, &gc, font_info); /* create GC for text and drawing */
	XMapWindow(display, win); /* Display window */

	while (1)  { /* get events, use first to display text and graphics */
		XNextEvent(display, &report);
		switch  (report.type) {
		case Expose:
	/* unless this is the last contiguous expose,don't draw the window */
			if (report.xexpose.count != 0)
				break;

			if (window_size == TOO_SMALL) /* if window too small to use */
				TooSmall(win, gc, font_info);
			else { /* place text in window */
				draw_text(win, gc, font_info, width, height);
				/* place graphics in window, */
				draw_graphics(win, gc, width, height);
			}
			break;
		case ConfigureNotify:
/* window has been resized, change width and height to send to draw_text and draw_graphics in
next Expose */
			width = report.xconfigure.width;
			height = report.xconfigure.height;
			if ((width < size_hints->min_width) ||
					(height < size_hints->min_height))
				window_size = TOO_SMALL;
			else
```

```
                        window_size = BIG_ENOUGH;
                    break;
            case ButtonPress: /* trickle down into KeyPress (no break) */
            case KeyPress:
                    XUnloadFont(display, font_info->fid);
                    XFreeGC(display, gc);
                    XCloseDisplay(display);
                    exit(1);
            default:
/* all events selected by StructureNotifyMask except ConfigureNotify are thrown away here, since
nothing is done with them */
                    break;
            } /* end switch */
    } /* end while */
}

getGC(win, gc, font_info)
Window win;
GC *gc;
XFontStruct *font_info;
{
    unsigned long valueMask = 0; /* ignore XGCvalues and use defaults */
    XGCValues values;
    unsigned int line_width = 6;
    int line_style = LineOnOffDash;
    int cap_style = CapRound;
    int join_style = JoinRound;
    int dash_offset = 0;
    static char dash_list[] = {12, 24};
    int list_length = 2;

    /* Create default Graphics Context */
    *gc = XCreateGC(display, win, valueMask, &values);

    /* specify font */
    XSetFont(display, *gc, font_info->fid);

/* specify black foreground since default window background is white and default foreground is
undefined. */
    XSetForeground(display, *gc, BlackPixel(display,screen_num));

    XSetLineAttributes(display, *gc, line_width, line_style,
                    cap_style, join_style); /* set line attributes */

    /* set dashes */
    XSetDashes(display, *gc, dash_offset, dash_list, list_length);
}

load_font(font_info)
XFontStruct **font_info;
{
    char *fontname = "9x15";

    /* Load font and get font information structure. */
    if ((*font_info = XLoadQueryFont(display,fontname)) == NULL)
    {
            (void) fprintf( stderr, "%s: Cannot open 9x15 font\n",
                        progname);
            exit( -1 );
    }
}

draw_text(win, gc, font_info, win_width, win_height)
Window win;
GC gc;
XFontStruct *font_info;
unsigned int win_width, win_height;
{
    char *string1 = "Hi! I'm a window, who are you?";
    char *string2 = "To terminate program; Press any key";
    char *string3 = "or button while in this window.";
    char *string4 = "Screen Dimensions:";
    int len1, len2, len3, len4;
    int width1, width2, width3;
    char cd_height[50], cd_width[50], cd_depth[50];
    int font_height;
    int initial_y_offset, x_offset;

    /* need length for both XTextWidth and XDrawString */
    len1 = strlen(string1);
    len2 = strlen(string2);
    len3 = strlen(string3);
```

```
        /* get string widths for centering */
        width1 = XTextWidth(font_info, string1, len1);
        width2 = XTextWidth(font_info, string2, len2);
        width3 = XTextWidth(font_info, string3, len3);

        font_height = font_info->ascent + font_info->descent;

        /* output text, centered on each line */
        XDrawString(display, win, gc, (win_width - width1)/2,
                        font_height,
                        string1, len1);
        XDrawString(display, win, gc, (win_width - width2)/2,
                        (int)(win_height - (2 * font_height)),
                        string2, len2);
        XDrawString(display, win, gc, (win_width - width3)/2,
                        (int)(win_height - font_height),
                        string3, len3);

        /* copy numbers into string variables */
        (void) sprintf(cd_height, " Height - %d pixels",
                        DisplayHeight(display,screen_num));
        (void) sprintf(cd_width, " Width  - %d pixels",
                        DisplayWidth(display,screen_num));
        (void) sprintf(cd_depth, " Depth  - %d plane(s)",
                        DefaultDepth(display, screen_num));

        /* reuse these for same purpose */
        len4 = strlen(string4);
        len1 = strlen(cd_height);
        len2 = strlen(cd_width);
        len3 = strlen(cd_depth);

/* To center strings vertically, we place the first string so that the top of it is two
font_heights above the center of the window.  Since the baseline of the string is what we need to
locate for XDrawString, and the baseline is one font_info->ascent below the top of the character,
the final offset of the origin up from the center of the window is one font_height + one descent.
*/

        initial_y_offset = win_height/2 - font_height - font_info->descent;
        x_offset = (int) win_width/4;
        XDrawString(display, win, gc, x_offset, (int) initial_y_offset,
        string4,len4);

        XDrawString(display, win, gc, x_offset, (int) initial_y_offset +
                        font_height,cd_height,len1);
        XDrawString(display, win, gc, x_offset, (int) initial_y_offset +
                        2 * font_height,cd_width,len2);
        XDrawString(display, win, gc, x_offset, (int) initial_y_offset +
                        3 * font_height,cd_depth,len3);
}

draw_graphics(win, gc, window_width, window_height)
Window win;
GC gc;
unsigned int window_width, window_height;
{
        int x, y;
        int width, height;

        height = window_height/2;
        width = 3 * window_width/4;
        x = window_width/2 - width/2;  /* center */
        y = window_height/2 - height/2;
        XDrawRectangle(display, win, gc, x, y, width, height);
}

TooSmall(win, gc, font_info)
Window win;
GC gc;
XFontStruct *font_info;
{
        char *string1 = "Too Small";
        int y_offset, x_offset;

        y_offset = font_info->ascent + 2;
        x_offset = 2;

        /* output text, centered on each line */
        XDrawString(display, win, gc, x_offset, y_offset, string1,
                        strlen(string1));
}
```