А.П. Полищук, С.А. Семериков

ПРОГРАММИРОВАНИЕ В X WINDOW

А.П. Полищук, С.А. Семериков

Программирование в X Window

Учебное пособие

Кривой Рог
Издательский отдел КГПУ
2003

Полищук А.П., Семериков С.А.

Программирование в X Window: Учебное пособие. – Кривой Рог: Издательский отдел КГПУ, 2003. – 192 с.

Учебное пособие, ориентированное на программирующего пользователя, посвящено методам программирования в системе X Window. Приведен лабораторный практикум по созданию сетевых графических приложений средствами Xlib и Xt. Изложение ориентировано на ОС Linux, однако все приведенные сведения могут быть применены в любой ОС семейства UNIX.

Для студентов высших учебных заведений, аспирантов, научных и инженерно-технических работников.

Рецензенты:

д. т. н., проф. **А.Н. Марюта**
(Днепропетровский национальный университет)
д. ф.-м. н., проф. **В.Н. Соловьёв**
(Институт социального управления, экономики и права)

Содержание

Введение

Операционная система UNIX существует очень давно. Созданная более двадцати лет назад, она прошла в своем развитии несколько стадий, и в настоящее время представляет, пожалуй, наиболее развитую, но вместе с тем простую и элегантную (если не сказать больше) операционную систему. В UNIX есть все: параллельное выполнение многих программ, одновременная работа нескольких пользователей, виртуальная память, поддержка большого количества внешних устройств и сетей, развитые средства обработки текстов, мощные инструментальные средства для создания программного обеспечения. Система работает во всем мире на миллионах компьютеров разных типов.

В нашей стране UNIX был не очень распространен, и тому были свои причины. Во-первых, это существовавшая направленность на использование небольшого количества типов ЭВМ. В основном это были ЕС и СМ, на которых функционировали специально, под конкретную архитектуру разработанные, ОС, такие как ОС ЕС (IBM 360/370), ОС РВ (RSX-11) и РАФОС (RT-11). Во-вторых, созданные у нас во второй половине 80-х версии UNIX (МОС для ЕС, ИНМОС и ДЕМОС для СМ) несколько запоздали. Аппаратура, на которой предполагалась их эксплуатация, морально устарела и в настоящее время практически не используется. В-третьих, до недавнего времени свободно распространяемые версии UNIX (в первую очередь Linux) не имели удобного графического интерфейса, столь привычного пользователям операционных систем семейства Windows.

Графические интерфейсы UNIX имеют давнюю историю. Первые программные разработки в этом направлении появились более 20 лет назад. Стандартом стала распределенная система X Window, которая

позволяет рисовать на экране дисплея графические изображения, поддерживает концепцию окон и унифицирует работу с различными устройствами ввода-вывода на основе библиотеки Xlib. Для того чтобы облегчить программирование с применением Xlib (X11) и упростить создание пользовательских интерфейсов, существует несколько пакетов, из которых наиболее широко распространены X Toolkit Intrinsics (Xt), Athena (Xaw) и Motif (Xm). В последние годы появились два новых пакета: GTK+ и Qt, лежащих в основе популярных среди пользователей Linux графических интерфейсов GNOME и KDE.

Именно о программировании пользовательского интерфейса UNIX в системе X Window и будет идти речь в данной книге.

1. Основы программирования в системе X Window

X Window или просто X – это система для создания графического пользовательского интерфейса на компьютерах, работающих под управлением операционной системы UNIX. X была создана в Массачусетском Технологическом Институте (США). В настоящее время уже выпущена версия 11.6 (X Window System Version 11 Release 6 или X11R6).

Особенностью системы является то, что она поддерживает работу как на отдельной ЭВМ, так и в сети. Это означает, что программа, «живущая» на одном компьютере, может с помощью X Window общаться с пользователем, сидящим за другой машиной. Система обеспечивает вывод графической информации на экран машины, воспринимает сигналы от внешних устройств, таких как клавиатура и мышь, и передает их программам. Заметим, что устройство вывода может иметь несколько экранов. X обеспечивает рисование на любом из них. Все это: экран (или экраны), а также устройства ввода (клавиатура или мышь) называются в терминах X Window *дисплей*.

X позволяет пользователю общаться со многими программами одновременно. Чтобы вывод из них не смешивался, система создает на экране дисплея «виртуальные» подэкраны – *окна*. Каждое приложение, как правило, рисует лишь в своем окне или окнах. X предоставляет набор средств для создания окон, их перемещения по экрану и изменения их размеров.

Как правило, программы имеют набор конфигурационных параметров – *ресурсов*. Это может быть цвет окна, тип шрифта, которым рисуется текст, и многое другое. Система стандартизует способ задания ресурсов приложений и содержит ряд процедур для работы с ними.

Эта совокупность функций называется *менеджером ресурсов* (X recource manager или сокращенно Xrm). «Хранилище» параметров программы называется *базой данных ресурсов*.

Особенностью X Window является то, что она организует общение между самими программами и между программами и внешней средой путем рассылки событий. *Событие* – это единица информации, идентифицирующая происходящие в системе изменения или действия, и содержащая дополнительные сведения о них.

1.1. Основные понятия

1.1.1. Общее устройство X Window

Система X Window представляет совокупность программ и библиотек. Сердцем ее является отдельный UNIX-процесс, существующий на компьютере, к которому присоединен дисплей. Именно *сервер* знает особенности конкретной аппаратуры, знает, что надо предпринять, чтобы закрасить пиксель на экране, нарисовать линию или другой графический объект. Он также умеет воспринимать сигналы, приходящие от клавиатуры и мыши.

Сервер общается с программами-клиентами, посылая или принимая от них порции (пакеты) данных. Если сервер и клиент работают на разных машинах, то данные посылаются по сети, если же компьютер один, то для передачи данных используется внутренний канал. Например, если сервер обнаруживает, что нажата кнопка мыши, то он подготавливает соответствующий пакет и посылает его тому клиенту, в чьем окне находится курсор мыши. И наоборот, если программе надо что-либо вывести на экран дисплея, то она создает необходимый пакет данных и посылает его серверу.

Состав пакетов и их последовательность определяются специальным протоколом. Но чтобы программировать для

X, совсем не обязательно знать детали реализации сервера и протокола обмена. Система предоставляет библиотеку процедур, с помощью которых программы осуществляют доступ к услугам X на высоком уровне. Так для того, чтобы вывести на экран точку, достаточно вызвать процедуру `XDrawPoint()`, передав ей соответствующие параметры. Последняя выполняет всю черновую работу по подготовке и передачи пакетов данных серверу. Упомянутая библиотека называется *Xlib*. Она помещается в файле `lX11.a (libX11.so)`, который, как правило, находится в каталоге `/usr/X11R6/lib`. Прототипы функций библиотеки, используемые ею структуры данных, типы и прочее определяется в файлах-заголовках из директории `/usr/include/X11`.

На рис. 1.1 представлена схема общения клиентов и сервера.

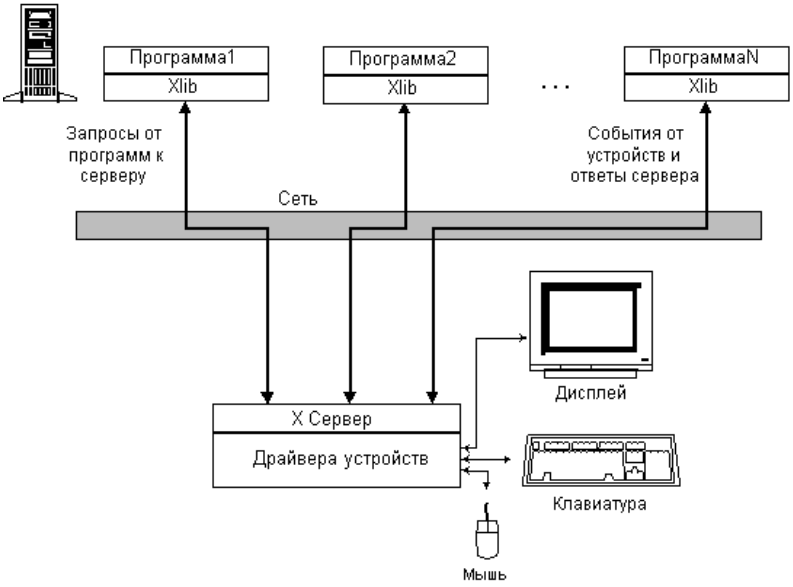


Рис. 1.1. Общая схема общения программ-клиентов и X-сервера

Посылка порций данных, особенно если она осуществляется через сеть, операция достаточно медленная. Чтобы повысить производительность системы, Xlib не отправляет пакеты сразу, а буферизует их в памяти машины, на которой выполняется программа-клиент. Собственно передача выполняется в тот момент, когда клиент вызывает процедуру, ожидающую получения событий от сервера, например `XNextEvent()`. Программа может явно инициировать отправку пакетов, обратившись к функциям `XFlush()` или `XSync()`.

1.1.2. X-окно

Как уже упоминалось ранее, окно – это базовое понятие в X. Оно представляет прямоугольную область на экране, предоставляемую системой программе-клиенту. Последняя использует окно для вывода графической информации. На рис. 1.2 показан общий вид окна в X Window.

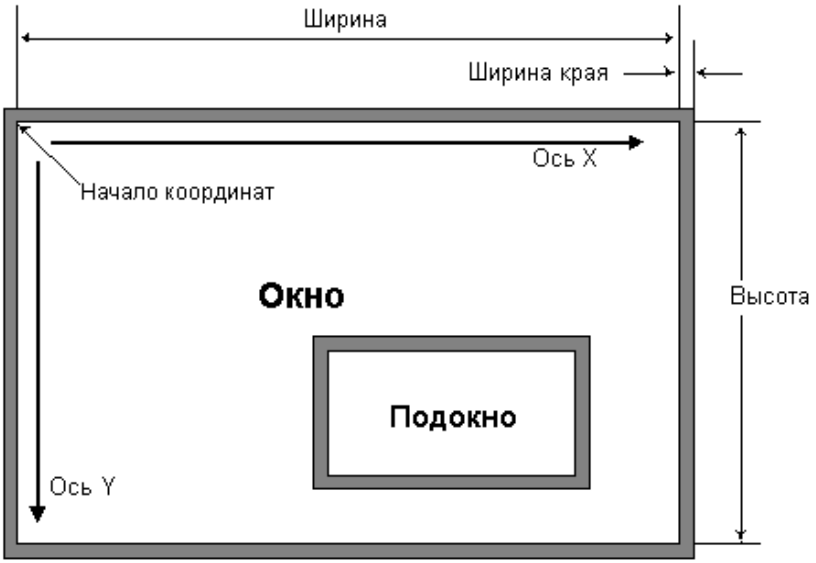


Рис. 1.2. Общий вид окна X Window

Из рисунка видно, что окно имеет внутренность и край. Основными атрибутами окна являются ширина и высота внутренности, а также ширина края. Далее мы будем говорить ширина и высота, а слово «внутренность» станем опускать. Упомянутые параметры окна называются его *геометрией*.

С каждым окном связывается система координат. Ее начало находится в левом верхнем углу окна. Ось *x* направлена вправо, а ось *y* – вниз. Единица измерения по обеим осям – пиксель.

Окна могут быть двух типов: `InputOutput` (для ввода-вывода) и `InputOnly` (только для ввода). Окно первого типа – это обычное окно. Окно второго типа не может использоваться для рисования. У данного окна нет края, оно «прозрачно». Заметим, что окна этого типа используются достаточно редко.

X Window позволяет программе создавать несколько окон одновременно. Они связаны в иерархию, в которой одни являются *родителями*, а другие *потомками*. Сам сервер на каждом экране создает одно основное окно, которое является самым верхним родителем всех остальных окон. Это окно мы будем называть главным или *корневым*.

Корневое окно всегда занимает весь экран. Это окно не может уничтожаться, меняться размеры или сворачиваться. Когда приложение создает окна, сначала должно быть создано по крайней мере одно окно верхнего уровня. Это окно становится прямым потомком корневого окна до тех пор, пока не отобразится. Прежде, чем это окно отобразится, оконный менеджер извещается об операции размещения окна. Оконный менеджер имеет привилегию «удочерить» новое окна верхнего уровня. Это используется для добавления окна, которое будет содержать новое окно и использоваться, чтобы нарисовать рамку, заголовок окна, системное меню и т.п.

Если такое окно верхнего уровня (который в действительности не окно верхнего уровня после того, как «удочерение» произошло) создано, приложение может создать в этом окне дочернее. Потомок может отображаться только в своем родительском окне – при перемещении его окно «обрезается» границей родительского. Любое окно может содержать более чем одно дочернее окно, и в этом случае эти окна упорядочиваются во внутренний стек. Когда окно верхнего уровня «поднимается», все его окна-потомки «поднимаются» вместе с ним, с сохранением их внутреннего упорядочения. Если окно потомка «поднято», оно поднимается только относительно своих собратьев.

1.1.3. Управление окнами

Окна могут располагаться на экране произвольным образом, перекрывая друг друга. X имеет набор средств, пользуясь которыми программа-клиент может изменять размеры окон и их положение на экране. Особенностью системы является то, что она не имеет встроенной возможности управлять окнами с помощью клавиатуры или мыши. Чтобы это можно было осуществить, нужен специальный клиент, который называется *менеджер окон* (Window manager). Стандартный дистрибутив X содержит такую программу – `twm`. Возможности этого менеджера ограничены, но, тем не менее, он позволяет осуществлять базовые действия: передвигать окна с помощью мыши, изменять их размер и т.д. Более развитым оконным менеджером является, по всей видимости, программа `mwm` (Motif Window Manager), которая поставляется в рамках системы OpenMotif.

Но менеджер не может корректно управлять окнами, ничего о них не зная. В одних случаях удобно иметь заголовки окон, в других случаях окно не может быть сделано меньше определенных размеров, а в некоторых

окно не может быть слишком увеличено. Окно может быть минимизировано (превращено в пиктограмму), в этом случае менеджер должен знать имя и вид пиктограммы. Для того, чтобы сообщить менеджеру свои пожелания относительно окон, клиенты могут использовать два способа. Во-первых, при создании окна X могут быть переданы рекомендации (hints) о начальном .положении окна, его ширине и высоте, минимальных и максимальных размерах и т.д. Во-вторых, можно использовать встроенный в X способ общения между программами – *механизм свойств*.

1.1.4. Графические возможности X Window

Система X Window предназначена для работы на растровых дисплеях. В подобного рода устройствах изображение представляется матрицей светящихся точек – пикселей. Каждый пиксель кодируется определенным числом бит (как правило 2, 4, 8, 16 или 24). Число бит-на-пиксель называют «толщиной» или *глубиной дисплея*. Биты с одинаковыми номерами во всех пикселях образуют как бы плоскость, параллельную экрану. Ее называют *цветовой плоскостью*. X позволяет рисовать в любой цветовой плоскости (или плоскостях), не затрагивая остальные.

Значение пикселя не задает непосредственно цвет точки на экране. Последний определяется с помощью специального массива данных, называемого *палитрой*. Цвет есть содержимое ячейки палитры, номер которой равен значению пикселя.

X имеет большой набор процедур, позволяющих рисовать графические примитивы – точки, линии, дуги, выводить текст и работать с областями произвольной формы.

1.1.5. Свойства и атомы

В X Window встроены средства для обеспечения

обмена информацией между программами-клиентами. Для этого используется механизм свойств (properties). *Свойство* – это порция данных, связанная с некоторым объектом (например, окном), и которая доступна всем клиентам X.

Каждое свойство имеет имя и уникальный идентификатор – *атом*. Обычно имена свойств записываются большими буквами, например: `MY_SPECIAL_PROPERTY`. Атомы используются для доступа к содержимому свойств с тем, чтобы уменьшить количество информации, пересылаемой по сети между клиентами и X сервером.

В X предусмотрен набор процедур, позволяющих перевести имя свойства в уникальный атом, и, наоборот, по атому получить необходимые данные.

Некоторые свойства и соответствующие им атомы являются предопределенными и создаются в момент инициализации сервера. Этим атомам соответствуют символические константы, определенные в файлах-заголовках библиотеки Xlib. Эти константы начинаются с префикса `XA_`.

1.1.6. Первый пример

Продолжая традиции многих изданий, посвященных программированию, начнем с программы, рисующей на экране строку «Hello, world!». В этом примере приведены основные шаги, необходимые для работы в X Window.

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

#define WND_X 0
#define WND_Y 0
#define WND_WDT 100
#define WND_HGH 100
```

```
#define WND_MIN_WDT 50
#define WND_MIN_HGH 50
#define WND_BORDER_WDT 5
#define WND_TITLE "Hello!"
#define WND_ICON_TITLE "Hello!"
#define PRG_CLASS "Hello!"

/*
* SetWindowManagerHints - процедура передает
информацию о
* свойствах программы менеджеру окон.
*/

static void SetWindowManagerHints (
 Display   *      prDisplay,   /*Указатель   на
структуру Display */
 char *   psPrgClass, /*Класс программы */
 char *   argv[],   /*Аргументы программы */
 int     argc,    /*Число аргументов */
 Window   nWnd,    /*Идентификатор окна */
 int      x,      /*Координаты левого верхнего
*/
 int     y,     /*угла окна */
 int     nWidth,
 int     nHeight,  /*Ширина и высота окна */
 int     nMinWidth,
 int       nMinHeight, /*Минимальные ширина и
высота окна */
 char *   psTitle,  /*Заголовок окна */
 char *    psIconTitle,/*Заголовок пиктограммы
окна */
 Pixmap   nIconPixmap /*Рисунок пиктограммы */
)
{
 XSizeHints   rSizeHints;   /*Рекомендации   о
размерах окна*/

 XWMHints rWMHints;
 XClassHint rClassHint;
```

```
 XTextProperty prWindowName, prIconName;

 if ( !XStringListToTextProperty (&psTitle, 1,
&prWindowName ) ||
    !XStringListToTextProperty   (&psIconTitle,
1, &prIconName ) ) {
  puts ( "No memory!\n");
  exit ( 1 );
}

rSizeHints.flags   =   PPosition   |   PSize   |
PMinSize;
rSizeHints.min_width = nMinWidth;
rSizeHints.min_height = nMinHeight;
rWMHints.flags = StateHint | IconPixmapHint |
         InputHint;
rWMHints.initial_state = NormalState;
rWMHints.input = True;
rWMHints.icon_pixmap= nIconPixmap;

rClassHint.res_name = argv[0];
rClassHint.res_class = psPrgClass;

XSetWMProperties     (     prDisplay,     nWnd,
&prWindowName,
  &prIconName,    argv,    argc,    &rSizeHints,
&rWMHints,
  &rClassHint );
}

/*
*main - основная процедура программы
*/

void main(int argc, char *argv[])
{
 Display   *prDisplay;      /*   Указатель   на
структуру Display */
 int nScreenNum;    /* Номер экрана */
```

```
 GC prGC;
 XEvent rEvent;
 Window nWnd;

 /* Устанавливаем связь с сервером */
 if ( ( prDisplay = XOpenDisplay ( NULL ) ) ==
NULL ) {
  puts ("Can not connect to the X server!\n");
  exit ( 1 );
 }

 /* Получаем номер основного экрана */
 nScreenNum = DefaultScreen ( prDisplay );

 /* Создаем окно */
 nWnd = XCreateSimpleWindow ( prDisplay,
     RootWindow ( prDisplay, nScreenNum ),
     WND_X,     WND_Y,     WND_WDT,     WND_HGH,
WND_BORDER_WDT,
     BlackPixel ( prDisplay, nScreenNum ),
     WhitePixel ( prDisplay, nScreenNum ) );

 /* Задаем рекомендации для менеджера окон */
 SetWindowManagerHints ( prDisplay, PRG_CLASS,
argv, argc,
   nWnd,   WND_X,   WND_Y,   WND_WDT,   WND_HGH,
WND_MIN_WDT,
   WND_MIN_HGH,  WND_TITLE,  WND_ICON_TITLE,  0
);

 /*     Выбираем     события,     обрабатываемые
программой */
 XSelectInput ( prDisplay, nWnd, ExposureMask
| KeyPressMask );

 /* Показываем окно */
 XMapWindow ( prDisplay, nWnd );

 /* Цикл получения и обработки событий */
```

```
 while ( 1 ) {
  XNextEvent ( prDisplay, &rEvent );

  switch ( rEvent.type ) {
    case Expose :
     /* Запрос на перерисовку */
     if ( rEvent.xexpose.count != 0 )
      break;

     prGC = XCreateGC ( prDisplay, nWnd, 0 ,
NULL );

     XSetForeground ( prDisplay, prGC,
       BlackPixel ( prDisplay, 0) );
     XDrawString ( prDisplay, nWnd, prGC, 10,
50,
       "Hello,  world!",  strlen  (  "Hello,
world!" ) );
     XFreeGC ( prDisplay, prGC );
     break;

    case KeyPress :
     /* Нажатие клавиши клавиатуры */
     XCloseDisplay ( prDisplay );
     exit ( 0 );
  }
 }
}
```

Для сборки программы используется команда:

```
cc -o hello hello.c -lX11 -L/usr/X11R6/lib
```

Здесь `cc` – имя исполняемого файла компилятора. Как правило, это символическая ссылка на реальное имя компилятора (например, `gcc`). Параметр `-o` задает имя исполняемого файла; в нашем случае это `hello`. `-lX11` указывает на необходимость подключения библиотеки Xlib, а `-L/usr/X11R6/lib` определяет путь к ней.

В современных версиях UNIX для создания программных продуктов используются не только

компиляторы командной строки, но и самые разнообразные интегрированные среды. Одной из наиболее удобных, по нашему мнению, является интегрированная среда разработки *Анюта* (Anjuta). Ее создатель – индийский программист Наба Кумар – позаботился о том, чтобы мы чувствовали себя в ней комфортно.

Для того, чтобы разрешить в Анюте поддержку русского языка, необходимо добавить в файл свойств этой программы (`~/.anjuta/session.properties`) строку

```
character.set=204
```

Для подключения библиотеки Xlib необходимо добавить ее имя (`X11`) в закладке «Библиотеки» установок компилятора и компоновщика, а путь к ней (`/usr/X11R6/lib`) – в закладке «Путь к библиотекам».

На рис. 1.3 показан внешний вид приложения после его запуска.

Рис. 1.3. Окно приложения xhello в среде KDE

Программа использует ряд функций, предоставляемых библиотекой Xlib: `XOpenDisplay()`, `XCreateSimpleWindow()` и др. Их прототипы, стандартные структуры данных, макросы и константы описаны в следующих основных файлах-заголовках: `Xlib.h`, `Xutil.h`, `Xos.h`. Эти и другие файлы поставляются вместе с X Window.

Перейдем к рассмотрению самой программы. Она начинается установлением связи с X-сервером. Делает это функция `XOpenDisplay()`. Ее аргумент определяет сервер,

с которым надо связаться. Если в качестве параметра `XOpenDisplay()` получает `NULL`, то она открывает доступ к серверу, который задается переменной среды (environment) `DISPLAY`. И значение этой переменной, и значение параметра функции имеют следующий формат: `host:server.screen`, где `host` – имя компьютера, на котором выполняется сервер, `server` – номер сервера (обычно это 0), а `screen` – это номер экрана. Например, запись kiev:0.0 задает компьютер kiev, а в качестве номера сервера и экрана используется 0. Заметим, что номер экрана указывать не обязательно.

Процедура `XOpenDisplay()` возвращает указатель на структуру типа `DISPLAY`. Это большой набор данных, содержащий информацию о сервере и экранах. Указатель следует запомнить, т.к. он используется в качестве параметра во многих процедурах Xlib.

`XOpenDisplay()` соединяет программу с X сервером, используя протоколы TCP или DECnet, или же с использованием некоторого локального протокола межпроцессного взаимодействия. Если имя машины и номер дисплея разделяются одним знаком двоеточия (:), то `XOpenDisplay()` производит соединение с использованием протокола TCP. Если же имя машины отделено от номера дисплея двойным двоеточием (::), то для соединения используется протокол DECnet. При отсутствии поля имени машины в имени дисплея, то для соединения используется наиболее быстрые из доступных протоколов. Конкретный X сервер может поддерживать как все, так и некоторые из этих протоколов связи. Конкретные реализации Xlib могут дополнительно поддерживать другие протоколы.

Если соединение проведено удачно, `XOpenDisplay()` возвращает указатель на структуру `Display`, которая определяется в `<X11/Xlib.h>`. Если же установить соединение не удалось, то `XOpenDisplay()` возвращает

NULL. После успешного вызова XOpenDisplay() клиентской программой могут использоваться все экраны дисплея. Номер экрана возвращается макроопределением DefaultScreen() и функцией XDefaultScreen(). Доступ к полям структур Display и Screen возможен только посредством использования макроопределений и функций.

После того, как связь с сервером установлена, программа «Hello» определяет номер экрана. Для этого используется макрос DefaultScreen(), возвращающий номер основного экрана. Переменная nScreenNum может иметь значение от 0 до величины (ScreenCount(prDisplay)-1). Макрос ScreenCount() позволяет получить число экранов, обслуживаемых сервером.

Следующий шаг – создание окна и показ его на дисплее. Для этого программа обращается к процедуре XCreateWindow() или XCreateSimpleWindow(). Для простоты мы используем вторую процедуру, параметры которой задают характеристики окна.

```
  PrWind = XCreateSimpleWindow (
      prDisplay, /* указатель на структуру
Display,
                описывающую сервер */
      RootWindow (prDisplay, nScreenNum),
            /* родительское окно, в данном
случае,
                это основное окно программы */
      WND_X, WND_Y,
            /* начальные x и y координаты
верхнего
                левого угла окна программы */
      WND_WIDTH, WND_HEIGHT,
            /* ширина окна и высота окна */
      WND_BORDER_WIDTH, /* ширина края окна */
      BlackPixel ( prDisplay, nScreenNum ),
            /* цвет переднего плана окна */
```

```
        WhitePixel ( prDisplay, nScreenNum )
              /* цвет фона окна */
    );
```

Для задания цветов окна используются макросы `BlackPixel()` и `WhitePixel()`. Они возвращают значения пикселей, которые считаются на данном дисплее и экране соответствующими «черному» и «белому» цветам. Функция `XCreateSimpleWindow()` (`XCreateWindow()`) возвращает значение типа `Window`. Это целое число, идентифицирующее созданное окно.

Среди параметров функций, создающих окна, есть те, которые определяют положение окна и его размеры. Эти аргументы принимаются во внимание системой X Window. Исключение составляет случай, когда родительским для создаваемого окна является «корневое» окно экрана. В этом случае решение о положение окна и его размерах принимает менеджер окон. Программа может пытаться повлиять на решение менеджера окон, сообщив ему свои «пожелания» с помощью функции `XSetWMProperties()`.

Из листинга видно, что программа может сообщить менеджеру следующие параметры:

- имя (заголовок) окна;
- имя пиктограммы окна;
- саму пиктограмму;
- параметры `argc` и `argv`, передаваемые от UNIX функции main();
- желаемое положение окна, его размеры, другие рекомендации о его геометрии.

Имя окна и имя пиктограммы должны быть в начале преобразованы в «текстовые свойства», описываемые структурами типа `XTextProperty`. Это выполняется процедурой `XStringListToTextProperty()`.

Для передачи информации о желаемой геометрии окна используется структура `XSizeHints()`.

X Window позволяет сообщить менеджеру также

следующее:

- начальное состояние окна; нормальное или минимизированное;
- воспринимает ли окно ввод с клавиатуры;
- класс программы и ее имя для чтения ресурсов из базы данных ресурсов.

После того, как «рекомендации» менеджеру окон переданы, программа выбирает события, на которые она будет реагировать. Для этого вызывается функция `XSelectInput()`. Ее последний аргумент есть комбинация битовых масок (флагов). В нашем случае это `ExposureMask | KeyPressMask`. `ExposureMask` сообщает X Window, что программа обрабатывает событие `Expose`. Оно посылается сервером каждый раз, когда окно должно быть перерисовано. `KeyPressMask` выбирает событие `KeyPress` – нажатие клавиши клавиатуры.

Теперь окно программы создано, но не показано на экране. Чтобы это произошло, надо вызвать процедуру `XMapWindow()`. Заметим, что из-за буферизации событий библиотекой Xlib, окно не будет реально нарисовано, пока программа не обратится к процедуре получения сообщений от сервера `XNextEvent()`.

Программы для X построены по принципу управляемости событиями. Поэтому, после того, как окно создано, заданы необходимые параметры для менеджера окон, основная ее работа – это получать сообщения от сервера и откликаться на них. Выполняется это в бесконечном цикле. Очередное событие «вынимается» процедурой `XNextEvent()`. Само оно есть переменная типа `XEvent`, который представляет собой объединение структур. Каждое событие (`Expose`, `KeyPress` и т.д.) имеет свои данные (и, следовательно, свое поле в объединении `XEvent`).

При получении сообщения `Expose` программа

перерисовывает окно. Это событие является одним из наиболее важных событий, которые приложение может получить. Оно будет послано нашему окну в одном из различных случаев:

- другое окно перекрыло часть нашего;
- наше окно было выведено поверх всех других окон;
- наше окно впервые прорисовывается на экране;
- наше окно было восстановлено после сворачивания.

Когда мы получаем событие `Expose`, мы должны взять данные события из члена `xexpose` объединения `XEvent`. Он содержит различные интересные поля:

`count` – количество других событий `Expose`, ожидающие в очереди событий сервера. Это может быть полезным, если мы получаем несколько таких сообщений подряд – рекомендуется избегать перерисовывать окно, пока мы не получим последнее из их (то есть пока `count` не равно 0).

`window` – идентификатор окна, которому было послано сообщение `Expose` (в случае, если приложение зарегистрировало это событие в различных окнах).

`x`, `y` – координаты верхнего левого угла области окна, которая должна быть перерисована.

`width`, `height` – ширина и высота области окна, которая должна быть перерисована.

Действия по обработке `Expose` начинаются с создания графического контекста – структуры, которая содержит данные, необходимые для вывода информации, в нашем случае – текста:

```
prGC = XCreateGC (prDisplay, prWnd, 0, NULL);
```

После этого рисуется строка «Hello, world!». Более графический контекст не нужен – он уничтожается:

```
XFreeGC (prDisplay, prGC);
```

Окно может получить несколько событий `Expose`

одновременно. Чтобы не перерисовывать себя многократно, программа дожидается прихода последнего из них и только потом осуществляет вывод.

Приход события `KeyPress` означает, что программу надо завершить: прекратить связь с сервером

```
XCloseDisplay (prDisplay);
```

и вызвать функцию `exit()`.

`XCloseDisplay()` закрывает соединение с X сервером, закрывает все окна и удаляет идентификаторы ресурсов, созданных клиентом на дисплее. Для удаления только окна без разрыва связи с X сервером необходимо использовать функции `XDestroyWindow()` и `XDestroySubWindows()`.

1.1.7. События

Когда пользователь нажимает на кнопку мыши или клавишу клавиатуры, или когда окно программы нуждается в перерисовке, или когда происходят другие изменения в системе, сервер подготавливает соответствующий пакет данных и отправляет его той или иной программе (или программам). Этот пакет данных называется *событием*.

Типичная GUI программа имеет следующую структуру:

1. Выполняются инициализационные процедуры.
2. Устанавливается соединение с X сервером.
3. Выполняются инициализационные процедуры, связанные с X.
4. Пока не завершились:
    1. Получаем следующее событие от X сервера.
    2. Обрабатываем событие, возможно посылая различные запросы на рисование к X серверу.
    3. Если событие было завершающим, заканчиваем цикл.
5. Закрываем соединение с X сервером.
6. Выполняем завершающие действия.

Возможных событий достаточно много; их список можно найти в файле `X.h`. Каждое из них имеет свой тип и

соответствующую структуру данных. Все они вместе, как было сказано выше, описываются объединением `XEvent`.

Как мы видели из примера в предыдущем пункте, программа для каждого из своих окон может выбрать события, которые будут ему передаваться. Делается это с помощью функции `XSelectInput()`. При вызове этой процедуры требуемые события идентифицируются соответствующими флагами. Так событию, `ButtonPress` (нажатие кнопки мыши) соответствует флаг `ButtonPressMask`. Когда кнопка отпускается, сервер порождает событие `ButtonRelease`, которому соответствует флаг – `ButtonReleaseMask`.

Маска выбираемых событий может составляться с помощью побитового “ИЛИ” из таких значений:

`0` – не ожидать никаких событий

`KeyPressMask` – ожидать событие нажатия клавиши

`KeyReleaseMask` – ожидать событие отпускания клавиши

`ButtonPressMask` – ожидать событие нажатия кнопки мыши

`ButtonReleaseMask` – ожидать событие отпускания кнопки мыши

`EnterWindowMask` – ожидать событие входа в окно

`LeaveWindowMask` – ожидать событие выхода из окна

`PointerMotionMask` – ожидать событие движения мышиного курсора

`PointerMotionHintMask` – ожидать событие движения мышиного курсора с дополнительными указаниями

`Button1MotionMask` – ожидать событие движения мышиного курсора при нажатой первой кнопке

`Button2MotionMask` – ожидать событие движения мышиного курсора при нажатой второй кнопке

`Button3MotionMask` – ожидать событие движения

мышиного курсора при нажатой третьей кнопке

`ButtonMotionMask` – ожидать событие движения мышиного курсора при любой нажатой кнопке

`ExposureMask` – ожидать событие необходимости перерисовки окна

`VisibilityChangeMask` – ожидать событие изменения видимости

`ResizeRedirectMask` – ожидать событие изменения размеров окна

`FocusChangeMask` – ожидать событие изменения фокуса ввода

Некоторые события посылаются окну независимо от того, выбраны они или нет. Это:

`MappingNotify` – посылается, когда изменяется состояние клавиатуры (соответствие физических и логических кодов;

`ClientMessage` – так идентифицируются события, посылаемые от клиента к клиенту с помощью процедуры `XSendEvent()`;

`SelectionClear`, `SelectionNotify`, `SelectionRequest` – эти события используются в стандартном механизме общения между программами, работающими в X;

`Expose`, `NoExpose` – эти события могут посылаться, когда клиент пытается копировать содержимое одного окна в другое.

Частой ошибкой начинающих программистов является добавление кода для обработки нового события без добавления маски для этого события в `XSelectInput()`. Можно часам сидеть и отлаживать программу в недоумении, почему она не реагирует на отпускание кнопки, только из-за того, что для кнопок событие нажатия зарегистрировано, а событие отпускания – нет.

Программа получает события в своем основном цикле. Для этого можно использовать ряд процедур. Наиболее

простая из них `XNextEvent(Display *prDisplay, XEvent *prEvent)`. Она «вынимает» из очереди событие, находящееся в ее «голове», сохраняет информацию о нем в переменной, на которую указывает параметр `prEvent`, и возвращается. При этом само событие удаляется из очереди. Функция `XPeekEvent()` также возвращает переданное сервером событие, но не удаляет его из очереди.

Процедура `XPending()` возвращает общее число событий в очереди программы.

Итак, если событие выбрано для окна, то оно будет передано ему на обработку. А если нет? В этом случае событие передается родителю окна. Если и тот не желает обращать внимание на данное событие, то оно отправляется дальше, вверх по иерархии окон, и так до тех пор, пока либо не будет найдено окно, выбравшее это событие, либо событие не потеряется.

Задача может влиять на этот процесс продвижения события по иерархии окон. Если программа включает флаг, соответствующий событию, в специальный атрибут окна, то оно, достигнув это окно, не будет передано родителю, а будет тут же «снято с повестки дня». Этот атрибут – `do_not_propagate`.

1.1.8. Атрибуты окна

Многие атрибуты окна задаются при его создании с помощью процедуры `XCreateWindow()` или `XCreateSimpleWindow()`. Впоследствии параметры можно изменить, обратившись к процедуре `XChangeWindowAttributes()`.

Характеристики окна описываются структурами типа `XSetWindowAttributes` и `XWindowAttributes`. Получить их можно с помощью процедуры `XGetWindowAttributes()`.

Все они делятся на две группы. В первую входят

параметры, доступные «на чтение» и «на запись». Вторая группа представляет собой внутренние данные. Программа может прочитать их, но не может менять.

Сначала перечислим поля этих структур, которые относятся к «изменяемым» параметрам.

Фон окна определяется атрибутами `background_pixmap` и `background_pixel`. Первый из них задает картинку (карту пикселей), которая используется для заливки фона окна. При необходимости картина повторяется слева направо и сверху вниз. Если параметр `background_pixmap` равен `None` (задается по умолчанию), то он игнорируется. Если же при этом поле `background_pixel` не задано (установлено по умолчанию), то окно считается «прозрачным», в противном случае его фон заливается цветом `background_pixel`. Атрибуты `background_pixmap` и `background_pixel` могут также принимать значение `ParentRelative`. В этом случае характеристики фона заимствуются у родительского окна.

Вид края окна определяется полями `border_pixmap` и `border_pixel`. Первый атрибут определяет карту пикселей, используемую для заполнения края. Если он равен `None`, то край заполняется цветом `border_pixel`. Если же и поле `border_pixel` не задано, то для изображения края используются соответствующие характеристики родителя. То же самое происходит, если параметр `border_pixmap` равен `CopyFromParent` (взять у родителя). Последнее значение есть значение по умолчанию.

На *перерисовку* окна после изменения его размеров влияют атрибуты `bit_gravity` и `win_gravity`. Когда окно меняет размер, например, увеличивается или уменьшается, то, в принципе, нет необходимости перерисовывать все его содержимое. Часть окна остается неизменной. Правда, эта часть может поменять свое

положение: переместиться вправо, влево, вверх или вниз. Поле `bit_gravity` говорит серверу, что делать с оставшейся частью изображения. Возможные значения параметра следующие:

`ForgetGravity` – содержимое окна перерисовывается (считается значением по умолчанию);

`StaticGravity` – остающаяся часть не должна менять положение по отношению к главному (корневому (root)) окну сервера;

`NorthWestGravity` – остающаяся часть смещается к левому верхнему углу;

`NorthGravity` – остающаяся часть смещается к верху окна;

`NorthEastGravity` – остающаяся часть смещается к правому верхнему углу;

`WestGravity` – остающаяся часть смещается к левому краю окна;

`CenterGravity` – остающаяся часть смещается к центру окна;

`EastGravity` – остающаяся часть смещается к правому краю окна;

`SouthWestGravity` – остающаяся часть смещается к левому нижнему углу;

`SouthGravity` – остающаяся часть смещается к нижнему краю окна;

`SouthEastGravity` – остающаяся часть смещается к правому нижнему углу.

Параметр `win_gravity` говорит о том, что делать с *подокнами* окна после изменения размеров последнего. Возможные значения параметра следующие (при перечислении используются следующие обозначения: `H` – изменение размеров окна по горизонтали, `V` – изменение размеров по вертикали, `(H, V)` – смещение подокна на `H` пикселей по горизонтали и на `V` пикселей по вертикали):

UnmapGravity – подокна удаляются с экрана; окну посылается событие UnmapNotify, в ответ на которое оно может переместить свои подокна и показать их с помощью процедуры XMapSubWindow();

StaticGravity – подокна остаются на месте по отношению к главному (корневому) окну сервера;

NorthWestGravity – устанавливается по умолчанию; соответствует смещению (0, 0);

NorthGravity – смещение (H/2, 0);

NorthEastGravity – смещение (H, 0);

WestGravity – смещение (0, V/2);

CenterGravity – смещение (H/2, V/2);

EastGravity – смещение (H, V/2);

SouthWestGravity – смещение (0, V);

SouthGravity – смещение (H/2, V);

SouthEastGravity – смещение (H, V);

Автоматическое сохранение содержимого окна, когда его часть перекрывается другими окнами, или, когда окно удаляется с экрана, определяется параметрами backing_store, backing_planes и backing_pixel. Сохраненные данные могут использоваться для восстановления окна, что значительно быстрее, чем его перерисовка программой в ответ на событие Expose. Параметр backing_store имеет следующие возможные значения:

NotUseful (устанавливается по умолчанию) – серверу не рекомендуется сохранять содержимое окна;

WhenMapped – серверу рекомендуется спасти содержимое невидимых частей окна, когда окно показывается на экране;

Always – серверу рекомендуется сохранить содержимое окна даже, если оно не показано на экране.

Сохранение изображений требует, как правило, довольно большого расхода памяти. Атрибуты

backing_planes и backing_pixel призваны *уменьшить этот расход*. Первый из указанных параметров говорит серверу, какие плоскости изображения надо сохранять; backing_pixel означает, какой цвет использовать при восстановлении изображения в тех плоскостях, которые не сохранялись. По умолчанию backing_planes – маска, состоящая из единиц, а backing_pixel равно 0.

Иногда при показе окна полезно *сохранить содержимое экрана под окном*. Если окно невелико, и показывается не на долго, то это позволяет экономить время, которое надо будет затратить на перерисовку экрана после того, как окно будет закрыто. Если атрибут save_under равен True, то сервер будет пытаться сохранить изображение под окном. Если же он равен False (по умолчанию), то сервер ничего не предпринимает.

Когда обрабатывает (или не обрабатывает) событие, последнее может быть *передано его родительскому окну*. Атрибут do_not_propagate_mask (по умолчанию 0) говорит и о том, какие события не должны доходить до родителей.

Изменение размеров окна и его положения на экране контролируется атрибутом override_redirect. Если он равен False, то размер окна и его положение меняются с помощью менеджера окон. Если же он равен True, то окно само решает, где ему быть, и какую ширину и высоту иметь.

Цветовую гамму окна задает параметр colormap. Значение по умолчанию – CopyFromPatent, которое говорит, что окно использует палитру своего непосредственного родителя.

Теперь рассмотрим «неизменяемые» параметры окна. Строго говоря, атрибуты, о которых пойдет речь, нельзя назвать неизменяемыми. Некоторые из них могут меняться сервером или менеджером окон. Но для обычных программ-клиентов они действительно являются таковыми.

Положение окна и его размеры сообщают поля `x`, `y`, `width` и `height`. Они дают координаты левого верхнего угла, ширину и высоту окна соответственно. Координаты измеряются в пикселях по отношению к родительскому окну.

Ширина края окна определяется параметром `border_width`.

Маска, говорящая о том, *какие события выбраны для передачи окну* породившим его клиентом, содержится в поле флагов `your_event_mask`. Значение параметра образуется комбинацией флагов, идентифицирующих события.

Информация о дисплее, на котором показано окно, содержится в структуре `Visual`, на которую показывает поле `visual`. Эти данные, как правило, не обрабатываются обычными программами-клиентами (заметим, что для получения информации о дисплее, в системе предусмотрена процедура `XGetVisualInfo()`).

Класс окна сообщает поле `class`. Возможные значения: `InputOutput` и `InputOnly`.

Число цветовых плоскостей дисплея (число бит-на-пиксел) помещается в поле `depth`.

На *информацию об экране*, на котором помещается окно, указывает поле `screen`. Она, как правило, не используется обычными программами.

Идентификатор главного (корневого) окна экрана, на котором помещается окно, находится в поле `root`.

Если окно имеет палитру, и она в настоящее время активна, то поле `map_installed` равно `True`, в противном случае – `False`.

Видно в настоящее время окно на экране или нет, сообщает атрибут `map_state`.

Маска всех событий, выбранных всеми программами для данного окна, содержится в атрибуте `all_event_mask`.

Дело в том, что окно обрабатывается не только порождающим его клиентом, но, возможно, и другими приложениями, например, менеджером окон.

Мы рассказали о том, как получить атрибуты окна, и что они означают. Теперь рассмотрим, как их изменить. Для этого можно использовать несколько процедур X Window, основной из которых является `XChangeWindowAttributes()`, имеющая следующий прототип:

```
int XChangeWindowAttributes (Display
*prDisplay,
     Window nWnd, unsigned long nValueMask,
     XSetWindowAttributes *prWinAttr);
```

Требуемые установки атрибутов передаются через аргумент `prWinAttr`. Он указывает на переменную типа `XSetWindowAttributes`. Ее поля те же, что и соответствующие поля `XWindowAttributes`. Разница заключается лишь в разных именах некоторых из них. Так, поле `your_event_mask` в `XWindowAttributes` соответствует полю event_mask в `XSetWindowAttributes`.

Структура `XSetWindowAttributes` содержит дополнительное поле `cursor`. Оно определяет *вид курсора мыши*, когда последний находится в окне. Если поле равно `None` (значение по умолчанию), то используется курсор родительского окна, в противном случае значением параметра должен быть идентификатор того или иного курсора.

Параметр `nValueMask` при вызове указанной процедуры представляет комбинацию флагов, говорящих о том, какие из полей переменной `prWinAttr` принимать во внимание.

В следующем примере приведен фрагмент кода, в котором изменяются параметры `border_pixmap` и `win_gravity` некоторого окна:

```
.......
```

```
Display *prDisplay;
Window prWnd;
XSetWindowAttributes rWndAttr;
unsigned long nValMask;
Pixmap nPixmap=0;
.......
nValMask = CWBorderPixmap | CWWinGravity;
rWndAttr.border_pixmap = nPixmap;
rWndAttr.win_gravity = StaticGravity;
.......
XChangeWindowAttributes (prDisplay, prWnd,
nValMask, &rWndAttr);
.......
```

Отдельные атрибуты окна можно изменить более просто с помощью специальных процедур. Так функция `XSetWindowBackground()` меняет фон окна, `XSetWindowBorder()` – его край.

1.1.9. Операции над окнами

Манипулировать окнами можно не только с помощью атрибутов: Xlib предоставляет набор функций для изменения их размеров, перемещения на экране и в стеке окон, сворачивания и т.п.

Первая пара операций, которые можно применить к окну – *отображение или скрытие*. Отображение окна заставляют окно появиться на экране, скрытие приводит к удалению с экрана (хотя логическое окно в памяти все еще существует). Например, если в вашей программе есть диалоговое окно, вместо создания его каждый раз по запросу пользователя, мы можем создать окно один раз в скрытом режиме и, когда пользователь запросит открыть диалог, просто отобразить окно на экране. Когда пользователь нажимает «`OK`» или «`Cancel`», окно скрывается. Это значительно быстрее создания и уничтожения окна, однако стоит ресурсов, как на стороне клиента, так и на стороне X сервера.

Отображение окна может быть выполнено с помощью `XMapWindow()`, скрытие – с помощью `XUnmapWindow()`. Функция отображения заставит событие `Expose` послаться программе, если только окно полностью не закрыто другими окнами.

Другое действие, которое можно выполнить над окнами – *переместить* их в другую позиции. Это может быть выполнено функцией `XMoveWindow()`, которая принимает новые координаты окна. Имейте в виду, что после перемещения окно может быть частично скрытым другими окнами (или наоборот, открыто ими), и таким образом, может быть сгенерировано сообщение `Expose`.

Изменить размер окна можно с помощью функции `XResizeWindow()`. Мы можем также объединить перемещение и изменение размеров, используя одну функцию `XMoveResizeWindow()`.

Все приведенные выше функции изменяли свойства одного окна. Существует ряд свойств, связанных с данным окном и другими окнами. Одно из них – *порядок засылки в стек*: порядок, в котором окна располагаются друг над другом. Говорят, что окно переднего плана находится на верхе стека, а окно заднего плана – на дне стека. Перемещение окна на вершину стека осуществляет функция `XRaiseWindow()`, перемещение окна на дно стека – функция `XLowerWindow()`.

С помощью функции `XIconifyWindow()` окно может быть *свернуто*, а с помощью `XMapWindow()` – *восстановлено*. Для того, чтобы понять, почему для `XIconifyWindow()` нет обратной функции, необходимо заметить, что, когда окно сворачивается, на самом деле оно скрывается, а вместо него отображается окно иконки. Таким образом, чтобы восстановить исходное окно, нужно просто отобразить его снова. Иконка является на самом деле другим окном, которое просто тесно связано сильно с нашим нормальным окном – это не другое состояние

нашего окна.

Следующий пример демонстрирует использование операций над окнами:

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

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/*
 create_simple_window – создает окно с белым фоном заданного размера.
 Принимает в качестве параметров дисплей, размер окна (в пикселях)
 и положение окна (также в пикселях). Возвращает дескриптор окна.
 Окно создается с черной рамкой шириной в 2 пикселя и автоматичсеки
 отображается после создания.
*/
Window create_simple_window(Display* display,
              int width, int height, int x, int y)
{
 int screen_num = DefaultScreen(display);
 int win_border_width = 2;
 Window win;

/*
 создаем простое окно как прямой потомок корневого окна экрана,
 используя черный и белый цвета в качестве основного и фонового, и
 размещая новое окно в верхнем левом углу по заданным координатам
*/
 win       =       XCreateSimpleWindow(display, RootWindow(display, screen_num),
```

```
                x,      y,      width,      height,
win_border_width,
                BlackPixel(display, screen_num),
                WhitePixel(display,
screen_num));

 /* Отображаем окно на экране. */
 XMapWindow(display, win);

 /*  Заставляем  выполниться  все  запросы  к  X
серверу. */
 XFlush(display);

 return win;
}


void main(int argc, char* argv[])
{
 Display*  display;  /*  указатель  на  структуру
дисплея X */
 int  screen_num;  /*  количество  экранов  для
размещения окон */
 Window  win;  /*  дескриптор  создаваемого  окна
*/
 unsigned int display_width,
        display_height;  /*  высота  и  ширина  X
дисплея */
 unsigned int win_width,
        win_height;  /*  высота  и  ширина  нового
окна */
 char  *display_name  =  getenv("DISPLAY");  /*
имя X дисплея */

 /* устанавливаем соединение с X сервером */
 display = XOpenDisplay(display_name);
 if (display == NULL) {
  fprintf(stderr, "%s: не могу соединиться с X
сервером '%s'\n",
```

```
      argv[0], display_name);
  exit(1);
 }

 /* получаем геометрию экрана по умолчанию для
нашего дисплея */
 screen_num = DefaultScreen(display);
 display_width      =      DisplayWidth(display,
screen_num);
 display_height     =     DisplayHeight(display,
screen_num);

 /* создаем новое окно в 1/9 площади экрана */
 win_width = (display_width / 3);
 win_height = (display_height / 3);
 /* отладочная печать в стандартный вывод */
 printf("ширина окна - '%d'; высота - '%d'\n",
win_width, win_height);

 /* создаем простое окно как прямой потомок
корневого окна экрана,  */
 /* используя черный и белый цвета в качестве
основного и фонового, и*/
 /* размещая новое окно в верхнем левом углу
по заданным координатам */
 win       =       create_simple_window(display,
win_width, win_height, 0, 0);

 XFlush(display);

 /* отдохнем после трудов праведных */
 sleep(3);

 /* пример изменения размеров окна */
 {
  int i;

  /* в цикле уменьшаем окно */
  for (i=0; i<40; i++) {
```

```
   win_width -= 3;
   win_height -= 3;
   XResizeWindow(display,    win,    win_width,
win_height);
   XFlush(display);
   usleep(20000);
  }

  /* в цикле увеличиваем окно */
  for (i=0; i<40; i++) {
   win_width += 3;
   win_height += 3;
   XResizeWindow(display,    win,    win_width,
win_height);
   XFlush(display);
   usleep(20000);
  }
 }

 sleep(1);

 /* пример перемещения окна */
 {
  int i;
  XWindowAttributes win_attr;
  int x, y;
  int scr_x, scr_y;
  Window child_win;
  /*   переменная   для   хранения   дескриптора
родительского окна */
  Window parent_win;

  /* вначале получаем текущие атрибуты окна */
  XGetWindowAttributes(display,            win,
&win_attr);

  x = win_attr.x;
  y = win_attr.y;
```

```
  /* затем находим окно родителя */
  {
   /* эта переменная будет хранить дескриптор
корневого окна */
   /* экрана, на котором отображено наше окно
*/
   Window root_win;
   /* эта переменная будет хранить массив
дескрипторов */
   /* дочерних окон нашего окна, */
   Window* child_windows;
   /* а эта – их количество */
   int num_child_windows;

   /* выполним запрос необходимых значений */
   XQueryTree(display, win,
         &root_win,
         &parent_win,
         &child_windows, &num_child_windows);

   /* мы должны освободить список дочерних
дескрипторов, */
   /* так как он был динамически выделен
XQueryTree()  */
   XFree(child_windows);
  }

  /* Транслируем локальные координаты в
экранные, используя  */
  /* корневое окно как окно, относительно
которого выполняется */
  /* трансляция. Это работает потому, что
корневое окно всегда */
  /*занимает весь экран, и его левый верхний
угол совпадает  */
  /*    с    левым    верхним    углом    экрана
*/
  XTranslateCoordinates(display,
             parent_win, win_attr.root,
```

```
                x, y,
                &scr_x, &scr_y,
                &child_win);

  /* перемещаем окно влево */
  for (i=0; i<40; i++) {
   scr_x -= 3;
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   usleep(20000);
  }

  /* перемещаем окно вниз */
  for (i=0; i<40; i++) {
   scr_y += 3;
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   usleep(20000);
  }

  /* перемещаем окно вправо */
  for (i=0; i<40; i++) {
   scr_x += 3;
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   usleep(20000);
  }

  /* перемещаем окно вверх */
  for (i=0; i<40; i++) {
   scr_y -= 3;
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   usleep(20000);
  }
 }

 sleep(1);
```

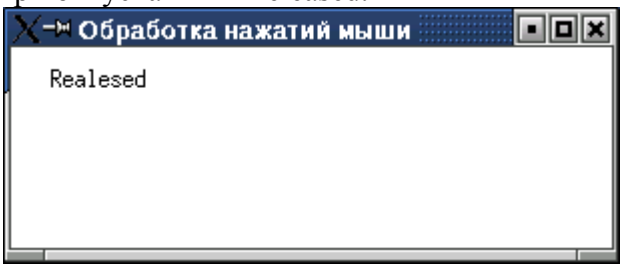
```
 /* пример сворачивания и восстановления окна
*/
 {
  /* сворачиваем окно */
  XIconifyWindow(display,                    win,
DefaultScreen(display));
  XFlush(display);
  sleep(2);
  /* восстанавливаем окно */
  XMapWindow(display, win);
  XFlush(display);
  sleep(2);
 }

 XFlush(display);

 /* короткая передышка */
 sleep(2);

 /* закрываем соединение с X сервером */
 XCloseDisplay(display);
}
```

1.1.10. Лабораторная работа №1 «Основные понятия Xlib»

1. Используя компилятор командной строки, выполните компиляцию примера из п. 1.1 и выполните полученную программу.

2.Настройте интегрированную среду Анюта для работы с кириллицей и выполните компиляцию в ней предыдущей программы.

3.На основе примера напишите программу, которая при нажатии клавиши мыши пишет сообщение Pressed, а при отпускании – Released.



4.Используя функции XFlush()/XSync() и sleep(), напишите программу без цикла обработки сообщений, отображающую черное окно размером 100x100 пикселей в течение 5 секунд.

1.2. Текст и графика

В данном разделе описываются возможности, которые имеет программист для вывода текста и произвольных графических изображений. Особенностью X является то, что рисовать можно не только в окне, но и в специально подготовленной области памяти. Данная область называется картой пикселей и идентифицируется целым числом, имеющим тип `Pixmap`. Карта толщиной в один бит имеет специальное название – *битовая*.

1.2.1. Графический контекст

Прежде чем начать работу с графикой, программа должна выделить себе специальную структуру данных и получить указатель на нее. Эта структура называется *графическим контекстом* (Graphic Context (GC)). Указатель на GC используется в качестве одного из параметров при вызове «рисующих» функций X. Графический контекст содержит ряд атрибутов, влияющих на отображение объектов: текста, линий, фигур и др. Выделенный GC должен быть освобожден до завершения работы программы.

Графический контекст создается процедурой `XCreateGC()`, имеющей следующий прототип:

```
GC XCreateGC (Display *prDisplay, Drawable
nDrawable,
       unsigned long nValueMask, XGCValues
*prValues);
```

Первый аргумент – это указатель на структуру типа `Display`, который программа получает после вызова `XOpenDisplay()`; второй – идентификатор окна (или карты пикселей), в котором программа будет рисовать; третий – битовая маска, определяющая, какие атрибуты GC задаются; последний аргумент – структура типа `XGCValues`, определяемая следующим образом:

```
typedef struct {
   int               function;
   unsigned long     plane_mask;
   unsigned long     foreground;
   unsigned long     background;
   int               line_width;
   int               line_style;
   int               cap_style;
   int               join_style;
   int               fill_style;
   int               fill_rule;
   int               arc_mode;
```

```
    Pixmap          tile;
    Pixmap          stipple;
    int             ts_x_origin;
    int             ts_y_origin;
    Font            font;
    int             subwindow_mode;
    Bool            graphics_exposures;
    int             clip_x_origin;
    int             clip_y_origin;
    Pixmap          clip_mask;
    int             dash_offset;
    char            dashes;
} XGCValues;
```

Значения полей данной структуры будут объяснены ниже. Каждому из них соответствует бит в маске, которая передается в качестве третьего параметра при вызове процедуры `XCreateGC()`. Эти биты обозначаются символическими константами, определенными в файле `<X11/Xlib.h>`. Если бит установлен, то значение соответствующего атрибута должно быть взято из переданной `XCreateGC()` структуры `XGCValues`. Если бит сброшен, то атрибут приникает значение по умолчанию.

Следующий пример показывает процесс создания графического контекста, в котором устанавливаются два атрибута: цвет фона и цвет переднего плана.

```
. . . . . .
GC prGC;
XGCValues rValues;
Display prDisplay;
int nScreenNum;
. . . . . .
rValues.foreground = BlackPixel (prDisplay,
nScreenNum);
rValues.background = WhitePixel (prDisplay,
nScreenNum);
. . . . . .
prGC = XCreateGC (prDisplay, RootWindow
(prDisplay, nScreenNum),
```

```
(GCForeground | GCBackground), &rValues);
```

Вызов `XCreateGC()` – не единственный способ создания графического контекста. Так, например, новый контекст может быть получен из уже существующего GC с помощью `XCopyGC()`.

Когда контекст порожден, его атрибуты могут изменяться процедурой `XChangeGC()`. Например:

```
rValues.line_width = 10;
XChangeGC (prDisplay, prGC, GCLineWidth,
&rValues);
```

Приведенный фрагмент кода меняет ширину линий, рисуемых с помощью графического контекста.

Для того, чтобы получить значение полей GC, используется процедура `XGetGCValues()`.

1.2.2. Характеристики графического контекста

В предыдущем разделе мы говорили, что GC имеет ряд атрибутов, воздействующих на вывод изображений. Для текста это цвет и шрифт, для линий – цвет и толщина и т.д. Как уже упоминалось выше, атрибуты контекста задаются в момент его создания. Потом они могут меняться с помощью функции `XChangeGC()`. Кроме того, X поддерживает специальные функции для изменения параметров GC.

Ниже перечисляются основные характеристики графического контекста и процедуры, меняющие их.

Режим рисования (поле `function` в структуре `XGCValues`) указывает, каким образом комбинируются при рисовании цвет графики и цвет изображения, на которое накладывается графика. Данное поле задает некоторую логическую функцию. Возможные значения:

```
GXclear         0x0  0
GXand           0x1  src AND dst
GXandReverse    0x2  src AND NOT dst
GXcopy          0x3  src
GXandInverted   0x4  (NOT src) AND dst
```

```
      GXnoop           0x5  dst
      GXxor            0x6  src XOR dst
      GXor             0x7  src OR dst
      GXnor            0x8  (NOT src) AND (NOT
dst)
      GXequiv          0x9  (NOT src) XOR dst
      GXinvert         0xa  NOT dst
      GXorReverse      0xb  src OR (NOT dst)
      GXcopyInverted   0xc  NOT src
      GXorInverted     0xd  (NOT src) OR dst
      GXnand           0xe  (NOT src) OR (NOT
dst)
      GXset            0xf  1
```

По умолчанию `function` равно `GXcopy`. Устанавливается режим рисования с помощью процедуры `XSetFunction()`.

Изменяемые цветовые плоскости. Каждый пиксель задается с помощью `N` бит. Биты с одним номером во всех пикселях образуют как бы плоскости, идущие параллельно экрану. Получить число плоскостей для конкретного дисплея можно с помощью макроса `DisplayPlanes()`. Поле *plane_mask* структуры графического контекста определяет, в каких плоскостях идет рисование при вызове функций X. Если бит поля установлен, то при рисовании соответствующая плоскость изменяется, в противном случае она не затрагивается.

Цвет переднего плана и фона (поля `foreground` и `background`) задают цвета, используемые при рисовании линий текста и других графических элементов. Устанавливаются значения указанных полей функциями `XSetForeground()` и `XSetBackground()` соответственно.

Атрибуты, влияющие на рисование линий. Шесть параметров определяют вид прямых, дуг и многоугольников, изображаемых с помощью X Window.

5.Поле `line_width` задает толщину линии в пикселях.

Нулевое значение поля соответствует тому, что

линия должна быть толщиной в один пиксель и рисоваться с помощью наиболее быстрого алгоритма для данного устройства вывода.

6.Поле `line_style` определяет тип линии. Возможные значения следующие:

`LineSolid` – сплошная линия,

`LineOnOffDash` – пунктирная линия; промежутки между штрихами не закрашиваются;

`LineDoubleDash` – пунктирная линия; промежутки между штрихами закрашиваются цветом фона;

3.Параметр `cap_style` определяет вид линии в крайних точках, если ее ширина больше 1 пикселя. На рис. 1.4 приведены значения параметра и соответствующий вид конца линии.

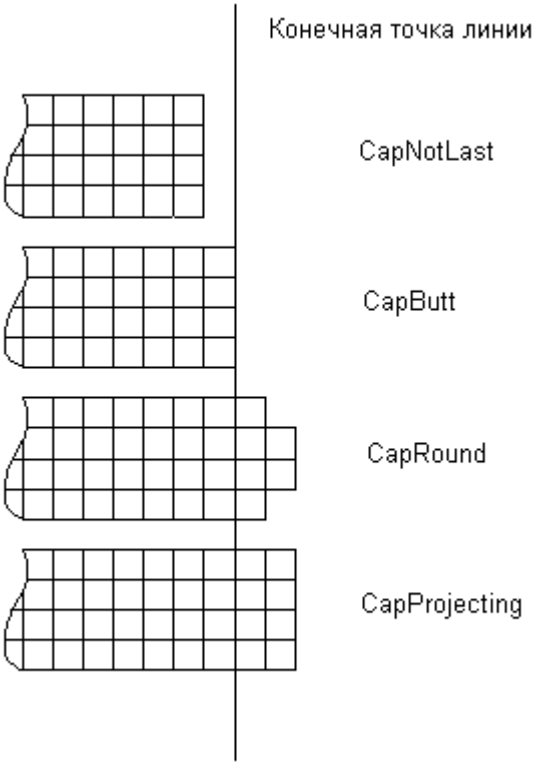


Рис. 1.4. Значения параметра `cap_style` графического контекста

4. Поле `join_style` определяет, как соединяются линии друг с другом. На рис. 1.5 показаны соответствующие возможности. Параметр имеет смысл при толщине линии большей 1.

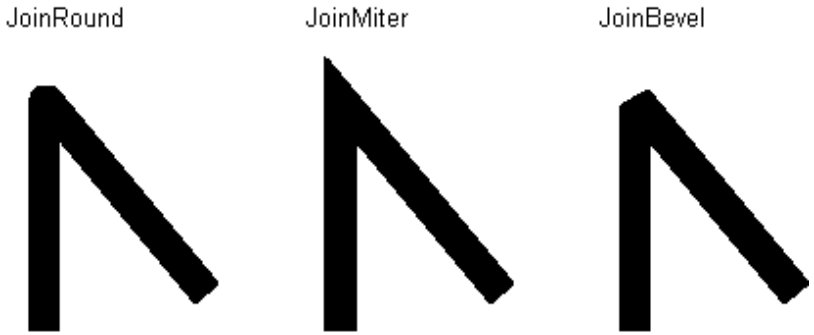


Рис. 1.5. Значения параметра `join_style` графического контекста

5. Если линия пунктирная, то поле `dashes` дает длину пунктира и промежутков в пикселях.
6. Параметр `dash_offset` указывает, с какого места начинать рисование первой черточки пунктирной линии.

Для установки параметров линии используется процедура `XSetLineAttributes()`.

Шрифт. Поле `font` определяет шрифт, используемый для вывода текста. Задать этот параметр можно с помощью процедуры `XSetFont()`.

Шаблоны, используемые для заполнения рисуемых фигур. Процесс рисования включает в себя два этапа. На первом определяются пиксели, которые должны быть

закрашены. После этого цвет выделенных точек изменяется. Так, для линии входящие в нее пиксели определяются по специальному алгоритму, а потом закрашиваются, например, цветом переднего плана.

Способ закраски определяется полем `fill_style`. Он устанавливается процедурой `XSetFillStyle()` и воздействует на все функции, рисующие линии, текст и фигуры. Исключение составляет случай, когда выводится линия, для которой значение `line_width` равно 0. Возможные значения параметра `fill_style` перечислены ниже.

`FillSolid` – для закраски используются цвета переднего плана и фона.

`FillTiled` – для закраски используется карта пикселей, определяемая параметром `tile` графического контекста; при этом карта как бы располагается в окне так, что ее левый верхний угол имеет координаты `ts_x_origin` и `ts_y_origin`; затем определяется ее пересечение с рисуемой графикой, и пиксели, попавшие в пересечение, закрашиваются; значения полей `ts_x_origin`, `ts_y_origin` устанавливаются процедурой `XSetTSOrigin()`; карта `tile` должна иметь ту же толщину (число бит-на-пиксел), что и окно, в котором производится рисование.

`FillStippled` – для закраски используется карта пикселей, задаваемая полем `stipple`; данная карта должна иметь толщину в 1 бит; способ закраски такой же, как и в случае `FillTiled` с той лишь разницей, что рисуются лишь те пиксели графики, которым соответствует установленный бит в карте `stipple`; цвет пикселя задается полем `foreground`.

`FillOpaqueStippled` – аналогично значению `FillStippled`, только пиксели, для которых не установлен бит в карте `stipple`, закрашиваются цветом

фона.

Для задания полей `tile` и `stipple` можно использовать карты любого размера. На некоторых устройствах при определенных размерах рисование идет намного быстрее. Для получения таких размеров можно использовать процедуры `XQueryBestSize()`, `XQueryBestStipple()`, `XQueryBestTile()`.

Режим заполнения многоугольников указывает, как заполнять цветом многоугольники, стороны которых пересекаются. Возможные значения следующие:

`EvenOddRule` – заполняются точки фигуры, определяемые по следующему правилу: пусть для некоторой линии растра n_1, n_2, ..., n_k – стороны многоугольника, которые ее пересекают; тогда закрашиваются точки между n_1 и n_2, n_3 и n_4, и т.д.

`WindingRule` – заполняется вся внутренность фигуры.

Режим заполнения дуг (поле `arc_mode`). Параметр задается процедурой `XSetArcMode()` и влияет на вид фигур, рисуемых процедурами `XFillArc()` и `XFillArcs()`.

Влияние подокон на рисование графических примитивов определяется полем `subwindow_mode`. Оно устанавливается процедурой `XSetSubwindowMode()` и имеет следующие значения:

`ClipByChildren` – часть графики, перекрываемая подокнами, не видна;

`IncludeInferiors` – графика рисуется поверх всех подокон.

Генерация запроса на перерисовку при копировании частей окон (поле `graphics_exposures`). Когда часть окна копируется куда-либо, то вполне вероятна ситуация, что исходное изображение перекрыто, возможно не полностью, другими окнами или недоступна по другим причинам. В этом случае может быть необходимо сообщить клиенту, в окно которого происходит

копирование, что часть нового изображения не может быть получена простым переносом пикселей, а должна быть перерисована. Если поле `graphics_exposures` равно `True`, то X посылает при копировании следующее:

– одно или несколько событий `GraphicsExpose`, если перерисовка необходима;
– событие `NoExpose`, если исходное окно полностью доступно и дополнительного рисования не требуется.

Если поле равно `False`, то событие не посылается. Устанавливается параметр процедурой `XSetGraphicsExposures()`.

Область отсечения (задается полями `clip_mask`, `clip_x_origin`, `clip_y_origin`). Это битовая карта, говорящая о том, какие пиксели выводятся, а какие нет при всех операциях рисования. Если бит карты установлен, то соответствующий пиксель появится в окне, а если бит сброшен, то пиксель будет пропущен. Положение в окне верхнего левого угла области отсечения определяется параметрами `clip_x_origin` и `clip_y_origin`.

Эти параметры устанавливаются процедурой `XSetClipOrigin()`. Сама область отсечения задается с помощью процедур `XSetClipMask()`, `XSetClipRectangles()` или `XSetClipRegion()`.

1.2.3. Вывод текста

Текст был и, видимо, будет важным средством информационного обмена между программами и пользователем. X Window позволяет выводить строки в любой части окна, используя большое количество шрифтов.

Для вывода текста используются процедуры `XDrawString()`, `XDrawImageString()` и `XDrawText()`. Каждая из них имеет две версии. Первая используется для шрифтов, имеющих не более 256 символов. Если же

символов больше («большие» шрифты), то применяется вторая версия. Функции, работающие с «большими» шрифтами, имеют имена `XDrawString16()`, `XDrawImageString16()` и `XDrawText16()`. Параметры процедур, выводящих текст, задают дисплей, окно, графический контекст, строку, ее положение и т.д. Рисование идет в соответствии с областью отсечения контекста. Буквы или их части, находящиеся за пределами области отсечения, не изображаются. Наиболее часто употребляется процедура `XDrawString()` (`XDrawString16()`). Ее параметры дают строку, ее длину и положение в окне. Текст рисуется цветом переднего плана, выбранного в GC.

Функция `XDrawImageString()` (`XDrawImageString16()`) похожа на предыдущую процедуру с той лишь разницей, что фон символов при рисовании закрашивается цветом фона, установленного в GC. `XDrawString()` и `XDrawImageString()` выводят символы, используя шрифт, установленный в GC.

`XDrawText()` (`XDrawText16()`) позволяет рисовать несколько строк сразу, используя при этом разные шрифты. Каждая рисуемая единица задается структурой `XTextItem`.

Процедура `XDrawText16()` использует структуру `XDrawText16`.

Поле `font` в приведенных структурах (`XTextItem` и `XDrawText16`) задает шрифт, используемый для рисования. Если значение поля `font` – `None`, то применяется шрифт, выбранный в GC.

Как мы уже говорили ранее, текст, как правило, рисуется шрифтом, выбранным в графическом контексте. X версии 11.4 и ниже поддерживает только растровые шрифты, а начиная с версии 11.5 и выше X Window имеет также и векторные шрифты.

В растровых шрифтах каждому символу соответствует некоторый битовый шаблон, определяющий порядок

закраски пикселей при рисовании. Если бит шаблона равен 1, то соответствующий элемент изображения закрашивается цветом переднего плана GC, если же он равен 0, то он закрашивается либо цветом фона, либо вообще не рисуется.

В векторных шрифтах каждый символ описывается последовательностью линий, которые, будучи составлены вместе, и дают его изображение. Размеры символов варьируются от шрифта к шрифту. Для их описания используется структура `XCharStruct`. Сам шрифт описывается структурой `XFontStruct`.

Перед тем, как выводить текст, используя тот или иной шрифт, последний должен быть загружен в X Window и выбран в графическом контексте.

Загрузка шрифта осуществляется процедурой `XLoadFont()`. Она берет в качестве аргумента имя шрифта, находит его и возвращает программе соответствующий идентификатор. Этот идентификатор передается затем процедуре `XSetFont()`, чтобы выбрать шрифт в GC. Заметим, что реально шрифт с данным именем загружается сервером лишь один раз. После этого при обращениях к `XLoadFont()` с тем же именем шрифта, функция возвращает ссылку на шрифт, уже находящийся в памяти компьютера.

По умолчанию X ищет файл со шрифтом в директории `/usr/lib/X11/fonts`. Программист может задать дополнительные директории для поиска с помощью процедуры `XSetFontPath()`.

Имя шрифта в X начинается с «-» и состоит из двух частей. Между ними стоит «--». В свою очередь, каждая из частей состоит из полей-слов, разделенных «-».

В первой части указывается следующее:

1)изготовитель шрифтам (foundry), например adobe;

2)семейство шрифта (font family), например courier, helvetica;

3)жирность шрифта (weight), например bold;
4)наклон шрифта (slant);
5)ширина букв шрифта (width).

Во второй части указывается следующее:

1)размер шрифта в пикселах (pixels);
2)размер шрифта в десятых долях «точки» («точка» равна 1/72 дюйма);
3)горизонтальное разрешение устройства, для которого разработан шрифт (horizontal resolution in dpi); величина измеряется в числе точек на дюйм;
4)вертикальное разрешение устройства, для которого разработан шрифт (vertical resolution in dpi); величина измеряется в числе точек на дюйм;
5)тип шрифта (spacing); возможные значения параметра следующие: *m* – шрифт с фиксированной шириной символов; *p* – пропорциональный шрифт с переменной шириной символов;
6)средняя ширина символов шрифта, измеренная в десятых долях пикселя (average width);
7)множество символов шрифта в кодировке ISO (International Standards Organisation) или других (character set).

Ниже приведен пример названия шрифта.

```
-adobe-courier-bold-o-normal--10-100-75-75-m-
60-iso8859-1
```

Части имени могут заменяться символом «*» или «?». В этом случае X подбирает шрифт, сличая имена имеющихся шрифтов с предоставленным шаблоном, так, как это делается при поиске файлов в UNIX. Например, шаблону

```
*charter-medium-i-*-240-*
```

соответствуют имена

```
-hit-charter-medium-i-normal-25-240-75-75-p-
```

```
136-iso8859-1
-hit-charter-medium-i-normal-33-240-100-75-p-
136-iso8859-1
```

Названия шрифтов, доступных в системе, хранятся в соответствующей базе данных. Получить список имен шрифтов можно с помощью процедуры `XListFonts()` или `XListFontsWithInfo()`. Список шрифтов, возвращаемый этими функциями, должен быть освобожден вызовом `XFreeFontNames()`.

Некоторые шрифты, такие как «`fixed`» или «`9x15`», доступны всегда.

Получить информацию о загруженном шрифте можно с помощью функции `XQueryFont()`, которая возвращает заполненную структуру типа `XFontInfo()`. Одновременно загрузить шрифт и получить информацию о нем можно с помощью процедуры `XLoadQueryFont()`.

Когда информация о шрифте больше не нужна, ее следует освободить с помощью `XFreeFontInfo()`. Когда становится не нужен и сам шрифт, последний надо «сбросить», обратившись к процедуре `XUnloadFont()`. Функция `XFreeFont()` объединяет в себе `XFreeFontInfo()` и `XUnloadFont()`.

Следующий фрагмент кода загружает шрифт «`courier`», создает GC и выводит с его помощью строку «Hello, world!».

```
Display *prDisplay;
GC prGC;
Window nWnd;
XFontStruct *prFontInfo;
. . . . . .
/* Загружаем шрифт */
if ( (prFontInfo=
  XLoadQueryFont(prDisplay, "*-courier-*" ))
== NULL){
  printf("Font not found!\n");
  exit(1);
```

```
}
. . . . . .
/* Создаем GC и рисуем строку */
prGC=XCreateGC(prDisplay, nWnd, 0, NULL);
XSetForeground (prDisplay, prGC,
BlackPixel(prDisplay, 0));
XSetFont (prDisplay, prGC, prFontInfo->fid);
XDrawString (prDisplay, nWnd, prGC, 10, 50,
"Hello, world!",
      strlen ("Hello, world!") );
XFreeGC (prDisplay, prGC);
. . . . . .
/* "Сбрасываем" шрифт */
XUnloadFont (prDisplay, prFontInfo->fid);
. . . . . .
```

Для корректного отображения символов кириллицы необходимо использовать один из локализованных шрифтов в той кодировке, которая поддерживается вашей системой (как правило, это koi8-r (koi8-u)). По умолчанию загружается первый из шрифтов, соответствующий шаблону, поэтому для корректного отображения текста с кириллицей необходимо в шаблоне указывать кодировку.

1.2.4. Использование цвета

Во времена не столь отдаленные экранные контроллеры могли поддерживать одновременно ограниченное количество цветов (вначале 16, позже 256). В связи с этим приложение не могло просто запросить рисование ярко-красным цветом, и ожидать, что этот цвет будет доступным. Каждое приложение распределяло цвета, которые ему были нужны, и когда все 16 или 256 цветовых элементов использовались, следующее распределение цвета заканчивалось неудачей.

В связи с этим появилось понятие «цветовой карты» – *палитры*. Палитра является таблицей того же размера, что и количество одновременно отображаемых данным экранным контроллером цветов. Каждый элемент палитры

содержит RGB (Красные, Зеленые и Синие) величины различных цветов (все цвета могут быть нарисованы, используя некоторую комбинацию красного, зеленого и синего).

Для того, чтобы сделать использование цветов близким к тому, которое предполагал программист, были введены функции выделения палитры. Вы можете попросить, выделить вас элемент палитры для цвета, заданного набором RGB-значений. Если он уже существовал, вы получите индекс в таблице. Если цвет не существовал, и таблица не заполнена, должна быть выделена новая ячейка палитры, содержащая данные значения RGB, и возвращен ее индекс. Если таблица была заполнена, процедура должна была закончиться неудачей. Вы могли затем запросить получение элемента палитры с цветом, ближайшим к требуемому. Это означает, что фактически рисование на экране будет произведено с использованием цвета, близкого к желаемому, но не того же самого.

На сегодняшних современных экранах, где работают сервера X с поддержкой миллионов цветов, эти ограничения кажутся устаревшими, но помните, что есть и старые компьютеры со старыми графическими картами внутри, равно как и 256-цветные X-терминалы. С использованием палитры, поддержка этих экранов становится прозрачной. На дисплее, поддерживающем миллионы цветов, любой запрос распределения цветового элемента будет удовлетворен. На дисплее, поддерживающем ограниченное количество цветов, некоторые цветовые запросы распределения должно возвращать подобные цвета. Они не выглядит столь хорошо, как требуемые, но ваше приложение все еще будет работать.

При рисовании с использованием Xlib можно выбрать стандартную палитру экрана, на котором отображается ваше окно, или создать новую палитру и применить ее для

окна. В последнем случае, всякий раз, когда мышь «наезжает» на ваше окно, экранная палитра заменится палитрой вашего окна, и вы увидите, что все другие окна на экране изменят свои цвета на нечто весьма экзотическое.

Для доступа к стандартной экранной палитре, определен макрос `DefaultColormap`,

```
Colormap screen_colormap =
     DefaultColormap(display,
DefaultScreen(display));
```

возвращающий дескриптор палитры, используемой по умолчанию на первом экране (напоминаем, что сервер X может поддерживать несколько различных экранов, каждый из которых может иметь свои собственные ресурсы).

Другой макрос, связанный с распределением новой палитры, работает так:

```
Visual*             default_visual             =
DefaultVisual(display,
DefaultScreen(display));
/* Создаем новую палитру, количество цветов в
которой       */
/*     определяется     количеством     цветов,
поддерживаемых данным экраном. */
Colormap              my_colormap              =
XCreateColormap(display,
                    win,
                    default_visual,
                    AllocNone);
```

Имейте в виду, что дескриптор окна используется только для того, чтобы позволить серверу X создать палитру для данного экрана. Мы можем затем использовать эту палитру для любого окна, нарисованного на том же экране.

Как только мы получили доступ к некоторой палитре, мы можем начать распределять цвета. Это делается с помощью функций `XAllocNamedColor()` и `XAllocColor()`. Первая из них – `XAllocNamedColor()` –

принимает имя цвета (например, «red», «blue», «brown» и т.д.) и распределяет ближайший цвет, который может в действительности рисоваться на экране. XAllocColor() принимает цвет RGB, и распределяет ближайший цвет, который может отображаться на экране. Обе функции используют структуру XColor, содержащую следующие поля:

unsigned long pixel – индекс палитры, используемый, для рисования данным цветом.

unsigned short red – красная составляющая RGB-значения цвета.

unsigned short green – зеленая составляющая RGB-значения цвета.

unsigned short blue – синяя составляющая RGB-значения цвета.

Пример использования этих функций:

```
/* Эта структура будет содержать выделенные
цветовые данные */
XColor system_color_1, system_color_2;
/* Эта структура будет содержать точные RGB-
значения именованных */
/* цветов, которые могут отличаться от
выделенных */
XColor exact_color;

/* Выделяем "красный" элемент палитры */
Status rc = XAllocNamedColor(display,
                screen_colormap,
                "red",
                &system_color_1,
                &exact_color);
/* проверяем успешность выделения */
if (rc == 0) {
  fprintf(stderr,
      "XAllocNamedColor - выделить 'красный'
цвет не удалось.\n");
}
```

```
else {
  printf("Элемент палитры 'красный' выделен
как (%d,%d,%d) в"
      " RGB-значениях.\n", system_color_1.red,
      system_color_1.green,
system_color_1.blue);
}

/* выделяем цвет со значениями (30000, 10000,
0) в RGB. */
system_color_2.red = 30000;
system_color_2.green = 10000;
system_color_2.blue = 0;
Status rc = XAllocColor(display,
            screen_colormap,
            &system_color_2);
/* проверяем успешность выделения */
if (rc == 0) {
 fprintf(stderr,
     "XAllocColor   -   цвет   (30000,10000,0)
выделить не удалось.\n");
}
else {
  /* что-то делаем с выделенным цветом... */
  .
  .
}
```

После того, как мы распределили желаемые цвета, мы можем использовать их, рисуя текст или графику. Для этого нам нужно установить эти цвета как передний план и цвет фона для некоторого GC (графического контекста), и затем используйте этот GC для рисования. Это делается с помощью функций `XSetForeground()` и `XSetBackground()`:

```
XSetForeground(display,                  my_gc,
screen_color_1.pixel);
XSetForeground(display,                  my_gc,
screen_color_2.pixel);
```

Само же рисование осуществляется с помощью тех же функций, что и ранее. Для использования нескольких цветов, можно сделать одно из двух: мы можем либо изменить передний план и/или цвет фона GC перед любой функцией рисования, либо использовать несколько различных GC. Решение, какой из способов лучше, принимать вам: распределение многих GC будет использовать больше ресурсов X сервера, но где-то это приведет к более компактному коду, и может быть легче, чем замена цветов рисования.

1.2.5. Битовые и пиксельные карты

Xlib не имеет никаких средств для работы с популярными графическими форматами, такими как gif, jpeg или tiff. На программиста (или высокоуровневые графические библиотеки) оставлен перевод эти форматы изображений в форматы, с которыми знаком Ч сервер – битовыми и пиксельными картами.

Битовая карта X – двухцветное изображение, сохраненное в формате, специфическом для X Window. Сохраненные в файле, данные битовой карты выглядят похожими на исходный файл на языке C. Он содержит переменные, определяющие ширину и высоту битового изображения, массив, содержащие битовые величины битового изображения (размер массива равен произведению ширины на высоту), и позицию горячей точки (опционально).

Пиксельная карта X – формат, используемый для хранения изображений в памяти X сервера. Этот формат может сохранять как черно-белые изображения (те же битовые карты), так и цветные изображения. Это единственный графический формат, поддерживаемый протоколом X, и любое изображение, которое должно рисоваться на экране, должно сначала быть переведено в этот формат.

В действительности, пиксельная карта X может трактоваться как окно, которое не появляется на экране. Многие графические операции, которые работают в окнах, точно также будут работать в пиксельных картах – достаточно подставить дескриптор пиксельной карты вместо дескриптора окна. В страницах справочного руководства видно, что все эти функции принимают `Drawable`, не `Window`, поскольку как окна так и пиксельные карты – рисуемые элементы, и они оба могут использоваться, чтобы рисовать в них такими функциями, как, например, `XDrawArc()`, `XDrawText()`, и т.п.

Один из способов загрузки битового изображение из файла в память – включение файла побитового изображения в программу директиву `#include` препроцессора языка C. Покажем, как можно получить доступ к файлу непосредственно:

```
/* эта переменная будет содержать дескриптор
новой пиксельной карты */
Pixmap bitmap;

/* эти переменные будут содержать размер
загружаемой битовой карты */
unsigned int bitmap_width, bitmap_height;

/* эти переменные будут содержать положение
горячей точки */
/* загружаемой битовой карты                */
int hotspot_x, hotspot_y;

/* эта переменная будет содержать дескриптор
корневого окна экрана, */
/* для которого мы хотим создать пиксельную
карту          */
Window root_win = DefaultRootWindow(display);

/*   загружаем   битовую   карту   из   файла
"icon.bmp", создаем */
```

```
/* пиксельную карту, содержащую свои данные в
сервере, */
/* и сохраняем ее дескриптор в переменной
bitmap    */
int rc = XReadBitmapFile(display, root_win,
             "icon.bmp",
             &bitmap_width, &bitmap_height,
             &bitmap,
             &hotspot_x, &hotspot_y);
/* проверяем, удалось ли создать пиксельную
карту */
switch (rc) {
  case BitmapOpenFailed:
    fprintf(stderr, "XReadBitmapFile – не могу
открыть файл"
            " 'icon.bmp'.\n");
    break;
  case BitmapFileInvalid:
    fprintf(stderr,  "XReadBitmapFile  -  файл
'icon.bmp' не содержит"
            "       корректного       битового
изображения.\n");
    break;
  case BitmapNoMemory:
    fprintf(stderr,   "XReadBitmapFile   -   не
хватает памяти.\n");
    break;
  case BitmapSuccess:
    /* битовая карта успешно загружена – что-
то делаем с ней... */
    .
    .
    break;
}
```

Имейте в виду, что параметр `root_win` не имеет ничего общего с данным битовым изображением – битовая карта не связывается с этим окном. Этот дескриптор окна использован только для определения экрана, для которого

мы хотим создать пиксельную карту. Это существенно, так как для того, чтобы быть полезной, пиксельная карта должна поддерживать то же количество цветов, что и экран делает.

Как только мы получили дескриптор пиксельной карты, сгенерированный из битового изображения, мы можем нарисовать ее в некотором окне, используя функцию `XCopyPlane()`. Эта функция позволяет указать, в какой рисуемой области (окне, или даже другой пиксельной карте) и в какой позиции будет отображена данная пиксельная карта.

```
/* Рисовать ранее загруженную битовую карту в
заданном окне, в */
/* позиции x=100, y=50. Мы хоти скопировать
всю битовую карту, */
/* поэтому указываем координаты x=0, y=0 для
копирования с   */
/* начала битового изображения и его полный
размер       */
XCopyPlane(display, bitmap, win, gc,
     0, 0,
     bitmap_width, bitmap_height,
     100, 50,
     1);
```

Мы могли также скопировать заданный прямоугольный фрагмент пиксельной карты вместо полного ее копирования. Последний параметр в функции `XCopyPlane()` определяет, какой слой (цветовую плоскость) исходного изображения мы хотим скопировать в целевое окно. Для битовых изображений всегда копируется плоскость номер 1.

Часто бывает необходимо создать неинициализированную пиксельную карту, чтобы в дальнейшем в ней можно было рисовать. Это полезно для графических редакторов (создание нового пустого «холста» вызовет создание новой пиксельной карты, в которой будет

храниться изображение). Это полезно при чтении различных форматов изображений – мы загружаем графические данные в память, создаем на сервере пиксельную карту, а затем рисуем расшифрованный графические данные на этот пиксельной карте.

```
/* эта переменная будет содержать дескриптор
новой пиксельной карты */
Pixmap pixmap;

/* эта переменная будет содержать дескриптор
корневого окна экрана, */
/* для которого мы хотим создать пиксельную
карту          */
Window root_win = DefaultRootWindow(display);

/* эта переменная будет содержать глубину
цвета создаваемой */
/* пиксельной карты – количество бит,
используемых для   */
/* представления индекса цвета в палитре
(количество цветов */
/* равно степени двойки глубины)
*/
int depth = DefaultDepth(display,
DefaultScreen(display));

/* создаем новую пиксельную карту шириной 30 и
высотой в 40 пикселей */
pixmap = XCreatePixmap(display, root_win, 30,
40, depth);

/* для полноты ощущений нарисуем точку в
центре пиксельной карты */
XDrawPoint(display, pixmap, gc, 15, 20);
```

После получения дескриптора пиксельной карты мы можем отобразить ее в некотором окне, используя функцию `XCopyArea()`. Эта функция позволяет указать устройство рисования (окно или даже другую пиксельную

карту) и в какое позиция этого устройства пиксельная карта будет отображена.

```
/* Рисовать ранее загруженную битовую карту в
заданном окне, в */
/* позиции x=100, y=50. Мы хоти скопировать
всю битовую карту, */
/* поэтому указываем координаты x=0, y=0 для
копирования с   */
/* начала битового изображения и его полный
размер       */
XCopyArea(display, bitmap, win, gc,
     0, 0,
     bitmap_width, bitmap_height,
     100, 50);
```

Мы могли также скопировать заданный прямоугольный фрагмент пиксельной карты вместо полного ее копирования.

Отметим, что на одном и том же экране возможно создавать пиксельные карты различных глубин. Когда мы выполняем операции копирования (пиксельной карты в окно и т.п.), мы должны убедиться, что источник и приемник имеют одну и ту же глубину. Если их глубина различается, операция не удастся. Единственное исключение – копирование указанной битовой плоскости пиксельной карты с помощью показанной ранее функции `XCopyPlane()`. В этом случае мы можем скопировать указанную плоскость в окно-приемник – в действительности устанавливается указанный бит в цвете каждого копируемого пикселя. Это может быть использовано для создания забавных графических эффектов.

Наконец, когда все операции над данной пиксельной картой выполнены, ее необходимо освободить, чтобы освободить ресурсы X сервера. Это делается с помощью функции `XFreePixmap()`:

```
/* освобождение пиксельной карты с заданным
дескриптором */
```

```
XFreePixmap(display, pixmap);
```

1.2.6. Изменение формы мышиного курсора

Программы часто модифицируют форму указателя мыши (также называемого указателем X) в зависимости от своего состояния. Например, занятое приложение часто отображает над своим основным окном песочные часы, чтобы дать пользователю визуальный намек, что он должен ожидать. Без такого визуального намека пользователь мог бы подумать, что приложение зависло.

Есть два основных метода создания курсоров. Первый из них – использование набора предопределенных курсоров, поставляемых с Xlib. Второй – использование битовых изображений, определенных пользователем.

В первом методе используется специальный шрифт «`cursor`» и функция `XCreateFontCursor()`. Эта функция принимает идентификатор формы, и возвращает дескриптор на созданный курсор. Список возможных шрифтовых идентификаторов находится в файле include `<X11/cursorfont.h>`. Всего их более 70; вот некоторые из таких курсоров:

`XC_arrow` – обычный курсор в форме стрелки, отображаемый сервером.

`XC_pencil` – курсор в форме карандаша.

`XC_watch` – песочные часы.

Создать курсор с использованием этих идентификаторов несложно:

```
#include <X11/cursorfont.h> /* определяет
XC_watch и т.п. */
/* эта переменная содержит дескриптор
создаваемого курсора */
Cursor watch_cursor;

/* создаем курсор "песочные часы" */
watch_cursor = XCreateFontCursor(display,
XC_watch);
```

Другой метод создания курсора – использование пары пиксельных карт глубиной 1. Одна пиксельная карта определяет форму курсора, а другая работает как маска, определяющая, какие пиксели курсора действительно будут нарисованы. Остальная часть пикселей будет прозрачной. Создание такого курсора осуществляется с помощью функции `XCreatePixmapCursor()`. В качестве примера создадим курсор, используя битовое изображение "icon.bmp". Будем предполагать, что оно уже загружено в память и преобразовано в пиксельную карту, дескриптор которой сохранен в переменной `bitmap`. Мы хотим, что оно было полностью прозрачным. Это означает, что только черные фрагменты нарисуются, а белые будут прозрачными. Чтобы достигнуть такого эффекта, будем использовать иконку и как пиксельную карту курсора, и как маску пиксельной карты.

```
/* эта переменная содержит дескриптор
создаваемого курсора */
Cursor icon_cursor;

/* вначале необходимо определить основной и
фоновый цвета курсора */
XColor cursor_fg, cursor_bg;

/* получаем доступ к палитре экрана по
умолчанию */
Colormap screen_colormap =
        DefaultColormap(display,
DefaultScreen(display));

/* выделяем черный и белый цвета */
Status rc = XAllocNamedColor(display,
                screen_colormap,
                "black",
                &cursor_fg,
                &cursor_fg);
if (rc == 0) {
```

```
  fprintf(stderr,
    "XAllocNamedColor - невозможно
распределить цвет 'black'\n");
  exit(1);
}
Status rc = XAllocNamedColor(display,
               screen_colormap,
               "white",
               &cursor_bg,
               &cursor_bg);
if (rc == 0) {
  fprintf(stderr,
    "XAllocNamedColor - невозможно
распределить цвет 'white'\n");
  exit(1);
}

/* Наконец, создаем курсор. Горячую точку
устанавливаем ближе к */
/* верхнему левому углу курсора - позиции
(x=5, y=4). */
icon_cursor = XCreatePixmapCursor(display,
bitmap, bitmap,
                    &cursor_fg, &cursor_bg,
                5, 4);
```

Когда мы определяем курсор, необходимо определить, какой пиксель курсора является указателем, доставляемым пользователю в различные события от мыши. Обычно, мы выберем позицию курсора, которая визуально выглядит похожей на «горячую точку». Например, на курсоре в виде стрелки конец стрелки будет определен как горячая точка.

Когда курсор больше не нужен, его необходимо освободить, используя функцию `XFreeCursor()`:

```
XFreeCursor(display, icon_cursor);
```

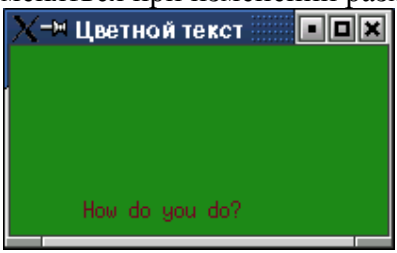
После того, как курсор создан, необходимо сообщить X серверу об окне, к которому он должен быть подключен. Это делается с помощью `XDefineCursor()`, и заставляет сервер X менять указатель мыши на форму этого курсора

всякий раз, когда указатель мыши перемещается внутри этого окно. Мы можем отключить этот курсор от нашего окна с помощью функции `XUndefineCursor()`, которая заставит отображаться встроенный курсор.

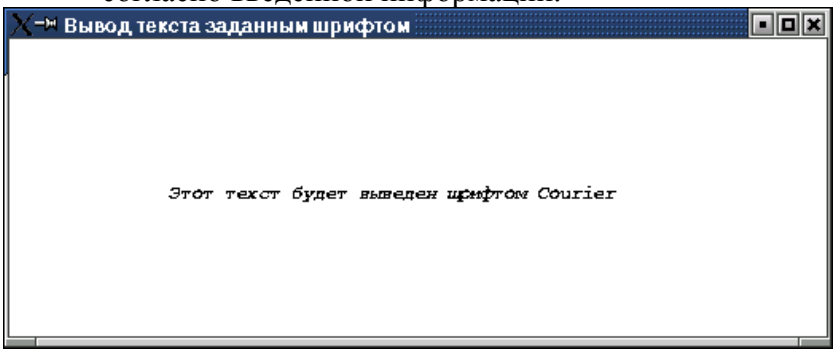
```
/* прикрепить курсор к окну */
XDefineCursor(display, win, icon_cursor);

/* отключить курсор от окна */
XUndefineCursor(display, win);
```

1.2.7. Лабораторная работа №2 «Текст и графика»

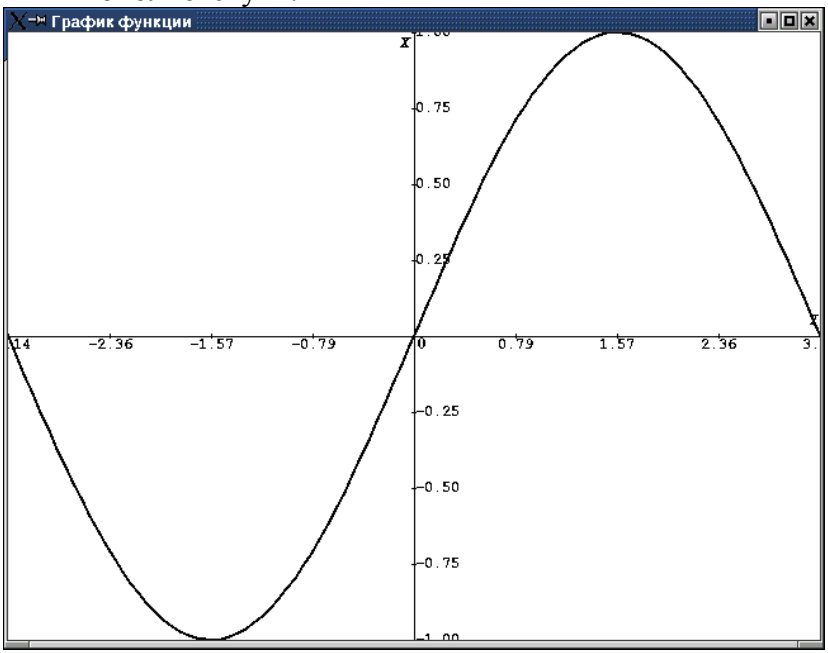
1. Напишите программу, выводящую текстовое сообщение в произвольную позицию (в пределах окна) произвольным цветом. Цвет и координаты должны меняться при изменении размеров окна.



2. Составьте программу, принимающую со стандартного ввода маску шрифта, выводимую строку, координаты *x*, *y* и отображающую окно с текстом согласно введенной информации.



3.Нарисуйте в окне график функции sin(x) на отрезке [–π; π]. Оси подпишите курсивом, метки по осям – обычным шрифтом, начало координат (0) выделите жирным шрифтом. Не забудьте добавить в перечень используемых библиотек математическую библиотеку m.



4.Нарисуйте в окне 100 окружностей. Цвет, координаты центра и радиус выбирать случайным образом.

1.3. Работа с внешними устройствами

1.3.1. Клавиатура

Как и большинство интерактивных программ, задачи, выполняющиеся в X Window, активно используют для ввода информации клавиатуру компьютера. Когда пользователь нажимает или отпускает клавишу, сервер получает соответствующий сигнал, который преобразуется в событие и отправляется в очередь программы, имеющей фокус ввода (input focus).

Поясним, что такое фокус ввода. Дело в том, что клавиатура у машины одна, и она разделяется всеми выполняющимися одновременно программами. Но в каждый момент времени поступающий от устройства сигнал доступен лишь одной из них, как правило, той, которой принадлежит активное окно. В этом случае говорят, что программа и ее окно имеют *фокус ввода*. Последний может переходить от окна к окну и от программы к программе.

Когда окно получает фокус, соответствующей программе посылается событие `FocusIn`, при потере – приходит событие `FocusOut`.

Когда пользователь нажимает клавишу клавиатуры, программа получает событие `KeyPress`. Сервер также может послать событие `KeyRelease`, когда клавиша отпускается, но это справедливо не для всех типов компьютеров.

Оба этих события сопровождаются структурой типа `XKeyEvent`. Ее поле `keycode` содержит код нажатой клавиши, а поле `state` – состояние клавиш-модификаторов и кнопок мыши. Модификаторами называются такие клавиши, как `Shift`, `Ctrl`, `Caps Lock`. Кроме этого, X предусматривает наличие дополнительных модификаторов, которые обозначаются `Mod1`, ..., `Mod5`. Каждой нажатой

клавише-модификатору и кнопке мыши соответствует флаг в поле state.

Коды, передаваемые через поле keycode структуры XKeyEvent, однозначно идентифицируют клавиши. Их конкретные значения зависят от типа машины и клавиатуры. Эти коды мы будем называть *физическими*. Чтобы обеспечить переносимость программ, сервер устанавливает соответствие между физическими кодами клавиш, которые могут меняться от компьютера к компьютеру, и целочисленными константами – *логическими* кодами (символами). Они имеют предопределенные значения, которые приведены в файле <X11/keysymdef.h> и начинаются с префикса «XK_». Так, букве «a» соответствует символ XK_a, клавише <Return> (<Enter>) – символ XK_Return и т.д.

Для разных алфавитов X поддерживает разные множества логических кодов. Возможные типы алфавитов перечисляются в файле <X11/keysym.h>.

Одному коду клавиши может соответствовать несколько символов в зависимости от состояния клавиш-модификаторов. Процедура

```
KeySym XKeycodeToKeysym (Display* prDisplay,
KeyCode nKeycode,
    int nIndex);
```

позволяет по коду nKeyCode получить соответствующий ему символ с номером nIndex. Если nIndex равен 0, то полученный символ соответствует просто нажатой клавише. Если nIndex равен 1, то возвращается символ, соответствующий ситуации, когда клавиша нажата одновременно с Shift.

Функция XKeysymToKeycode() осуществляет обратное преобразование.

Программа может получить карту соответствия кодов и символов, обратившись к процедуре XGetKeyboardMapping().

Изменяется соответствие физических и логических кодов процедурой XChangeKeyboardMapping(). Следующая последовательность операторов ставит клавише <F2> в соответствие символ XK_F3.

```
........
Keysym     nF2Sym, nF3Sym;
int   nF2Keycode;
Display    *prDisplay;
........
nF2Sym     = XStringToKeysym ("F2");
nF3Sym     = XStringToKeysym ("F3");
nF2Keycode   = XKeysymToKeycode (prDisplay,
nF2Sym);
XChangeKeyboardMapping (prDisplay, nF2Keycode,
1, &nF3Sym, 1);
........
```

Здесь использована процедура XStringToKeysym(), которая по строке «str» возвращает соответствующий символ XK_str.

Когда соответствие кодов меняется, всем работающим в настоящее время клиентам посылается событие MappingNotify.

Клавиши-модификаторы также имеют логические коды. Клавишам Shift сопоставлены символы XK_Shift_L и XK_Shift_R; Caps Lock соответствует XK_CapsLock; Control – XK_Control_L; Mod1 – XK_Meta_L и XK_Meta_R. Символы остальных модификаторов (Mod2 – Mod5) не определены. X содержит набор специальных процедур, которые позволяют получить и установить соответствие код-символ для модификаторов. Эти функции следующие: XGetModifierMapping(), XInsertModifiermapEntry(), XDeleteModifiermapEntry(), XSetModifierMapping().

X не останавливается на задании соответствия код клавиши – символы, а идет дальше. Система позволяет

программе сопоставить любой комбинации модификаторов и клавиш (например, <Shift+Ctrl+A>) ASCII строку (например, «EXIT»). Для некоторых клавиш соответствующие строки задаются сервером по умолчанию. Так, символу XK_A соответствует строка «A».

Макрос XRebindKeysym() берет символ, список модификаторов и сопоставляет им строку.

Процедура XLookupString(), наоборот, берет событие о нажатии (отпускании) клавиши и возвращает соответствующие ему символ и строку. Последний ее параметр – указатель на структуру типа XComposeStatus. Дело в том, что некоторые клавиатуры имеют специальную клавишу Compose, которая позволяет печатать символы, которым нет соответствия среди клавиш. Специальная таблица указывает, какой символ должен быть создан, если обычная клавиша нажимается одновременно с Compose. Ссылка на эту информацию и возвращается в структуре XComposeStatus.

Ниже приводится фрагмент программы, которая распознает функциональные клавиши <F1>-<F5>, и при их нажатии печатает соответствующую строку. Программа также сопоставляет комбинации <Shift+Control+A> строку «EXIT». Эта комбинация используется для завершения программы.

```
........
Display    *prDisplay;
int   nScreenNum;
GC    prGC;
XEvent     rEvent;
Window     nWnd;
char  sKeyStr[20];
KeySym     nKeySym, naModList[2];
int   n;

/* Устанавливаем связь с сервером, получаем
номер экрана . . . */
```

```
.........
/* Задаем соответствие символ-строка */
naModList[0] = XK_Control_L;
naModList[1] = XK_Shift_L;
XRebindKeysym (prDisplay, XK_F6, naModList, 2,
"EXIT",
   strlen ("EXIT"));
  /* Цикл получения и обработки событий */

while (1) {
  XNextEvent (prDisplay, &rEvent);
  switch (rEvent.type) {
   ......
   case KeyPress :
     /* Очищаем строку */
     memset (sKeyStr, 0, sizeof (sKeyStr));

     /* Получаем строку, соответствующую
событию */
     XLookupString (&rEvent.xkey, sKeyStr),
       sizeof (sKeyStr), &nKeySym, NULL);
     if ( !strcmp (sKeyStr, "EXIT"))
     {
      XFreeGC (prDisplay, prGC);
      XCloseDisplay (prDisplay);
      exit (0);
     }

     n = nKeySym == XK_F1 ? 1 :
       nKeySym == XK_F2 ? 2 :
       nKeySym == XK_F3 ? 3 :
       nKeySym == XK_F4 ? 4 :
       nKeySym == XK_F5 ? 5 : 0;

     if (n) {
      sprintf (sKeyStr, "F%d pressed.", n);
      XClearWindow (prDisplay, nWnd);
      XDrawString (prDisplay, nWnd, prGC, 10,
50,
```

```
            sKeyStr, strlen (sKeyStr));
      }
      break;
   }
}
........
```

Сервер имеет ряд атрибутов, воздействующих на обработку сигналов клавиатуры. Получить их можно с помощью функции `XGetKeyboardControl()`. Она возвращает указанные параметры в переменной, имеющей тип `XKeyboardState`, определенный следующим образом:

```
typedef struct {
 int key_click_percent;
 int bell_percent;
 unsigned int bell_pitch, bell_duration;
 unsigned long led_mask;
 int global_auto_repeat;
 char auto_repeats[32];
} XKeyboardsState;
```

Поле `key_click_percent` указывает, имеет ли нажатие клавиши звуковое сопровождение; значения поля задаются в %; 0 – звукового сопровождения нет, 100 – громкий звук. Поле `bell_percent`, `bell_pitch` и `bell_duration` указывают, какую силу, частоту и продолжительность имеет предупреждающий сигнал, возникающий при нажатии некоторых клавиш.

Некоторые клавиатуры используют для клавиш-модификаторов световую подсветку. Поле `led_mask` представляет собой комбинацию флагов, показывающую, для каких клавиш эта подсветка используется.

Когда клавиша нажата и удерживается, то сервер может автоматически имитировать ее повторное нажатие. Поле `global_auto_repeat` определяет, делает это сервер или нет. Особенностью X является то, что автоматическую генерацию событий о нажатии можно разрешать или запрещать для отдельных клавиш. Массив `auto_repeats`

содержит информацию о том, для каких клавиш автоповтор включен, а для каких нет. Каждый бит массива соответствует клавише с определенным физическим кодом. Если бит установлен, то генерация разрешена, если сброшен, то запрещена. Каждый байт `N` массива содержит биты для клавиш с кодами от `8N` до `8N+7`.

Изменить параметры клавиатуры можно с помощью `XChangeKeyboardControl()`.

Желаемые установки передаются через переменную, которая указывает на структуру типа `XKeyboardControl`, определяемую следующим образом:

```
typedef struct {
 int key_click_percnet;
 int bell_percent;
 int bell_pitch;
 int bell_duration;
 int led;
 int led_mode;
 int key;
 int auto_repeat_mode;
} XKeyboardControl;
```

Первые четыре поля совпадают с аналогичными полями структуры `XKeyboardState`. Поля `led` и `led_mode` позволяют сообщить серверу, какие из клавиш-модификаторов должны сопровождаться подсветкой. Если поле `led` не задано, и `led_mode` равно `LedModeOn`, то изменяется состояние всех клавиш, для которых поддерживается световое сопровождение. Если `led_mode` равно `LedModeOff`, то состояние клавиш не меняется. Если поле `led` задано, то это есть комбинация флагов, указывающих, для каких клавиш подсветку включить (`led_mode` равно `LedModeOn`) или выключить (`led_mode` равно `LedModeOff`).

Поля `key` и `auto_repeat_mode` определяют, для какой клавиши (клавиш) включить (`auto_repeat_mode` равно

AutoRepeatModeOn) или выключить (auto_repeat_mode равно AutoRepeatModeOff) режим автоматического повтора. Если поле key задано, то автоматический повтор включается или выключается только для клавиши с кодом key.

1.3.2. Мышь

С точки зрения программы общение с мышью похоже на работу с клавиатурой. X получает сигналы от устройства, преобразует их в события и помещает последние в очередь программы. Однако есть и существенная разница. Если события от клавиатуры передаются лишь программе, окно которой имеет фокус ввода, то события от мыши могут передаваться, в принципе, любой задаче, окно (окна) которой присутствуют на экране.

Чаще всего приходится обрабатывать события нажатия (отпускания) кнопки мыши. Для регистрации такого типа событий, необходимо добавить одну из следующих масок с помощью функции XSelectInput():

ButtonPressMask – уведомлять о нажатии любой кнопки в одном из окон программы.

ButtonReleaseMask – уведомлять об отпускании любой кнопки в одном из окон программы.

В цикле обработки сообщений могут проверяться такие события:

ButtonPress –нажата кнопка в одном из окон программы.

ButtonRelease – отпущена кнопка в одном из окон программы.

Структура для этих сообщений получается доступом к полю xbutton объединения XEvent и содержит, в частности, такие поля:

Window window – идентификатор окна, которому было послано сообщение (в случае, если оно было

зарегистрировано для нескольких окон программы).

`int x, y` – координаты x и y (в пикселях) мышиного курсора в момент нажатия.

`int button` – номер нажатой кнопки (может принимать значения Button1, Button2, Button3).

`Time time` – время (в миллисекундах), которое длилось событие. Может использоваться для определения «двойного щелчка».

В качестве примера приведем фрагмент кода, в котором рисуется черный пиксель в позиции мыши всякий раз, когда мы получаем событие «нажатие кнопки» от первой кнопки мыши, и стирается пиксель (то есть рисуется белый), когда нажата вторая кнопка мыши. Предполагается существование двух GC: gc_draw с черным цветом переднего плана и gc_erase с белым цветом переднего плана.

```
. . . . . .
 case ButtonPress:
  /* сохраняем координаты кнопки мыши в целых
переменных */
  /* также сохраняем идентификатор окна, в
котором была */
  /* нажата кнопка мыши                    */
  x = an_event.xbutton.x;
  y = an_event.xbutton.y;
  the_win = an_event.xbutton.window;

  /* проверяем, какая из кнопок была нажата,
*/
  /* и действуем соответственно        */
  switch (an_event.xbutton.button) {
    case Button1:
      /* рисуем пиксель в позиции мыши */
      XDrawPoint(display, the_win, gc_draw, x,
y);
      break;
    case Button2:
```

```
      /* стираем пиксель в позиции мыши */
      XDrawPoint(display,  the_win,  gc_erase,
x, y);
      break;
    default:  /*  возможно,  третья  кнопка  -
игнорируем */
      break;
  }
  break;
. . . . . .
```

Подобно событиям нажатия и отпускания кнопки мыши, нас также могут извещать о различных событиях перемещения мыши. Они делятся на два семейства. Первое – перемещение указателя мыши, пока никакие кнопки не нажимаются, и второе – движение указателя мыши при одной (или более) нажатых кнопках (это иногда называется операцией «перетаскивания» (drag)). Следующие маски событий должны быть добавлено в вызов `XSelectInput()` для получения извещений о таких событиях:

`PointerMotionMask` – события указателя, перемещающегося в одном из окон программы, когда ни одна кнопка мыши не нажата.

`ButtonMotionMask` – события перемещения указателя, пока одна (или более) кнопок мыши удерживается нажатой.

`Button1MotionMask` – тоже, что и `ButtonMotionMask`, но только когда первая кнопка мыши удерживается нажатой.

`Button2MotionMask`, `Button3MotionMask`, `Button4MotionMask`, `Button5MotionMask` – аналогично кнопок 2, 3, 4 или 5.

В цикле обработки сообщений проверяется событие `MotionNotify` – указатель мыши перемещался в одном из окон, для которых мы запросили уведомление о таких событиях.

Структура для этих сообщений получается доступом к полю `xmotion` объединения `XEvent` и содержит, в

частности, такие поля:

`Window window` – идентификатор окна, которому было послано сообщение движения мыши (в случае, если оно было зарегистрировано для нескольких окон программы).

`int x, y` – координаты x и y (в пикселях) мышиного курсора в момент генерации сообщения.

`unsigned int state` – маска кнопок (или клавиш), удерживаемых во время этого события (если таковые имеются). Эта поле –побитовое «ИЛИ» любого из следующих значений: `Button1Mask`, `Button2Mask`, `Button3Mask`, `Button4Mask`, `Button5Mask`, ShiftMask, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, `Mod5Mask`. Первые пять значений ссылаются на кнопки мыши, которые нажимаются, остальные соответствуют различным специальным клавишам (`Mod1` – обычно клавиша `Alt` или `Meta`).

`Time time` – время (в миллисекундах), которое длилось событие.

Как пример, следующий код определяет режим рисования для графического редактора, в котором, если пользователь перемещает мышь, удерживая первую ее кнопку, мы рисуем на экране. Этот код имеет недостаток: поскольку перемещение мыши может генерировать много событий, вполне возможно, что мы не получим событие движения мыши для каждого пикселя, над которым проходит мышь. Один из способов разрещения этой ситуации состоит в запоминании последнего пикселя, над которым была «протащена» мышь, и рисованием линии между запомненной и новой позициями указателя мыши.

```
. . . . . .
case MotionNotify:
  /* сохраняем координаты кнопки мыши в целых
переменных */
  /* также сохраняем идентификатор окна, в
котором была */
```

```
  /* протащена мышь                    */
  x = an_event.xmotion.x;
  y = an_event.xmotion.y;
  the_win = an_event.xbutton.window;

  /* если первая кнопка мыши удерживалась во
время этого события, */
  /* рисуем пиксель в позиции мышиного курсора
*/
  if (an_event.xmotion.state & Button1Mask) {
    XDrawPoint(display, the_win, gc_draw, x,
y);
  }
  break;
. . . . . .
```

Другой тип мышиных событий – вход указателя мыши в окно программы или выход из окна. Некоторые программы используют эти события, чтобы показать пользователю, что приложение получило фокус. Для регистрации событий этого типа необходимо добавить один (или более) из следующих масок в функции `XSelectInput()`:

`EnterWindowMask` – уведомлять, когда указатель мыши входит в любое из окон программы.

`LeaveWindowMask` – уведомлять, когда указатель мыши выходит из окна программы.

В цикле обработки сообщений проверяется одно из следующих событий?

`EnterNotify` – указатель мыши только что вошел в одно из окон программы.

`LeaveNotify` – указатель мыши только что вышел из окна программы.

Структура для этих сообщений получается доступом к полю `xcrossing` объединения `XEvent` и содержит, в частности, такие поля:

`Window window` – идентификатор окна, которому было

послано сообщение от мыши (в случае, если оно было зарегистрировано для нескольких окон программы).

`Window subwindow` – идентификатор дочернего окна ребенка, из которого мышь перешла в текущее (в событии `EnterNotify`), или в которое указатель мыши переместился (в событии `LeaveNotify`), или `None`, если мышь переместилась за пределы окон программы.

`int x, y` – координаты x и y (в пикселях) мышиного курсора в момент генерации сообщения.

`int mode` – номер нажатой кнопки (может принимать значения Button1, Button2, Button3).

`Time time` – время (в миллисекундах), которое длилось событие. Может использоваться для определения «двойного щелчка».

`unsigned int state` – маска кнопок (или клавиш), удерживаемых во время этого события (если таковые имеются). Эта поле –побитовое «ИЛИ» любого из следующих значений: `Button1Mask`, `Button2Mask`, `Button3Mask`, `Button4Mask`, `Button5Mask`, ShiftMask, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, `Mod5Mask`. Первые пять значений ссылаются на кнопки мыши, которые нажимаются, остальные соответствуют различным специальным клавишам (`Mod1` – обычно клавиша `Alt` или `Meta`).

`Bool focus` – устанавливается в `True`, если окно имеет клавиатурный фокус, и `False` в противном случае.

Обычно фокус ввода может свободно переходить от окна к окну. Но иногда программе необходимо запретить передачу фокуса. Это называется захватом клавиатуры. Для того, чтобы его реализовать, используется процедура `XGrabKeyboard()`.

Функция `XGrabKey()` запрещает передачу фокуса после нажатия определенной комбинации клавиш. Освободить клавиатуру можно, обратившись к процедуре

XUngrabKeyboard() (XGrabKey()).

Рассмотрим поведение системы при обработке событий от мыши. Как правило, если ее кнопка нажата в момент, когда ее курсор находится в неактивном окне, то последнее активизируется, и события от мыши передаются ему. Сказанное означает, что в нормальном состоянии окно получает только те события от мыши, которые соответствуют сигналам, пришедшим тогда, когда ее курсор находится в пределах окна. Но если программа вызывает

```
int XGrabPointer (Display* prDisplay, Window
nGrabWnd,
  Bool nOwnerEvents, unsigned int nEventMask,
  int nPointerMode, int nKeyboardMode, Window
nConfineTo,
  Cursor nCursor, Time nTime);
```

то положение меняется. Теперь все события будут направляться окну с дескриптором nGrabWnd. Освобождается мышь вызовом XUngrabPointer(). Процедура XGrabButton() указывает, что курсор должен быть захвачен после нажатия определенной кнопки. Обратной к ней является процедура XUngrabButton().

Процедуры, захватывающие устройство, – мышь или клавиатуру – имеют ряд аргументов, влияющих на поведение системы.

Так, параметр nConfineTo есть идентификатор окна, за пределы которого не должен выходить курсор мыши, если он захвачен.

Если аргумент nOwnerEvents равен True, то события мыши будут передаваться окнам программы. Если nOwnerEvents – False, или курсор находится в окне, не принадлежащем программе, то события мыши передаются окну nGrabWnd.

Если nOwnerEvents равен False, то параметр nEventMask указывает, какие события следует передавать

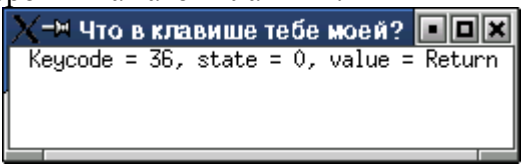
окну nGrabWnd.

Обработка событий от клавиатуры или ныши может быть приостановлена, если nPointerMode или nKeyboardMode равен GrabModeSync. В этом случае события буферизуются сервером, пока устройство не будет освобождено с помощью XUngrabKeyboard(), XUngrabKey(), XUngrabPointer() или XUngrabButton().

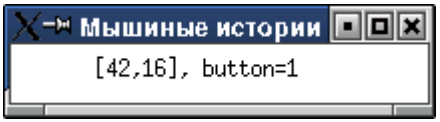
Параметр nCursor задает форму курсора во время того, как мышь захвачена. Аргумент nTime указывает, когда система должна активизировать режим захвата.

1.3.3. Лабораторная работа №3 «Работа с внешними устройствами»

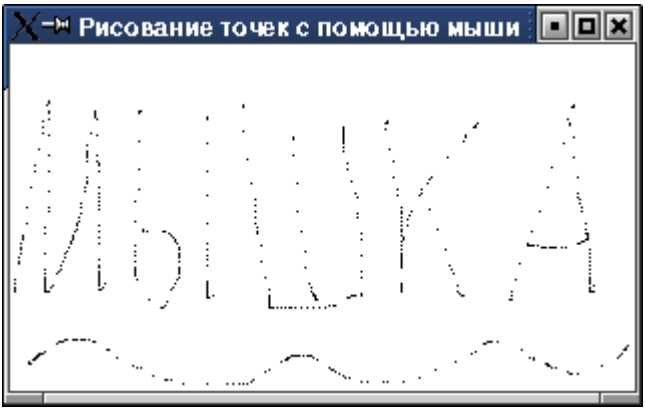
1. Используя функции XKeysymToString() и XKeycodeToKeysym(), напишите программу, которая реагирует на нажатие клавиш в окне выдачей в него кода символа, состояния модификаторов и символьной расшифровки нажатой клавиши.



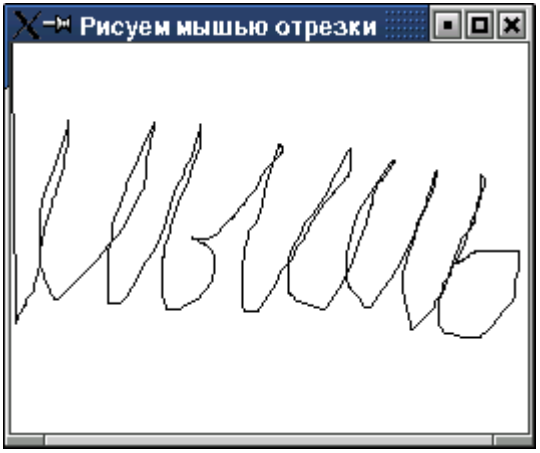
2. Напишите программу, определяющую координаты мыши в момент нажатия кнопки и печатающую в позицию мышиного курсора координаты мыши и номер нажатой кнопки.



3. Модифицируйте предыдущую программу для рисования точек в местах нажатий мыши и при ее движении с нажатой кнопкой.



4. Модифицируйте предыдущую программу для рисования отрезков между нажатиями мыши и при ее движении с нажатой кнопкой.



1.4. Программы и их ресурсы

Многие программы имеют различные заранее подготавливаемые данные, которые в терминах X называются – ресурсами. Это могут быть цвета окон приложения, строки сообщений пользователю и т.д.

Как правило, программисты создают ресурсы каждый по-своему. В X Window сделана попытка унифицировать этот процесс.

1.4.1. Формат файла ресурсов

В X файл ресурсов есть обычный текстовый файл, каждая строка которого задает тот или иной параметр (ресурс) программы. (При этом предполагается, что программу «населяют» именованные объекты, связанные в некоторую иерархию). Общий вид строки следующий:

```
<имя программы>.<подобъект1>.<подобъект2>. . .
        <подобъектN>.<имя ресурса>: <значение ресурса>
```

Подобная строка задает значение ресурса для подобъектов иерархии объектов программы. Например, запись

```
myprog.dialogwnd.background: Red
```

говорит, что в программе `myprog` у объекта с именем `dialogwnd` параметр `background` (цвет фона) имеет значение `Red` (красный цвет).

Вместо имен объектов могут указываться их классы. Обычно класс имеет то же самое имя, что и объект, но начинается с заглавной буквы, например,

```
Myprog.dialogwnd.Background: Red
```

Часть объектов или классов в левой части строки, задающей ресурс, может заменяться символом '*', например, строка

```
myprog*background: Red
```

указывает, что для всех объектов программы `myprog` ресурс `background` имеет значение `Red`.

Связка с помощью символа '.' имеет больший приоритет, чем связка с помощью '*'. Так, если в файле, задающем ресурсы, есть две строки

```
myprog*background: Red
myprog.dialogwnd.background: Green
```

то все объекты программы будут иметь ресурс `background` равный `Red`, кроме объекта `dialogwnd`, для которого этот параметр есть `Green`.

1.4.2. Доступ к ресурсам программ

Пусть ресурсный файл подготовлен. Как получить доступ к его данным во время работы программы? Для этого X предоставляет набор процедур, которые совокупно называются *менеджер ресурсов* (Resource Manager), и специальную программу `xrdb`, которая позволяет считать любой ресурсный файл и включить его в общую таблицу ресурсов сервера. Последняя называется базой данных ресурсов сервера, и представляет собой область памяти, ассоциированную со свойством (property) `XA_RESOURCE_MANAGER` корневого окна экрана дисплея.

Наиболее простой является процедура `XGetDefault()`. Она получает имя программы, имя ресурса и определяет значение последнего. При этом она последовательно совершает следующие шаги:

- сначала ресурс ищется в базе данных сервера (в свойстве XA_RESOURCE_MANAGER);
- если он не найден, то значение ресурса определяется по файлу «.Xdefaults», который ищется в домашней (home) директории пользователя;
- если задана переменная среды XENVIRONMENT, то ресурс ищется в файле, на который указывает эта переменная.

Если ресурс одновременно встречается в «`.Xdefaults`» и файле, определяемом `XENVIRONMENT`, то берется последнее значение.

В примере, приводимом ниже, используется `XGetDefault()`, чтобы получить строку, которую надо напечатать в окне программы. Предполагается, что имя программы – «`hello`», а строка – ресурс с именем «`helloWorld`», т.е. в файле «`.Xdefaults`» должна быть помещена, например, следующая запись:

```
. . . . . .
hello.helloWorld : Hello, World!
```

```
. . . . . .
```

Фрагмент программы, выполняющий чтение из файла ресурсов, будет выглядеть следующим образом:

```
  . . . . . .
Display    *prDisplay;
GC         prGC;
Window         nWnd;
char       *psString;
. . . . . .
/* Устанавливаем связь с сервером, получаем
номер экрана. . .*/
. . . . . .
/* Выбираем события, обрабатываемые программой
*/
XSelectInput (prDisplay, nWnd, ExposureMask |
KeyPressMask);

/* Получаем рисуемую строку */
psString = XGetDefault (prDisplay, "hello",
"helloWorld");
. . . . . .
XDrawString ( prDisplay, nWnd, prGC, 10, 50,
psString,
    strlen (psString) );
. . . . . .
```

Обратите внимание на то, что после изменения файла «`.Xdefaults`» он должен быть обработан программой `xrdb` для того, чтобы X сервер включил в свою таблицу обновленные ресурсы.

Функция `XGetDefault()` проста в обращении, но не достаточно гибка. Так, например, с ее помощью нельзя прочитать содержимое произвольного файла ресурсов. Рассмотрим другие более развитые возможности.

Вызов `XrmInitialize()` инициализирует менеджер ресурсов. Обращение к этой функции предшествует вызовам остальных процедур.

```
XrmParseCommand (XrmDatabase *prDB,
```

```
XrmOptionRec *prOptRec,
    int nOptRecNum, char *psProgName, int
argc, char **argv;
```
сканирует строку, с помощью которой вызвана программа, и «достает» из нее ресурсы и их значения, при этом создается специальная структура данных – база данных ресурсов. Ресурсы и их значения помещаются в нее. Указатель на базу данных передается программе через переменную `prDB`. Параметр `psProgName` содержит имя программы, `argc` – число опций в командной строке, `argv` – сами опции. Аргумент `prOptRec` определяет, как разбирать командную строку. `nOptRecNum` задает число элементов массива `prOptRec`.

В примере, приводимом ниже, определяется, что в командной строке опция «`-bg`» задает цвет фона; «`-fg`» – цвет переднего плана, а опция «`-xrm`» позволяет задать в командной строке любой ресурс программы.

```
. . . . . .
XrmOptionDescRec rOptRec[ ] = {
   { "-bg", "*background", XrmoptionSepArg,
"Red"  },
   { "-fg", "*foreground", XrmoptionSepArg,
"White" },
   { "-xrm", NULL,     XrmoptionResArg, NULL
},
};
XrmDatabase rDB;
. . . . . .
void main (int argc, char **argv)
{
  . . . . . .
  XrmInitialize( );
  XrmParseCommand (&rDB, rOptRec,
             sizeof (rOptRec) / sizeof
(rOptRec[0]),
             argv[0], argc, argv);
  . . . . . .
```

```
}
```

Процедура `XrmGetFileDataBase()` позволяет считать указанный ресурсный файл и создать по нему в памяти базу данных ресурсов.

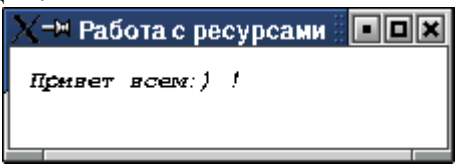
```
XrmGetResources (XrmDatabase *prDB, char
*psResName,
     char *psResClass, char *psResType,
XrmValue *psResVal);
```

Считывает ресурс с именем `psResName` и классом `psResClass` из базы данных `*prDB`. После возврата `psResType` есть указатель на строку, указывающую тип ресурса. На само значение ресурса указывает `psResVal`.

Функция `XrmPutResource()` сохраняет ресурс в базе данных. `XrmPutFileDatabase()` записывает базу данных ресурсов в файл.

1.4.3. Лабораторная работа №4 «Программы и их ресурсы»

1. Составьте программу, считывающую из файла ресурсов маску шрифта, строку, координаты *x*, *y* и отображающую окно с текстом согласно полученной информации.



1.5. Межклиентское взаимодействие

1.5.1. Механизм свойств

Как мы уже упоминали ранее, свойство есть набор данных, ассоциированных с окном. Они хранятся в специальных таблицах в памяти компьютера, на котором работает сервер. Каждое свойство имеет имя. Разные окна могут иметь свойства с одинаковыми именами.

Поскольку передача имен – строк произвольной длины – от клиента к серверу может увеличить нагрузку на сеть, X идентифицирует свойства с помощью целых чисел – *атомов*. Процедура `XInternAtom()` включает свойство с указанным именем в таблицу сервера и возвращает соответствующий атом.

Данные свойства рассматриваются сервером как массив единиц длиной 8, 16 или 32 бита. Их конкретная интерпретация осуществляется программами-клиентами.

Каждое свойство имеет тип, который, в свою очередь, также задается тем или иным свойством. Например, свойство, соответствующее атому `XA_STRING`, задает тип «строка».

Для работы со свойствами кроме `XInternAtom()` используются следующие процедуры: `XChangeProperty()` – меняет данные свойства: `XGetWindowProperty()` – позволяет получить данные свойства.

Особую роль играют свойства, данные которых содержат строки текста. Они так и называются текстовыми и имеют тип «`TEXT`». Таковыми являются, например, имена (заголовки) окно, имена пиктограмм и т.д. Данные текстового свойства описываются структурой `XTextProperty`. Процедура `XStringListToTextProperty()` переводит список строк в набор данных типа `XTextProperty`:

```
/* Эта переменная будет хранить созданное
свойство. */
XTextProperty window_title_property;

/* Строка, преобразуемая в свойство. */
char* window_title = "hello, world";

/* перевод строки в свойство X. */
int                    rc                    =
XStringListToTextProperty(&window_title,
```

```
                    1,
                    &window_title_property);
/* проверка успешности преобразования. */
if (rc == 0) {
  fprintf(stderr, "XStringListToTextProperty –
нет памяти\n");
  exit(1);
}
```

XTextPropertyToString() выполняет обратное преобразование.

1.5.2. Общение с менеджером окон

Менеджер окон – это специальный клиент, в задачи которого входит интерактивное перемещение окон по экрану, изменение их размеров, минимизация (превращение в пиктограмму) и многое другое. Чтобы облегчить менеджеру его нелегкую жизнь, программам рекомендуется при инициализации сообщить о себе определенную информацию. Передается она через предопределенные свойства, которые известны менеджеру и могут быть им прочитаны. Некоторые из свойств (так называемые стандартные) задавать обязательно. Все остальное определяется по усмотрению программы. Наиболее простой способ задать стандартные свойства – обратиться к процедурам XSetStandardProperties() или XSetWMProperties().

Ниже перечисляются свойства, создаваемые для менеджера окон программами, а также процедуры для работы с ними.

Имя (заголовок) окна. Идентифицируется атомом XA_WM_NAME и имеет тип «TEXT». Данные свойства – структура XTextProperty. Для задания свойства используется процедура XStoreName() (XSetWMName()). Получить его можно с помощью XFetchName() (XGetWMName()).

Имя пиктограммы. Идентифицируется атомом

XA_WM_ICONNAME и имеет тип «TEXT». Данные свойства – структура XTextProperty. Для задания свойства используется процедура XSetIconName() (XSetWMIconName()). Получить его можно с помощью XGetIconName() (XGetWMIconName()).

Рекомендации (hints) о геометрии окна. Идентифицируется атомом XA_WM_NORMAL_HINTS и имеет тип XA_WM_SIZE_HINTS. Данные свойства – структура типа XSizeHints. Для задания свойства используется процедура XSetNormalHints().

В ряде случаев стоит сообщить оконному менеджеру о том, какой размер окна мы хотим получить, и в каких пределах будут изменяться его размеры. Например, для терминальной программы (такой, как xterm), хотелось бы, чтобы окно всегда содержало полное количество строк и столбцов. В других случаях нежелательно давать возможность менять размер окна (например, в диалоговых окнах). Эти пожелания можно передать оконному менеджеру, хотя ничто не помешает ему их проигнорировать. Для этого необходимо создать структуру данных, заполнить ее необходимыми данными и затем использовать функцию XSetWMNormalHints():

```
/* указатель на структуру рекомендаций о
размерах. */
XSizeHints*           win_size_hints          =
XAllocSizeHints();
if (!win_size_hints) {
  fprintf(stderr,   "XAllocSizeHints   -   нет
памяти\n");
  exit(1);
}

/* Инициализация структуры */
/* Вначале укажем, что передаются пожелания о
размерах: */
/* установим минимальный и начальный размеры.
```

```
*/
win_size_hints->flags = PSize | PMinSize;
/*   Затем   указываем   требуемые   границы.
*/
/* в нашем случае – создаем окно минимальным
размером 300x200 пи */
/* и устанавливаем начальный размер в 400x250.
*/
win_size_hints->min_width = 300;
win_size_hints->min_height = 200;
win_size_hints->base_width = 400;
win_size_hints->base_height = 250;

/* Передаем пожелания о размерах оконному
менеджеру. */
XSetWMNormalHints(display,                win,
win_size_hints);

/* В конце необходимо освободить память из-под
структуры. */
XFree(win_size_hints);
```

Дополнительные параметры окна: способ работы с клавиатурой, вид и положение пиктограммы. Идентифицируется атомом `XA_WM_HINTS` и имеет тип `XA_WM_HINTS`. Данные свойства – структура типа `XWMHints`. Для задания свойства используется процедура `XSetWMHints()`. Структура типа `XWMHints`, передаваемая функции `XSetWMHints()`, должна быть подготовлена с помощью `XAllocWMHints()`:

```
XWMHints* win_hints = XAllocWMHints();
if (!win_hints) {
  fprintf(stderr,    "XAllocWMHints    –    нет
памяти\n");
  exit(1);
}

/*  установим  пожелания  о  состоянии  окна,
позиции его иконки */
```

```
/* и ее виде                              */
win_hints->flags      =      StateHint      |
IconPositionHint | IconPixmapHint;

/* Определим битовое изображение в формате X -
*/
/*  оно  может  быть  создано  программой  xpaint
*/
#define icon_bitmap_width 20
#define icon_bitmap_height 20
static char icon_bitmap_bits[] = {
 0x60,  0x00,  0x01,  0xb0,  0x00,  0x07,  0x0c,
0x03, 0x00, 0x04, 0x04, 0x00,
 0xc2,  0x18,  0x00,  0x03,  0x30,  0x00,  0x01,
0x60, 0x00, 0xf1, 0xdf, 0x00,
 0xc1,  0xf0,  0x01,  0x82,  0x01,  0x00,  0x02,
0x03, 0x00, 0x02, 0x0c, 0x00,
 0x02,  0x38,  0x00,  0x04,  0x60,  0x00,  0x04,
0xe0, 0x00, 0x04, 0x38, 0x00,
 0x84,  0x06,  0x00,  0x14,  0x14,  0x00,  0x0c,
0x34, 0x00, 0x00, 0x00, 0x00
};

/* Загрузим заданное битовое изображение */
/* и создадим из него пиксельную карту X. */
Pixmap               icon_pixmap              =
XCreateBitmapFromData(display,
                      win,
                      icon_bitmap_bits,
                      icon_bitmap_width,
                      icon_bitmap_height);
if (!icon_pixmap) {
 fprintf(stderr,        "XCreateBitmapFromData:
ошибка создания пиксмапа\n");
 exit(1);
}

/*  Затем  детализируем  желаемые  изменения.
*/
```

```
/* в нашем случае – сворачиваем окно,
определяем его иконку    */
/* и устанавливаем позицию иконки в левом
верхнем углу экрана.   */
win_hints->initial_state = IconicState;
win_hints->icon_pixmap = icon_pixmap;
win_hints->icon_x = 0;
win_hints->icon_y = 0;

/* Передаем пожелания оконному менеджеру. */
XSetWMHints(display, win, win_hints);

/* В конце необходимо освободить память из-под
структуры. */
XFree(win_hints);
```

Получить данные свойства можно с помощью `XGetWMHints()`.

Атрибут, характеризующий «временное» окно. Идентифицируется атомом `XA_WM_TRANSIENT_FOR` и имеет тип `XA_STRING`. Свойство задается для окон, появляющихся на экране для выполнения вспомогательных функций (диалоги, меню). Такие объекты рассматриваются менеджером по особому. Например, он может не добавлять к окну заголовок и рамку. Данные свойства – идентификатор окна родительского по отношению к данному. Задается свойство с помощью процедуры `XSetTransientForHint()`.

Имена программы и ее класса, идентифицируется атомом `XA_WM_CLASS` и имеет тип `XA_STRING`. Данные свойства – структура типа `XClassHints`. Задается свойство с помощью процедуры `XSetClassHint()` и может быть получено с помощью `XGetClassHint()`.

Если окно (окна) программы имеют собственную цветовую палитру, то приложение должно соответствующим образом задать для него атрибут `colormap`. Программа заносит идентификатор окна

(идентификаторы окон) в список, ассоциированный со свойством, имя которого WM_COLORMAP_WINDOWS. Делается это процедурой XSetWMColormapWindows(). Получить список, уже находящийся в свойстве, можно, обратившись к XGetWMColormapWindows().

Когда окно открыто, пользователь посредством менеджера совершает над ним разные действия. Программе может быть желательно перехватывать некоторые из них. Так, например, если окно представляет собой редактор текста, и пользователь пытается его закрыть, то разумно спросить у сидящего за компьютером человека, а не желает ли он предварительно сохранить результаты редакции. Начиная с X11R4 системой предусматривается свойство с именем WM_PROTOCOLS. Оно содержит список атомов, и каждый из них идентифицирует свойство, связанное с действиями, о которых надо оповещать программу. Эти свойства следующие:

WM_TAKE_FOCUS – задается, если программа хочет передавать фокус ввода между своими окнами самостоятельно; в этом случае менеджер не будет управлять фокусом, ввода, а пошлет приложению событие ClientMessage, у которого поле message_type равно атому, соответствующему свойству WM_PROTOCOLS, а поле data.l[0] равно атому, соответствующему свойству WM_TAKE_FOCUS; в ответ на это событие программа должна сама обратиться к XSetInputFocus() для задания окна, имеющего фокус ввода;

WM_SAVE_YOURSELF – задается, если программа хочет перехватить момент своего завершения; менеджер окон посылает приложению событие ClientMessage, у которого поле message_type равно атому, соответствующему свойству WM_PROTOCOLS, а поле data.l[0] равно атому, соответствующему свойству WM_SAVE_YOURSELF; в ответ программа может сохранить

свое текущее состояние;

WM_DELETE_WINDOW – задается, если программа хочет перехватить моменты, когда менеджер окон закрывает принадлежащие ей окна; менеджер окон посылает приложению событие ClientMessage, у которого поле message_type равно атому, соответствующему свойству WM_PROTOCOLS, а поле data.l[0] равно атому, соответствующему свойству WM_DELETE_WINDOW; далее программа сама решает, оставить окно на экране или удалить его с помощью XDestroyWindow().

Свойство WM_PROTOCOLS задается процедурой XSetWMProtocols() и может быть получено с помощью XGetWMProtocols().

Приведем фрагмент программы, задающей свойство WM_PROTOCOLS и производящей соответствующую обработку событий.

```
. . . . . .
Display          *prDisplay;
int         nScreenNum;
GC          prGC;
XEvent           rEvent;
Window           nWnd;
Atom        pnProtocol[2];
Atom        nWMProtocols;

/*
 *Устанавливаем связь с сервером, получаем
номер экрана,
 *создаем окно, выбираем события,
обрабатываемые программой
 */
. . . . . .

/* Задаем свойство WM_PROTOCOLS */

pnProtocol [0] = XInternAtom (prDisplay,
"WM_TAKE_FOCUS", True);
```

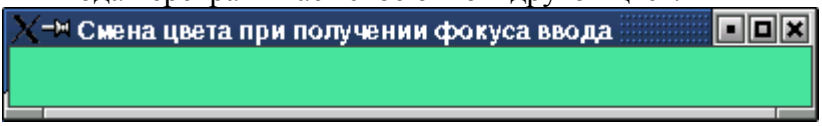
```
pnProtocol [1] = XInternAtom (prDisplay,
"WM_SAVE_YOURSELF", True);

nWMProtocols = XInternAtom (prDisplay,
"WM_PROTOCOLS", True);

XSetWMProtocols (prDisplay, nWnd, pnProtocol,
2);

/* Показываем окно */

XMapWindow (prDisplay, nWnd);

/* Цикл получения и обработки событий */

while (1) {
   XNextEvent (prDisplay, &rEvent);

switch (rEvent.type) {
  . . . . . .
  case ClientMessage :
  if (rEvent.xclient.message_type ==
nWMProtocols)
  {
    if (rEvent.xclient.data.l[0] ==
pnProtocol[0])
     puts ("Receiving the input focus.\n");
    else
     if (rEvent.xclient.data.l[0] ==
pnProtocol[1])
     {
      XCloseDisplay (prDisplay);
      exit (0);
     }
   }
   break;
   . . . . . .
 }
```

```
}
. . . . . .
```

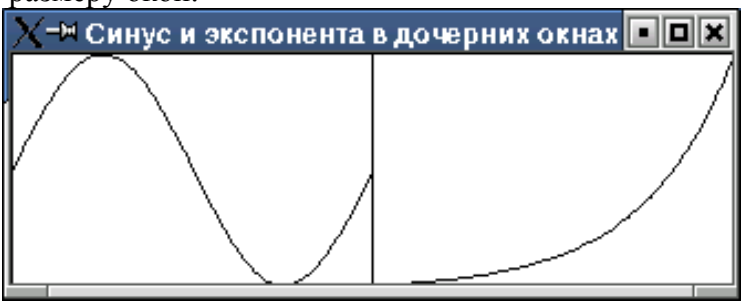
Заказывается реакция на два события: получение фокуса ввода (`WM_TAKE_FOCUS`) и завершение программы (`WM_SAVE_YOURSELF`). Когда сервер посылает событие первого типа, задача печатает соответствующее сообщение на устройства вывода. При приходе события второго типа, программа закрывает связь с сервером и завершается.

1.5.3. Лабораторная работа №5 «Межклиентское взаимодействие»

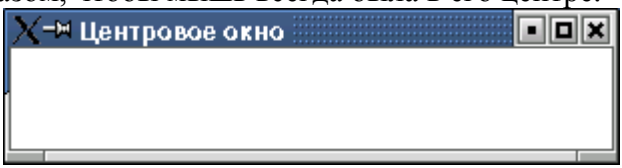
1. Составьте программу, которая при получении фокуса ввода перекрашивает свое окно в другой цвет.



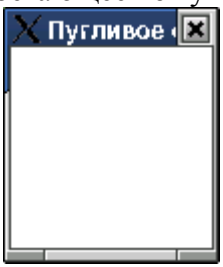
2. Составьте программу, порождающую два расположенных рядом дочерних окна, в которых отображаются графики функций sin(x) на отрезке [0; 2π] и exp(x) на отрезке [–2; 2]. Графики масштабировать по размеру окон.



3. Создайте окно, изменяющее свои размеры таким образом, чтобы мышь всегда была в его центре.



4. Создайте окно, «убегающее» от указателя мыши.



2. Программирование с использованием библиотеки X Toolkit Intrinsincs (Xt)

Чтобы облегчить программирование в системе X Window, было создано несколько пакетов. Стандартом де-факто в настоящее время стала библиотека *X Toolkit Intrinsincs (Xt)*, которая входит в комплект стандартной поставки системы. Xt упрощает инициализацию программ и создание окон. Кроме того, библиотека содержит средства для создания объектов (управляющих элементов), используемых программами при общении с пользователями. В терминах Xt управляющий элемент называется виджет (widget).

В настоящий момент на основе пакета реализованы различные наборы (множества) управляющих элементов (объектов), таких как Athena, OpenMotif, Open Look и многие другие. Эти наборы (вместе с самой Xt) применяются в качестве удобного инструмента для создания интерфейсов. Они берут на себя ту рутинную работу, которую необходимо выполнить программисту при написании собственного приложения с использованием только процедур X Window.

Функции Xt находятся в файле `/usr/X11R6/lib/libXt.a` (`libXt.so`).

2.1. Основы Xt

2.1.1. Что такое объекты Xt

Как мы уже упоминали ранее, пакет представляет собой базу для создания управляющих элементов (виджетов). В смысле Xt виджет – это просто структура данных, поля которой включают идентификатор самого элемента, идентификатор его окна, если таковое имеется, и многое другое. Атрибуты такого объекта называются *ресурсами*. Ресурсами виджета могут быть, например, цвет фона его

окна, шрифт выводимого текста, цвет границы окна и т.д.

Каждый объект принадлежит к одному из предопределенных *классов* (widget class). Класс можно рассматривать как множество экземпляров (объектов), имеющих одинаковые характеристики. Классы Xt образуют иерархию.

Во время работы программа создает сами *объекты* (экземпляры классов-виджетов). Они образуют совокупности, каждая из которых также представляет собой некоторую *иерархию*. Каждая такая иерархия называется *деревом объектов* (widget tree). Корнем дерева обязательно является виджет, принадлежащий к одному из подклассов специального класса – `Shell` (shell-объект или shell-виджет). Если среди двух виджетов A и B дерева объектов первый ближе к корню, чем второй, то A является родительским объектом (родителем) для B, а `B` есть подобъект (или дочерний объект – потомок) для A. Таким образом, shell-объект является родительским виджетом для всех остальных виджетов данного дерева объектов. Именно он осуществляет взаимодействие программы и менеджера окон.

Описанная иерархия виджетов соответствует взаимосвязи их окон, что является свойством X Window. Кроме этого, на объекты накладывается и другая иерархическая структура. Дело в том, что во время функционирования одни объекты могут управлять другими. Например, если объект A имеет два подобъекта B и C, то при изменении размеров A последний может автоматически перестроить B и C. Для того, чтобы это могло осуществиться, между виджетами устанавливается связь «по управлению». Каждый объект может иметь один или несколько «управляемых» (managed) подобъектов.

Теперь рассмотрим, как программа, использующая Xt, взаимодействует с виджетами и X Window. Предусмотрены три механизма.

Первый из них – *процедуры обратного вызова* (callback-процедуры или просто callback). Для любого класса определена совокупность действий, на которые должны реагировать принадлежащие ему объекты. Так, для любого класса предусмотрена реакция на уничтожение виджета. Когда действие производится, происходит вызов либо стандартной функции Xt, либо одной или нескольких процедур, предоставляемых программой. Такие функции и называются callback-процедурами.

Второй способ взаимодействия – *action-процедуры*. Программа может заказать реакцию на то или иное событие (или группу событий), приходящее от X. Если событие происходит, Xt осуществляет поиск и вызов соответствующей функции.

Третий механизм – *обработчики событий* (event handler). Этот способ аналогичен предыдущему, только более быстр в ущерб гибкости. Он позволяет реагировать только на единичные события, но не на их последовательности.

2.1.2. Инициализация программы. Контекст программы

Программы, работающие в X, должны выполнить ряд стандартных действий, а именно: установить связь с сервером, задать необходимые свойства для менеджера окон, и выполнить массу других шагов. Если используется Xt, то все это делается одной процедурой – `XtInitialize()`. Она инициализирует сам пакет, менеджер ресурсов X Window, и выполняет другие необходимые операции. Для каждого приложения обращение к `XtInitialize()` или аналогичной процедуре (см. ниже) должно предшествовать всем другим вызовам процедур Xt. После завершения, `XtInitialize()` возвращает идентификатор sheel-виджета, который может использоваться как корень дерева объектов программы. `XtInitialize()` имеет следующий прототип:

```
Widget XtInitialize(char *psShellName,
          char *psApplicationClass,
          XrmOptionDescRec          pOptions[],
Cardinal nNumOpt,
          Cardinal *argc, char *argv[]);
```

Первый аргумент процедуры – имя создаваемого функцией shell-объекта. Эта же строка задает и имя программы. Второй аргумент – класс приложения. Как правило, имя класса программы совпадает с именем самой программы, только начинается с заглавной буквы. Менеджер ресурсов использует имя и класс программы для поиска ее параметров в базе данных ресурсов.

Каждая программа при запуске может задать в командной строке некоторые дополнительные опции. В Xt определены стандартные параметры, которые могут быть переданы при запуске программы. Кроме этого, в командной строке могут быть и нестандартные, свойственные только данному приложению опции. `XtInitialize()` сканирует строку и размещает соответствующие значения ресурсов в базе данных. Если приложение использует нестандартные опции, то следует создать массив структур типа `XrmOptionDescRec` и передать указатель на него в качестве третьего параметра процедуры `XtInitialize()`. При этом четвертый аргумент определяет число элементов массива. Если нестандартные параметры не используются, то в качестве третьего и четвертого параметров следует задавать `NULL` и 0 соответственно. Пятый и шестой аргументы в рассматриваемой процедуре используются для передачи данных в программу из командной строки.

Обращение к `XtInitialize()` эквивалентно следующей последовательности вызовов:

```
XtToolkitInitialize(. . . . .);  /*
Инициализация Xt */
XtOpenDisplay(. . . . .);        /*  Связь  с
сервером */
```

```
XtAppCreateShell(. . . . .);     /*    Создание
shell-виджета */
```

При инициализации создается также контекст программы – структура, которая хранит всю необходимую информацию о приложении. Контекст помогает программе быть менее зависимой от различных модификаций в операционной системе. `XtInitialize()` автоматически создает некоторый контекст (default context). Но тем не менее рекомендуется, чтобы каждый экземпляр программы сам создавал свой личный контекст. Для этого инициализация задачи должна делаться функцией XtAppInitialize().

Обращение к XtAppInitialize() эквивалентно вызову процедур:

```
XtToolkitInitialize();          /*
Инициализация Xt */
XtOpenDisplay();                /*    Связь    с
сервером */
XtAppCreateShell();             /*      Создание
shell-виджета */
XtCreateApplicationContext();   /*      Создание
контекста */
```

2.1.3. Первый пример

Приведем в качестве примера простейшую программу, открывающую окно и рисующую в нем строку «Hello, world!».

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Core.h>

void    DrawHelloString    (Widget    prWidget,
XtPointer pData,
     XEvent *prEvent, Boolean *pbContinue)
{
  Display  *prDisplay = XtDisplay (prWidget);
  Window   nWindow = XtWindow (prWidget);
```

```
  GC       prGC;

  if (prEvent->type == Expose)
  {
    prGC = XCreateGC (prDisplay, nWindow, 0,
NULL);
    XDrawString(prDisplay, nWindow, prGC, 10,
50,
          "Hello,    world!",    strlen("Hello,
world!"));
    XFreeGC(prDisplay, prGC);
  }
}


int main (int argc, char **argv)
{
  Arg      args[2];
  Widget   toplevel, prCoreWidget;

  toplevel = XtInitialize (argv[0], "Simple",
NULL, 0,
                &argc, argv);
  prCoreWidget   =   XtCreateWidget   ("Core",
widgetClass,
                toplevel, NULL, 0);

  XtSetArg (args[0], XtNwidth, 100);
  XtSetArg (args[1], XtNheight, 100);
  XtSetValues (prCoreWidget, args, 2);

  XtManageChild (prCoreWidget);
  XtAddEventHandler              (prCoreWidget,
ExposureMask, False,
                DrawHelloString, NULL);
  XtRealizeWidget (toplevel);
  XtMainLoop ();
}
```

Для сборки программы используется команда:

```
cc -o hello hello.o -lXaw -L/usr/X11R6/lib
```

Обратите внимание на то, что мы подключается не библиотека Xt, а библиотека Xaw (Athena – *Афина*, набор виджетов, разработанный вместе с библиотекой Xt). Дело в том, что при компоновке программы с библиотекой Xt клавиатурный фокус ввода отдается тому окну, из которого запущена программа (чаще всего это окно терминала). Во избежание подобного поведения программы и используется библиотека Xaw. Ее использование приводит к подключению библиотеки Xt, а та, в свою очередь, подключает библиотеку Xlib,

Для подключения библиотеки Xt к интегрированной среде Anjuta необходимо добавить `Xaw` в закладке «Библиотеки» параметров компилятора.

Программа использует ряд функций, предоставляемых библиотекой Xt: `XtInitialize()`, `XtCreateWidget()` и др. Их прототипы, стандартные структуры данных, макросы и константы описаны в следующем основном файле-заголовке: `<X11/Intrinsic.h>`. В некоторых случаях необходимо также включить файл `<X11/StringDefs.h>`. Если программа создает и использует какой-либо объект (виджет), то она должна включить файл `<имя класса виджета>.h`. Например, в приведенной программе есть объект класса `Core`, поэтому мы используем файл `<X11/Core.h>` (заметим, что включение указанного файла не обязательно, т.к. файл `<X11/Intrinsic.h>` его включает).

Работа начинается с инициализации программы процедурой `XtInitialize()`. Если она завершается успешно, то приложение возвращается указатель на созданный shell-объект класса `ApplicationShell`.

На втором шаге создаются управляющие элементы программы. В нашем примере процедура `XtCreateWidget()` создает виджет класса `Core`, который и является единственным потомком для shell-виджета,

созданного процедурой `XtInitialize()`..

Вызов процедуры `XtManageChild()` делает объект `prCoreWidget` «управляемым» со стороны родителя.

После этого мы регистрируем для этого виджета обработчик события `Expose`. Это процедура `DrawHelloString()`. Она рисует в окне строку «Hello, world!». Регистрация осуществляется процедурой `XtAddEventHandler()`.

После того, как объекты программы подготовлены к работе, соответствующие им окна показываются на экране:
```
XtRealizeWidget (toplevel);
```
Далее начинается цикл получения и отправки событий:
```
XtMainLoop();
```
Когда программа использует только процедуры библиотеки Xlib, то она должна рассматривать каждое происходящее событие и соответствующий образом реагировать на него. Если окон у задачи несколько, то прежде чем производить какие-то действия, необходимо определить, в каком из них произошло событие. Все это достаточно утомительно. Xt берет работу на себя. `XtMainLoop()` получает очередное событие и определяет окно, для которого оно предназначено. По окну находится соответствующий виджет. Для последнего определяются обработчик сообщений, action-процедура или callback, зарегистрированные как реакция на произошедшее событие. Если таковые есть, то они вызываются. Указанный выше механизм называется рассылкой событий.

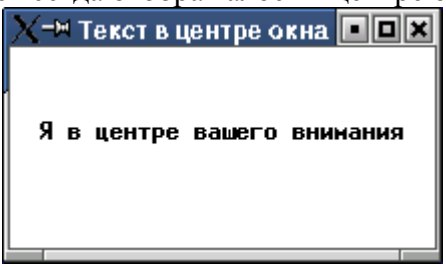
`XtMainLoop()` автоматически завершает программу по требованию менеджера окон.

Наш пример может работать во всех версиях Xt. Но начиная с X11R4 вместо `XtInitialize()` и `XtMainLoop()` следует использовать `XtAppInitialize()` и `XtAppMainLoop()`.

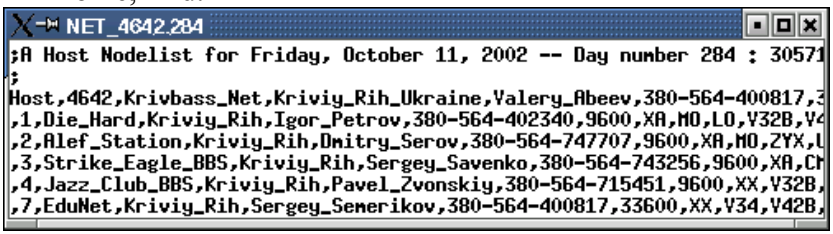
2.1.4. Лабораторная работа №6 «Основные понятия Xt»

1. Выполните компиляцию примера из п. 2.1 учебного пособия и запустите полученную программу на выполнение.

2. Используя StructureNotifyMask, модифицируйте предыдущую программу таким образом, чтобы сообщение всегда отображалось в центре окна.



3. Создайте программу, отображающую в окне содержимое текстового файла, имя которого задается в командной строке. Для скроллинга текста используйте клавиши Up, Down, Left, Right, PageUp, PageDown, Home, End.



4. Напишите программу, которая инициализирует Xt и запускает цикл обработки сообщений без создания виджета.

2.2. Объекты Xt и взаимодействие с ними

Как указывалось выше, Xt предоставляет набор средств для создания объектов, которые используются программами для общения с пользователем, а в общем случае и с остальным внешним миром. В данном разделе рассматриваются классы виджетов, процедуры для работы с ними, а также способы общения объектов Xt и остальных частей программы.

2.2.1. Классы объектов

Каждый создаваемый программой виджет является представителем того или иного класса. Xt и основывающиеся на нем пакеты, такие как OpenMotif, Open Look и др., имеют большое количество таких классов (заметим, что создание своих собственных виджетов, не предусмотренных в используемых библиотеках, требует и определения соответствующего класса, что, вообще говоря, является достаточно трудоемкой операцией).

Каждый класс имеет ряд фиксированных характеристик, являющихся общими для всех его экземпляров (например, список callback-процедур). Значения этих характеристик у самих объектов могут различаться.

Все классы в Xt составляют определенную иерархию. Если класс A ближе к вершине иерархии, чем класс B, то A называется *базовым* для B, а B называется *производным* классом (или подклассом) для A. Например, класс `Core` является базовым по отношению к `Composite`, а `Composite` есть подкласс `Core`.

Подклассы наследуют характеристики всех своих базовых классов. Это означает, что экземпляр класса имеет характеристики не только своего класса, но и атрибуты всех базовых классов. Например, основным классом в Xt является класс `Object`, характеристики которого непосредственно наследует класс `RectObj`. В класс `Shell`

переносятся атрибуты Core и Composite и т.д.

В программах каждый класс идентифицируется переменной, которая указывает на соответствующую структуру данных. Эту переменную называют *указателем на класс*. Данные этой структуры заполняются при инициализации Xt.

Указатель на класс используется в качестве второго аргумента в процедурах создания виджета (например, XCreateWidget() и др.). Для каждого класса его указатель определен в файле <имя класса>.h. Так, например, для Composite это файл <Composite.h>, а определение переменной выглядит следующим образом:

```
WidgetClass compositeWidgetClass;
```

Рассмотрим подробнее основные классы виджетов.

Object. Абстрактный класс (т.е. класс, не порождающий собственного объекта), который используется в качестве корня дерева всех объектов.

RectObj. Абстрактный класс, который используется для определения некоторых общих характеристик, необходимых для функционирования различных типов объектов (например, для объектов, не имеющих окна).

Core. Корень дерева классов виджетов, имеющих окна. Этот класс определяет характеристики, общие для всех объектов, например такие, как размер окна виджета и его положение на экране.

Composite. Подкласс класса Core. Виджеты, относящиеся к данному классу, могут быть родительскими по отношению к другим объектам. Экземпляры класса Composite определяют следующие особенности поведения своих подобъектов:

- задает местоположение дочерних виджетов согласно тем или иным ограничениям;
- при уничтожении освобождает память, используемую подобъектами (при уничтожении самого виджета класса Composite сначала будут

уничтожены все его потомки);

- управляет появлением на экране окон своих дочерних виджетов;
- управляет передачей фокуса ввода между подобъектами.

`Constraint`. Это подкласс класса `Composite`. Он представляет собой дальнейшее расширение базового класса. Его экземпляры имеют дополнительные возможности для управления размером и местоположением своих потомков. Например, подобъекты могут размещаться в специальном порядке – в ряд, в столбец и т.д.

`Shell`. Это специальный подкласс класса `Composite`, предназначенный для взаимодействия с менеджером окон. Виджет из этого класса может иметь только одного потомка.

`OverrideShell`. Это важный подкласс класса `Shell`. Для окна виджета данного класса атрибут `override_redirect` устанавливается в `True`. Это означает, что менеджер окон его не контролирует. Как правило, окна объектов данного класса не имеют аксессуаров, добавляемых менеджером (заголовка, рамки), и используются в основном для создания выпадающих (pop up) меню.

`WMShell`. Это специальный подкласс класса `Shell`, который содержит дополнительные поля, необходимые для взаимодействия с менеджером окон.

`VendorShell`. Этот класс предназначен для того, чтобы дать возможность взаимодействия со специальными менеджерами окон.

`TopLevelShell`. Виджеты данного класса, как правило, используются как shell-объекты дерева объектов.

`TransientShell`. Этот класс отличается от предыдущего только особенностями взаимодействия с менеджером окон. Окна виджета данного класса не могут быть минимизированы (превращены в пиктограмму). Но

если в пиктограмму превращается родитель объекта класса TransientShell, то окно виджета убирается с экрана. Класс TransientShell используется для создания диалогов.

ApplicationShell. Это подкласс класса TopLevelShell. Программа может иметь, как правило, только один экземпляр, принадлежащий классу ApplicationShell.

2.2.2. Атрибуты (ресурсы) объектов

Каждый из классов виджетов, а, следовательно, и создаваемые на его основе объекты, имеют достаточно большое количество параметров. В терминах Xt они называются *ресурсами*.

Ресурсы объектов однозначно идентифицируются символическими константами, начинающимися с букв «XtN». Например, класс Core имеет параметры XtNx, XtNy, XtNwidth, XtNheight, определяющие начальное положение окна на экране и его размеры. Класс Shell имеет атрибут XtNsaveUnder, говорящий о том, надо или нет сохранять содержимое экрана под окном при показе последнего.

2.2.3. Управление объектами

Для создания любого объекта, кроме shell-виджета, можно использовать процедуру

```
Widget XtCreateWidget (String psName,
WidgetClass pClass,
    Widget prParent, ArgList arArgs, Cardinal
nNumArgs);
```

Здесь первый аргумент – это произвольная строка, содержащая имя создаваемого виджета. Она используется для получения значений ресурсов объекта из базы данных ресурсов, а также для других целей. Второй аргумент задает класс виджета. Третий аргумент – это родительский

объект, который должен быть уже создан. В качестве последнего может быть указан shell-виджет или любой объект, который может иметь подобъекты (принадлежит классу `Composite` или его подклассам). Четвертый и пятый аргументы `XtCreateWidget()` определяют значения дополнительных ресурсов, задаваемых при создании виджета. Тип `ArgList` определен в файле `<X11/Intrinsic.h>` следующим образом:

```
typedef struct {
     String          name;
     XtArgVal    value;
} Arg, *ArgList;
```

Здесь `name` представляет предопределенную строковую константу, задающую имя соответствующего ресурса (например `XtNwidth`). `value` – это значение атрибута. Это поле имеет специальный тип `XtArgVal`, определение которого является системно-зависимым. Как правило, размер данного поля достаточен для того, чтобы поместить туда целое число или указатель. Но его не достаточно, например, для задания чисел типа `double`. Общее правило таково: если размер значения ресурса меньше `sizeof(XtArgVal)`, то в `value` содержится непосредственно само значение, в противном случае поле `value` есть адрес области памяти, содержащей значение ресурса. Если аргументы `arArgs` и `nNumArgs` в `XtCreateWidget()` не используются, то они должны быть равны `NULL` и 0 соответственно.

Если ресурс не найден в списке `arArgs`, то его значение ищется в базе данных ресурсов программы. В случае, когда его нет и там, параметр получает значение по умолчанию.

`XtCreateWidget()` возвращает идентификатор созданного объекта. В следующем примере создается виджет класса `Core`, имеющий ширину и высоту, равные 100 и 150 пикселей соответственно.

```
. . . . . .
Arg  aReserv [ ] = { { XtNwidth, (XtArgVal)
100 },
                {XtNheight, (XtArgVal) 150 },
              };
 /* Ресурсы и их значения */
Widget prShellParent;  /*  Родительский  объект
*/
Widget prCoreWidget;   /*         Идентификатор
объекта */
. . . . . .

prCoreWidget    =    XtCreateWidget    ("Core",
widgetClass,
       prShellParent,     aReserv,     XtNumber
(aReserv));
. . . . . .
```

Здесь `XtNumber()` – макрос, определяющий размер массива фиксированной длины.

Заметим также, что значение ресурса в массиве `aReserv` должно иметь тип, соответствующий данному ресурсу.

Вместо создания статического массива ресурсов можно заполнять его динамически, используя макрос `XtSetArg()`. Например, предыдущий фрагмент можно было бы переписать следующим образом:

```
. . . . . .
Arg  aReserv [5];    /*    Используется    для
задания ресурсов */

int  i = 0;
Widget prShellParent;  /*  Родительский  объект
*/
Widget prCoreWidget;   /*         Идентификатор
объекта */
. . . . . .

XtSetArg (aReserv [i], XtNwidth, 100);
```

```
i++;
XtSetArg (aReserv [i], XtNheight, 150);
i++;
prCoreWidget    =    XtCreateWidget    ("Core",
widgetClass,
       prShellParent, aReserv, i);
. . . . . .
```

В X11R4 и выше задавать ресурсы при создании виджета можно более удобно. Для этого надо использовать процедуру

```
XtVaCreateWidget (String psName, WidgetClass
prClass,
    Widget prParent, . . .);
```

На месте многоточия при вызове процедуры должен стоять завершающийся нулем список пар: «имя ресурса, его значение». Ниже приведен пример применения `XtVaCreateWidget()`.

```
. . . . . .
Widget prShellParent;  /*  Родительский  объект
*/
Widget prCoreWidget;   /*          Идентификатор
объекта */
. . . . . .

prCoreWidget    =    XtVaCreateWidget    ("Core",
widgetClass,
  prShellParent,   XtNwidth,   100,   XtNheight,
150, NULL);
. . . . . .
```

Каждый родительский виджет (за исключением shell-виджета) управляет своими потомками, т.е. распоряжается их расположением, размером, получением фокуса ввода и т.д. Например, в OpenMotif некоторые объекты располагают свои дочерние виджеты в ряды и колонки, другие группируют потомков в блоки и т.д. Для того, чтобы дочерний виджет мог управляться своим родителем, его необходимо включить в соответствующий список. Для

этого используется процедура
```
void XtManageChild(Widget prWidget);
```
Здесь единственный аргумент идентифицирует виджет, которым будет управлять родитель.

Xt предусматривает более удобную процедуру `XtCreateManagedWidget()` (`XtVaCreateManagedWidget()`), которая создает виджет и после этого автоматически вызывает функцию `XtManageChild()`. Аргументы указанной процедуры полностью идентичны аргументам процедуры `XtCreateWidget()` или `XtVaCreateWidget()`. Заметим, однако, что иногда более удобно создать сначала группу дочерних виджетов, а затем сразу всех включить в список управляемых (managed) объектов. Для этого можно воспользоваться функцией
```
void XtManageChildren (WidgetList
prWidgetChildren,
 Cardinal nChildrenNumber);
```
Здесь первый и второй аргументы задают массив подобъектов и число элементов в этом массиве соответственно.

Процедура `XtRealizeWidget(Widget prWidget)` создает окна заданного виджета, его управляемых подобъектов и показывает их на экране. Этот процесс называется «реализацией» объекта. Заметим, что нельзя реализовать виджет, если это не сделано предварительно с его родителем.

Если объект реализован, то вызов функции `XtManageChild()` для любого его подобъекта автоматически реализует последний.

Целый ряд процедур Xt позволяют получать различную информацию о виджете. Среди таких процедур укажем следующие.

Процедура `XtIsRealized(Widget prWidget)` позволяет проверить, реализован данный объект или нет.

Процедура `XtDisplay(Widget prWidget)` возвращает указатель на X структуру данных типа `Display`, которая используется данным виджетом.

Процедура `XtScreen(Widget prWidget)` возвращает для заданного объекта указатель на X структуру данных типа `Screen`.

Процедура `XtDestroyWidget(Widget prWidget)` используется для уничтожения заданного виджета и всех его потомков: указанная процедура освобождает все ресурсы, используемые уничтожаемыми объектами.

2.2.4. Модификация и чтение ресурсов объекта

Для модификации ресурсов уже созданного виджета приложение может использовать процедуру

```
void XtSetValues (Widget prWidget, ArgList prArgs,
Cardinal nNumArgs);
```

Здесь `prWidget` – объект, ресурсы которого устанавливаются, второй и третий аргументы содержат список пар: «имя ресурса/его значение» и число таких пар соответственно. Например, следующий фрагмент кода позволяет установить ширину и высоту виджета класса `Core`:

```
. . . . . .
Arg aReserv[5];     /* Используется для
задания ресурсов */

int  i = 0;
Widget prShellParent;  /* Родительский объект
*/
Widget prCoreWidget;   /*         Идентификатор
объекта */
. . . . . .
prCoreWidget    =    XtCreateWidget    ("Core",
widgetClass,
           prShellParent, NULL, 0);
```

```
XtSetArg (aReserv[i], XtNwidth, 100);
i++;
XtSetArg (aReserv[i], XtNheight, 150);
i++;
XtSetValues (prCoreWidget, aReserv, i);
. . . . . .
```

X11R4 и выше поддерживает также функцию

```
void XtVaSetValues (Widget prWidget, . . . );
```

При ее вызове вместо многоточия должен стоять завершающийся нулем список пар: «имя ресурса, его значение». Приведенный выше пример с использованием `XtVaSetValues()` выглядит следующим образом:

```
. . . . . .
Widget prShellParent;  /*  Родительский  объект
*/
Widget prCoreWidget;   /*          Идентификатор
объекта */
. . . . . .
prCoreWidget    =    XtCreateWidget    ("Core",
widgetClass,
           prShellParent, NULL, 0);

XtVaSetValues (prCoreWidget, XtNwidth, 100,
          XtNheight, 150, NULL);
. . . . . .
```

Возможно задание следующих ресурсов:

Имя	**Класс**	**Тип возвращаемого значения**	**Значение по умолчанию**
background	Background	Pixel	XtDefaultBackground
bitmap	Pixmap	Pixmap	None
border	BorderColor	Pixel	XtDefaultForeground
borderWidth	BorderWidth	Dimension	1
cursor	Cursor	Cursor	None

Имя	Класс	Тип возвращаемого значения	Значение по умолчанию
cursorName	Cursor	String	NULL
destroyCallback	Callback	XtCallbackList	NULL
encoding	Encoding	unsigned char	XawTextEncoding8bit
font	Font	XFontStruct*	XtDefaultFont
foreground	Foreground	Pixel	XtDefaultForeground
height	Height	Dimension	text height
insensitiveBorder	Insensitive	Pixmap	Gray
internalHeight	Height	Dimension	2
internalWidth	Width	Dimension	4
justify	Justify	XtJustify	XtJustifyCenter
label	Label	String	NULL
leftBitmap	LeftBitmap	Pixmap	None
mappedWhenManaged	MappedWhenManaged	Boolean	TRUE
pointerColor	Foreground	Pixel	XtDefaultForeground
pointerColorBackground	Background	Pixel	XtDefaultBackground
resize	Resize	Boolean	TRUE
sensitive	Sensitive	Boolean	TRUE
width	Width	Dimension	text width
x	Position	Position	0
y	Position	Position	0

Xt позволяет получить текущее значение ресурса виджета, используя процедуру

```
void XtGetValues (Widget prWidget, ArgList
```

```
prArgs,
Cardinal nNumArgs);
```

Здесь `prWidget` – это объект, значения ресурсов которого будут получены. Второй аргумент содержит список пар, каждая из которых задает имя ресурса и адрес, по которому будет сохранено его значение. Третий аргумент есть число таких пар. Например, приводимый ниже фрагмент кода позволяет получить текущее значение высоты созданного объекта.

```
. . . . . .
Arg      aReserv[5];
Dimension nHeight;
Widget         prWidget;
. . . . . .
XtSetArg (aReserv[0], XtNheight, &nHeight);
XtGetValues (prWidget, aReserv, 1);
. . . . . .
```

В результате в переменную `nHeight` будет скопировано текущее значение высоты для созданного виджета. Переменная, в которую копируется значение ресурса, должна иметь соответствующий ему тип.

Начиная с X11R4 можно использовать процедуру

```
void XtVaGetValues (Widget prWidget, . . .);
```

При ее вызове вместо многоточия должен стоять завершающийся нулем список пар, задающих имена ресурсов и адреса, по которым будут сохранены их значения. Приведенный выше пример с использованием `XtVaGetValues()` выглядит следующим образом:

```
      . . . . . .
      Dimension nHeight;
      Widget         prWidget;
      . . . . . .
      XtVaGetValues     (prWidget,     XtNheight,
&nHeight, NULL);
      . . . . . .
```

2.2.5. Динамические ресурсы объектов. Процедуры обратного вызова

Описанные нами ресурсы виджетов в большинстве своем задают «статику» объекта: размер, цвет и т.д. Но дело в том, что управляющий элемент предназначен для того, чтобы дать возможность пользователю производить определенные действия (например, вводить текст, выбирать элементы из списков данных, рисовать график и т.д.). Кроме того, виджет должен соответствующим образом реагировать на запросы о перерисовке и т.п. Следовательно, управляющие элементы должны иметь возможность вызывать процедуры, осуществляющие эти действия. При этом ясно, что процедуры, поставляемые самой системой можно использовать только для выполнения некоторого набора стандартных действий. Для возможности же реализации разнообразных специфичных операций приложению необходимо предоставить механизмы, позволяющие устанавливать связь между управляющими элементами и процедурами самой программы, которые выполняют необходимые функции. В Xt предусмотрены следующие три способа (механизма) осуществления такой связи и передачи управления в процедуры пользователя:

- использование процедур обратного вызова (callback-процедур);
- использование action-процедур;
- использование обработчиков событий (event handler).

Рассмотрим каждый из них более подробно.

С каждым виджетом сопоставлен набор операций, которые над ним можно производить. Так, любой элемент может быть создан и уничтожен, «нажимаемая» кнопка (push button) может быть «нажата» с помощью мыши или клавиши пробел и т.д. Каждому из таких предопределенных действий соответствует ресурс объекта,

значение которого – список процедур библиотеки Xt и самой программы, которые автоматически вызываются, когда совершается предусмотренное действие. Эти процедуры и называют процедурами обратного вызова (callback-процедуры). Рассмотренные нами ресурсы виджетов имеют тип `XtCallbackList`.

Все виджеты имеют по крайней мере один список callback-процедур. Он соответствует операции уничтожения объекта и наследуется от класса `Core`. Имя ресурса – `XtNdestroyCallback`.

Программа может добавлять и убирать процедуры в (из) список (списка) callback-процедур. Добавить функцию в список можно с помощью процедуры

```
XtAddCallback (Widget prWidget, String psCallbackName,
     XtCallbackProc pProcedure, XtPointer pUserData);
```

Здесь `prWidget` задает объект, список которого модифицируется, `psCallbackName` есть имя списка, `pProcedure` – добавляемая функция, `pUserData` указывает на данные, передаваемые callback-процедуре при ее вызове.

Каждая процедура обратного вызова должна иметь прототип:

```
void CallbackProc (Widget prWidget, XtPointer pUserData,
      XtPointer pCallData);
```

Здесь первый аргумент задает виджет, в списке которого находится функция. Второй аргумент указывает на данные, передаваемые из программы в процедуру. Третий аргумент указывает на данные, подготовленные для callback-процедуры самой Xt или пакетом, построенным на ее основе (например, OpenMotif). Обычно это структура, содержащая дополнительную информацию о действии, приведшем к вызову функции.

Для модификации списка процедур обратного вызова

можно также использовать функцию

```
void XtAddCallbacks (Widget prWidget,
     String   psCallbackName,   XtCallbackList
pCallbacks);
```

которая позволяет задавать несколько callback-процедур одновременно для одного и того же списка.

Здесь первый аргумент задает виджет, чей список callback-процедур меняется. Его имя определяется вторым аргументом функции. Третий аргумент задает сами процедуры и передаваемые им параметры. Этот аргумента имеет специальный тип, определяемый следующим образом:

```
typedef struct {
 XtCallbackProc callback;   /*   Указатель   на
процедуру */
 caddr_t     client_data;   /*   Указатель   на
данные */
} XtCallbackRec, *XtCallbackList;
```

Здесь первое поле задает callback-процедуру, а второе поле используется для передачи ей при вызове некоторых данных. При обращении к `XtAddCallbacks()` последний элемент массива `pCallbacks` должен быть равен `NULL`.

Удалить процедуру из списка callback-процедур можно с помощью функции `XtRemoveCallback()`.

2.2.6. Использование action-процедур

Описанный в предыдущем пункте механизм использования callback-процедур достаточно прост и удобен, но имеет и существенный недостаток. В частности, указанные процедуры не могут вызываться в ответ на действие, не определенное для данного класса объектов. Например, в OpenMotif класс `XmDrawingArea` (область для рисования) поддерживает список callback-процедур, вызываемый, когда виджет получает фокус ввода. Однако, этого явно недостаточно, чтобы реализовать более сложное взаимодействие с пользователем.

Наряду с callback-процедурами, Xt Intrinsics поддерживает еще один механизм, позволяющий пользователю связывать код программы с виджетом. Указанный механизм состоит в использовании процедур – «ответов на действия» (action-процедур).

Как уже отмечалось ранее, *action-процедура* – это функция, которая вызывается в ответ на определенные события, порождаемые сервером. Они, в свою очередь, могут быть связаны с теми или иными действиями пользователя – нажатием на кнопку мыши, нажатие клавиш клавиатуры и т.д.

Процесс создания и подключения таких процедур состоит из следующих шагов, описываемых ниже.

1. Сначала создается специальная *таблица действий* (action table), связывающая символические имена действий (action) с указателями на соответствующие action-процедуры.
2. *Регистрация созданной таблицы* с помощью функций `XtAppAddActions()` или `XtAddActions()`.
3. Создание *таблицы соответствия* (*таблица трансляции* – translation table). Каждый ее элемент есть пара «последовательность событий / имя (или имена) действия». Таблица размещается в файле ресурсов программы или непосредственно в ее коде. В последнем случае перед использованием таблица должна быть переведена процедурой `XtParseTranslationTable()` во внутренне представление Xt и зарегистрирована для объекта с помощью `XtAugmentTranslations()` или `XtOverrideTranslations()`. Подробно формат таблицы соответствия описан ниже.

С учетом представленных шагов вызов action-процедуры, когда в виджете происходит какое-то событие (события), производится следующим образом:

- Xt проверяет для данного виджета таблицу

трансляции; если в ней такого события нет, то ничего не делается;

- если событие есть, то по нему определяются имена соответствующих действий;
- сканируется зарегистрированная таблица действий, и по именам определяются адреса процедур; когда адрес становится известен, вызывается соответствующая функция.

Рассмотрим подробнее, что представляют из себя таблица действий и таблица трансляции.

Первая из них – это массив структур типа `XtActionRec`, который определяется следующим образом:

```
typedef struct {
  String      string;
  XtActionProc      proc;
} XtActionRec,   *XtActionRecList;
```

Здесь первое поле есть символическое имя действия. Указатель на соответствующую процедуру задается полем `proc`. Например, таблица действий, содержащая процедуру `KeyboardPrint()`, вызываемую, когда происходит действие с именем «`KeyboardPrint`», может быть определена следующим образом:

```
static void KeyboardPrint();
static XtActionRec paAct[ ] = {
      { "KeyboardPrint", KeyboardPrint },
};
```

Сделаем ряд замечаний.

Если используется множество управляющих элементов OpenMotif, то следует помнить, что каждый класс виджета указанного множества предусматривает набор стандартных action-процедур, которые могут использоваться всеми экземплярами объектов данного класса. Функции же, регистрируемые с помощью `XtAppAddActions()` (`XtAddActions()`), могут быть использованы произвольным виджетом любого класса данной программы.

Попытка добавить action-процедуру, имеющую имя, совпадающее с именем стандартной action-функции, предусмотренной данным классом, ни к чему не приведет. Ваша процедура будет игнорироваться и, следовательно, не будет вызываться.

Имена действия и реализующей его процедуры в принципе могут не иметь ничего общего. Но для удобства принято, чтобы они совпадали.

Каждая action-процедура должна иметь следующий прототип:

```
void ActionProc (Widget prW, XEvent *prEvent,
         String       *psParams,       Cardinal
*nNumParams);
```

Здесь первый аргумент – это виджет, для которого зарегистрирована данная процедура. `prEvent` – указатель на событие, инициировавшее ее вызов. Если процедура вызывается в ответ на последовательность событий, то в функцию передается первое из них. Третий и четвертый аргументы задают соответственно список параметров, описанных в таблице трансляции, и их число.

Как уже указывалось ранее, после регистрации action-процедуры, например, с помощью `XtAppAddActions()`, ее могут использовать произвольные виджеты данной программы. У каждого объекта есть ресурс `XtNtranslations`, значение которого – таблица трансляции, представленная во внутреннем формате Xt.

В ресурсном файле или в исходном коде программы эта таблица представляется как строка, содержащая пары «последовательность событий / имя (или имена) действия», разделенные символом «`\n`». В начале строки может стоять директива «`#replace`», «`#augment`» или «`#override`». Первая говорит, что таблица должна заменить уже имеющуюся у виджета таблицу. Вторая и третья означают, что новая таблица должна быть объединена с таблицей, уже имеющейся у виджета. При этом, если указана директива

«#augment», то, если в таблицах имеются записи, соответствующие одному и тому же действию, то приоритет имеет старая запись. Если указана директива «#override», то наоборот – новая.

Каждая пара «последовательность событий / имя (или имена) действия» (трансляция) имеет следующий вид:

```
[модификатор, . . , модификатор]<событие>, . .<событие>[ (число
повторений события)] [детализация] : имя action([аргументы])
[имя action([ аргументы) . . . . ]
```

Здесь элементы, заключенные в индексные скобки [ ], являются необязательными.

В поле модификатор могут использоваться стандартные модификаторы, поддерживаемые Xt Intrinsics, например: Ctrl, Shift, Caps Lock и др. Список используемых в таблице трансляции модификаторов приводится ниже.

Модификатор	Описание
Ctrl	Клавиша Control
Shift	Клавиша Shift
Lock	Клавиша Caps Lock
Meta	Клавиша Meta
Hyper	Клавиша Hyper
Super	Клавиша Super
Alt	Клавиша Alt
Mod1, . . . , Mod5	Дополнительные клавиши-модификаторы
Button1, . . . , Button5	Кнопки мыши

В поле <событие> должно стоять имя события X Window.

Если модификаторов в трансляции нет, то указанные в записи действия и соответствующие им процедуры будут вызываться в ответ на определенные в угловых скобках

события или последовательности событий независимо от того, в каком состоянии находятся клавиши-модификаторы и кнопки мыши, например:

```
<KeyPress> : KeyboardPrint()
```

означает, что процедура, соответствующая действию с именем «KeyboardPrint», будет вызвана при нажатии любой клавиши, при этом кнопки мыши или клавиши-модификаторы могут находиться как в нажатом состоянии, так и в освобожденном.

Для конкретизации различных событий предусмотрено специальное поле «детализация». Так для событий от клавиатуры в этом поле указывается символ клавиши, определенный в файле «X11/keysymdef.h» (префикс «XK_» опускается). Например, следующая запись

```
<KeyPress>a : KeyboardPrint()
```

используется для определения события, соответствующего нажатию клавиши «a». Для событий ButtonPress, ButtonRelease, MotionNotify детализация есть номер нажатой кнопки.

Для событий EnterNotify, LeaveNotify, FocusIn, FocusOut в поле «детализация» может быть использован любой из следующих символов:

Normal – для событий с обычным режимом получения и передачи фокуса ввода;

Grab – для событий, когда устройство захвачено;

Ungrab – для событий, возникающих при отмене захвата.

Для событий ClientMessage детализация – тип сообщения (число).

Для того, чтобы облегчить задание таблицы трансляций, Xt предусматривает специальную аббревиатуру для некоторых событий.

Аббревиатура	Смысл
Ctrl	Событие KeyPress, когда нажата клавиша «Control»

Аббревиатура	Смысл
Meta	Событие KeyPress, когда нажата клавиша «Meta»
Shift	Событие KeyPress, когда нажата клавиша «Shift»
Btn1Down, ..., Btn5Down	Событие ButtonPress для 1, ..., 5-й кнопки мыши
Btn1Up, ..., Btn5Up	Событие ButtonRelease для 1, ..., 5-й кнопки мыши
BtnMotion	Событие MotionNotify
Btn1Motion, ..., Btn5Motion	Событие MotionNotify, когда нажата 1, ..., 5-я кнопка мыши

Например, запись:

```
<Btn2Down> : StartWindowing()
```

означает, что в ответ на нажатие второй кнопки мыши, вызывается действие `StartWindowing()`.

Xt поддерживает набор специальных символов, употребляемых с модификаторами для расширения возможностей использования последних.

Символ	Описание
None	Ни одна клавиша-модификатор не должна быть нажата
!	Событие должно сопровождаться только указанными модификаторами и никакими другими
:	Позволяет различать события от нажатия клавиш в зависимости от регистра
~	Модификатор, непосредственно следующий за данным символом, не должен быть в нажатом состоянии

Например, запись

```
!Alt<Key>a : KeyboardPrint()
```

означает, что соответствующая процедура будет вызвана только при нажатии комбинации <Alt+a>. Применение

символа «None» эквивалентно использованию в трансляции символа «!» без модификаторов; например, следующие записи эквивалентны:

```
None<Key>a : KeyboardPrint()
```

и

```
!<Key>a : KeyboardPrint()
```

Это означает, что действие будет реализовано, когда клавиша «a» нажата, а все модификаторы находятся в не нажатом состоянии.

Использование в трансляции символа «:» позволяет различать события, соответствующие нажатию клавиш на разных регистрах: верхнем или нижнем (т.е. при нажатой или освобожденной клавише Shift или Caps Lock). Дело в том, что, например, следующие записи эквиваленты по выполняемому действию:

```
<Key>b : KeyboardPrint()
```

и

```
<Key>B : KeyboardPrint()
```

т.е. соответствуют вызову процедуры при нажатии клавиши «b» или комбинации <Shift+b>. Использование символа «:» позволяет различить эти две ситуации, в частности, следующие записи будут уже не идентичны:

```
: <Key>b : KeyboardPrint()
```

и

```
: <Key>B : KeyboardPrint()
```

первая определяет вызов соответствующей процедуры при нажатии клавиши «b»; вторая же «трансляция» соответствует вызову функции при нажатии комбинации <Shift+b>.

Символ «~» используется для отрицания модификатора, т.е. следующий за символом модификатор не должен быть нажат. Например, запись:

```
Button2<Key> : KeyboardPrint()
```

означает, что соответствующая процедура будет вызвана, когда нажата вторая кнопка мыши и произвольная клавиша

на клавиатуре, в то же время запись

```
~Button2<Key> : KeyboardPrint()
```

говорит, что соответствующая функция будет вызвана, когда нажимается произвольная клавиша, и если при этом не нажата вторая кнопка мыши. Заметим, что использование символа «~» относится только к модификатору, непосредственно следующему за указанным символом, и не распространяется на остальные модификаторы. Например, трансляция

```
~Ctrl Alt<Key>a : KeyboardPrint()
```

указывает, что соответствующая функция будет вызвана при нажатии комбинации `<Alt+a>`, если при этом не нажата клавиша `Control`.

Модификаторы, расположенные в начале трансляции, относятся ко всем событиям, указанным в ней. Например, следующие записи эквивалентны:

```
Shift<Btn2Down>, <Btn2Up> : KeyboardPrint()
```

и

```
Shift<Btn2Down>, Shift<Btn2Up> : KeyboardPrint()
```

Для определения в трансляциях событий от кнопок мыши используются модификаторы `Button1`, `Button2`, `Button3`, `Button4` и `Button5`, например, следующая запись

```
Button2<Key>a : KeyboardPrint()
```

означает, что соответствующая процедура будет вызвана, когда нажимается вторая кнопка мыши и клавиша «`a`».

Каждая трансляция может определить одно или несколько событий, разделяемых запятыми. Например, для определения последовательности событий, соответствующих нажатию и отпусканию первой кнопки мыши, можно использовать следующую строку:

```
<Btn1Down>, <Btn1Up> : KeyboardPrint()
```

Для задания нескольких нажатий и отпусканий используется поле «число повторений события». Так, например, трансляция

```
<Btn2Up> (2) : KeyboardPrint()
```
указывает, что соответствующая процедура будет вызвана при двойном нажатии второй кнопки мыши. Приведенная выше запись эквивалентна следующей
```
<Btn2Down>, <Btn2Up>, <Btn2Down>, <Btn2Up> :
KeyboardPrint()
```
Для задания большего числа повторений можно использовать символ "+", например, следующая запись
```
<Btn3Up>(2+) : KeyboardPrint()
```
означает, что соответствующая функция будет вызвана при 2-х и более нажатиях на третью кнопку мыши (максимальное число, которое может быть указано в поле «число повторений» – 9). По умолчанию временной интервал, во время которого считаются нажатия кнопки мыши, составляет 200 миллисекунд. Если между очередным нажатием кнопки и следующим проходит меньше 200 миллисекунд, то система рассматривает это как одно двойное нажатие (double click) соответствующей кнопки мыши. Для изменения указанного интервала используется ресурс объекта `XtNmultiClickTime`. Он может быть установлен функцией `XtSetMultiClickTime()`.

Поле «`аргументы`» в трансляции используется для задания третьего и четвертого аргументов action-процедуры. Например, если трансляция была определена следующим образом:
```
<Key>a : KeyboardPrint(1, JUSTAS, ALEX)
```
то при вызове соответствующей функции аргумент `psParams` будет указывать на список, содержащий строки «`1`», «`JUSTAS`» и «`ALEX`», а аргумент `nNumParams` будет равен 3.

Ниже приведены примеры задания таблицы соответствия в файле ресурсов
```
justas*mywidget.translations : #augment \
      <Btn2Down>        : Arm()\n\
    <Btn2Down>,    <Btn2Up>       :    Activate()
```

```
Disarm()\n\
    <Btn2Down>(2+)      : MultiArm()\n\
    <Btn2Up>(2+)       : MultiActivate()\n\
    <LeaveWindow>       : Leave()
```

и коде программы:

```
static char *psTable : "#augment \
    <Btn2Down>        : Arm()\n\
    <Btn2Down>,    <Btn2Up>       :    Activate()
Disarm()\n\
    <Btn2Down>(2+)      : MultiArm()\n\
    <Btn2Up>(2+)       : MultiActivate()\n\
    <LeaveWindow>       : Leave()";
```

Как мы уже упоминали ранее, если таблица задана в коде программы, то перед использованием она должна быть переведена во внутренний формат Xt процедурой `XtParseTranslationTable()`. Регистрируется таблица с помощью функций

```
XtAugmentTranslations (Widget prWidget,
            XtTranslations pTranslationsTable);
```

или

```
XtOverrideTranslations (Widget prWidget,
           XtTranslations pTranslationsTable);
```

Обе они регистрируют таблицу трансляции для объекта, заданного виджетом, объединяя таблицу `pTranslationsTable` с таблицей, уже имеющейся у виджета. При этом вторая процедура замещает существующие трансляции на новые, а первая оставляет их без изменения.

Следующий пример показывает, как добавить action-процедуру к объекту.

```
. . . . . .
Widget prWidget;
static  XtActionRec actions [ ] = ("PressMe",
KeyboardPrint);
static  char UserTranslations [ ] =
     "Alt <Key>a : PressMe";
. . . . . .
```

```
XtAugmentTranslations (prWidget,
     XtParseTranslationTable
(UserTranslations));
. . . . . .
```

В заключении сделаем еще одно замечание. Порядок расположения трансляций в соответствующей таблице существен, т.к. поиск нужного события в таблице начинается сверху и выполняется до первого найденного. Поэтому следует располагать записи таким образом, чтобы наиболее существенные события располагались раньше. Например, если таблица трансляции содержит следующий фрагмент

```
<Key>        : InputSymbol() \n\
<Key>Return     : EndInput()
```

то процедура, соответствующая действию `InputSymbol`, будет вызвана при нажатии клавиши `<Return>`, а функция для `EndInput` никогда не будет вызвана.

2.2.7. Обработчики событий

В Xt предусмотрен еще один механизм использования процедур для выполнения определенных действий при наступлении тех или иных событий. Это механизм обработчиков событий (event handler). Он позволяет определять отдельное событие (или их группу), при наступлении которого будет вызвана указанная программой процедура-обработчик. Обработчики событий менее гибки, чем action-процедуры, но зато значительно быстрее. Последнее происходит потому, что при вызове обработчиков событий не требуется поиск процедур по соответствующим таблицам.

Для того, чтобы программа могла использовать какую-то процедуру как обработчик событий, ее необходимо зарегистрировать с помощью процедуры `XtAddEventHandler()` или `XtInsertEventHandler()`. Первая из них имеет следующий прототип:

```
void XtAddEventHandler (Widget prWidget,
```

```
    EventMask nEventMask, Boolean
nNonMaskable,
    XtEventHandler pHandler, XtPointer
pUserData);
```

Здесь `prWidget` идентифицирует объект, к которому будет добавлена процедура-обработчик `pHandler`. `nEventMask` – комбинация флагов, задающая события, в ответ на которые будет вызываться регистрируемая процедура. Третий аргумент в процедуре `XtAddEventHandler()` – это логическая переменная, равная `True`, если обработчик будет вызываться для событий, которые всегда посылаются программе. Это `MappingNotify`, `ClientMessage`, `SelectionClear`, `SelectionNotify`, `SelectionRequest` (заметим, что последние три события связаны с механизмом общения между программами через системный буфер (clipboard)). Наконец, последний аргумент процедуры – `pUserData` – это указатель на данные, передаваемые в обработчик при его вызове. Если таковых нет, то аргумент должен быть равен `NULL`.

Заметим, что для одного и того же события в списке обработчиков может быть зарегистрировано несколько функций. `XtAddEventHandler()` добавляет функцию `pHandler` в конец этого списка.

Процедура, используемая как обработчик событий, должна иметь следующий прототип:

```
void HandlerProc (Widget prWidget, XtPointer
pUserData,
          XEvent *prEvent, Boolean
*pnContinue);
```

Здесь первый аргумент указывает на объект, для которого заказана реакция на событие. Второй аргумент указывает на данные, передаваемые в обработчик из программы. Он равен аргументу `pUserData`, задаваемому при вызове функции `XtAddEventHandler()`. Третий

аргумент – указатель на событие, инициировавшее обращение к обработчику событий. И, наконец, последний аргумент указывает, следует ли системе вызывать следующие зарегистрированные для данного события обработчики или нет. Перед вызовом процедуры система делает этот параметр равным `True`. Если программа не желает, чтобы событие обрабатывалось дальше, значение переменной, на которую указывает `pnContinue`, должно быть изменено на `False`.

Функция `XtInsertEventHandler()` также заносит процедуру в список обработчиков того или иного события, но она позволяет поместить ее в произвольное место списка.

`XtInsertEventHandler()` имеет следующий прототип:

```
void XtInsertEventHandler (Widget prWidget,
    EventMask nEventMask, Boolean
nNonMaskable,
    XtEventHandler pHandler, XtPointer
pUserData,
    XtListPosition nPosition);
```

Первые пять аргументов полностью идентичны описанным выше для процедуры `XtAddEventHandler()`. Последний аргумент указывает, в какое место списка обработчиков будет помещена функция `pHandler`. Аргумент может принимать одно из двух предопределенных значений: `XtListHead`, которое означает, что функция будет помещена в голову списка и, следовательно, будет вызываться раньше любого другого обработчика, зарегистрированного для данного объекта и события. Второе значение – `XtListTail`, которое указывает, что функция будет помещена в конец списка и будет вызываться после всех зарегистрированных процедур.

Как мы упоминали ранее, для одного и того же события

можно зарегистрировать несколько обработчиков событий, но при этом каждая функция с одинаковыми параметрами pUserData может встретиться в списке только один раз. Однако, если та же самая функция регистрируется с помощью XtInsertEventHandler(), то она будет добавлена в список, но передвинута либо в конец списка, либо в его начало.

Вызов процедур XtAddEventHandler() и XtInsertEventHandler() может осуществляться до и после реализации соответствующего виджета.

Ниже приводится фрагмент кода, в котором с помощью функции XtAddEventHandler() регистрируется функция-обработчик, которая будет вызываться при изменении фокуса ввода для данного виджета. Ее можно использовать для выполнения определенных действий при наступлении соответствующих событий.

```
. . . . . .
void CheckFocus (Widget prWidget, XtPointer
pUserData,
     XEvent *prEvent)
{
  . . . . . .
  if (event->type == FocusIn)
  {
    /* Выполняются действия при получении
фокуса ввода */
    . . . . . .
  }
  else
  {
    /* Выполняются действия при потере фокуса
ввода */
    . . . . . .
  }
}
. . . . . .
main (int argc, char *argv)
```

```
{
  . . . . . .
  XtAddEventHandler (prCoreWidget,
FocusChangeMask, False,
                (XtEventHandler) CheckFocus,
NULL);
  . . . . . .
}
```

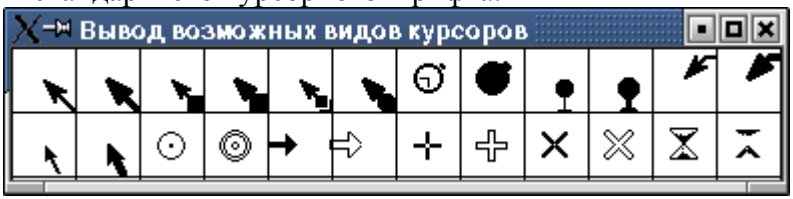
Когда процедура-обработчик становится не нужной, ее можно удалить из списка. Для этого можно использовать процедуру

```
void XtRemoveEventHandler (Widget prWidget,
     EventMask nEventMask, Boolean
nNonMaskable,
     XtEventHandler pHandler, XtPointer
pUserData);
```

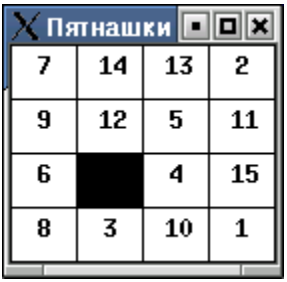
Она имеет те же самые аргументы, что и процедура `XtAddEventHandler()`. Если параметры в вызовах этих процедур не совпадают, то удаления указанного обработчика не произойдет. Так происходит, например, если параметр `pUserData` при вызове `XtRemoveEventHandler()` не совпадает с заданным при обращении к `XtAddEventHandler()`.

2.2.8. Лабораторная работа №7 «Объекты Xt и взаимодействие с ними»

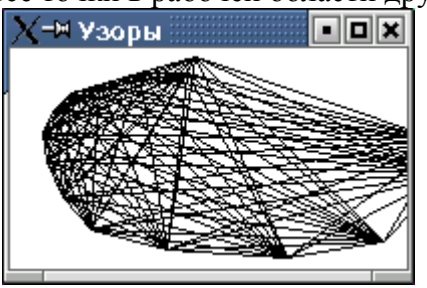
1. Составьте программу, выводящую в окно все символы стандартного курсорного шрифта.



2. Составьте программу игры «Пятнашки», выбирая перемещаемую ячейку мышью.



3. Составьте программу, которая по нажатию левой клавиши очищает рабочую область, при движении с нажатой левой клавишей рисует точку в позиции указателя мыши, а при отпускании левой клавиши соединяет все точки в рабочей области друг с другом.



4. Составьте программу, порождающую два расположенных рядом дочерних виджета, в которых отображаются графики функций $\sin(x)$ на отрезке $[-\pi; 2\pi]$ и $\exp(x)$ на отрезке $[-2; 2]$. Графики масштабировать по размеру виджетов.

2.3. Дополнительные возможности Xt

2.3.1. Ввод данных из файла или из внешнего устройства

В Xt предусмотрен механизм для работы с файлами (внешними устройствами) в асинхронном режиме. Приложение может зарегистрировать процедуру, которая будет вызываться по мере готовности данных или при возникновении ошибок чтения/записи. Для регистрации используется процедура:

```
XtInputId        XtAppAddInput    (XtAppContext
prAppContext,
           int nSource, XtPointer pCondition,
           XtInputCallbackProc pProcedure,
           XtPointer pUserData);
```

Здесь первый аргумент задает контекст программы, второй аргумент – это дескриптор файла (устройства), из которого будут читаться данные (до обращения к описываемой процедуре указанный файл должен быть открыт). Третий аргумент задает режим работы с файлом (устройством). Возможные значения этого аргумента перечислены ниже.

Значение	Описание
`XtInputReadMask`	файл используется только для чтения данных
`XtInputWriteMask`	файл используется для записи данных
`XtInputExceptMask`	процедура регистрируется для обработки ошибок
`XtInputNoneMask`	ввода-вывода нет, регистрируемая функция не вызывается

Заметим, что указанные константы не объединяются с помощью оператора `OR` (`|`).

Четвертый аргумент задает регистрируемую процедуру. Пятый аргумент используется для передачи данных в

pProcedure при ее вызове.

XtAppAddInput() возвращает число, позволяющее идентифицировать зарегистрированную функцию и условия ее работы.

Процедура чтения/записи данных и обработки ошибок должна иметь следующий прототип:

```
void InputDataProc (XtPointer pData,
       int          *nSource,          XtInputId
*nIdentificator);
```

Здесь первый аргумент определяет данные, являющиеся последним параметром в вызове процедуры XtAppAddInput(). Второй аргумент задает дескриптор файла (устройства), а третий аргумент есть идентификатор, возвращенный процедурой XtAppAddInput().

Когда обмен данными закончен, зарегистрированную функцию надо удалить с помощью XtRemoveInput (XtInputId nIdentificator).

Процедура XtAppAddInput() полностью аналогична описанной выше процедуре XtAppAddInput() за исключением того, что в ней отсутствует аргумент, указывающий на контекст программы.

Ниже приводится фрагмент кода, показывающий пример использования описанной процедуры XtAppAddInput().

```
. . . . . .
GetFileInputData(XtPointer    pUserData,    int
*nSource,
            XtInputId *identifier)
{
  char anBuffer [BUFSIZ];
  int nCountBytes;
  char *p = (char*) pUserData;

  if   (   (nCountBytes   =   read   (*nSource,
anBuffer, BUFSIZ) ) == -1)
    perror("GetFileInputData");
  else
```

```
   printf("%s : %d bytes\n", p, nCountBytes);
}

main (int argc, char **argv)
{
  XtAppContext prAppContext;
  Widget    prTopLevelWidget;
  int       nId;
  char     *pS = "Read data from TESTFILE";

  prTopLevelWidget                                =
XtVaAppInitialize(&prAppContext,
                  "XFileInput", NULL, 0,
                  &argc, argv, NULL, NULL);

  if    (nId    =    open("TESTFILE",    O_RDONLY,
S_IREAD) )<0
  {
   fprintf(stderr, "xfileinput:I/O error\n");
   exit (1);
  }

  XtAppAddInput(prAppContext,              nId,
XtInputReadMask,
             GetFileInputData,      (XtPointer)
pS);
  XtRealizeWidget(prTopLevelWidget);
  XtAppMainLoop(prAppContext);
}
```

Здесь программа открывает файл с именем «`TESTFILE`» и читает из него данные при помощи процедуры `GetFileInputData()`. Мы использовали последний аргумент функции `XtAppAddInput()` для передачи строки, которая должна печататься, если чтение прошло успешно.

2.3.2. Таймер

Xt предоставляет приложению возможность выполнять определенные действия через заданные промежутки

времени. Например, периодически отображать на экране текущее время в заданном окне и т.д. Для указанных целей используется специальный механизм Xt – таймер. Он обеспечивает вызов через заданный интервал времени специальной функции, заданной программой. Таймер создается, например, процедурой

```
XtIntervalId XtAppAddTimeOut (XtAppContext
prAppContext,
   unsigned long nInterval,
XtTimerCallbackProc pProcedure,
   XtPointer pUserData);
```

Здесь первый аргумент задает, как обычно, контекст программы. Второй аргумент определяет время задержки в миллисекундах, т.е. временной интервал, по истечении которого будет вызываться функция (таймер-процедура), заданная третьим аргументом данной процедуры. Используя четвертый аргумент, можно передать в вызываемую функцию произвольные данные.

Процедура возвращает значение типа `XtIntervalId`, которое однозначно идентифицирует зарегистрированный таймер и которое затем может быть использовано в качестве аргумента в процедуре `XtRemoveTimeOut(XtIntervalId)`, предназначенной для уничтожения таймера.

Регистрируемая функция, должна иметь следующий прототип:

```
void TimerProc (XtPointer pUserData,
XtIntervalId *Id);
```

Здесь `pUserData` – это указатель на данные, определенные четвертым параметром в вызове процедуры `XtAppAddTimeOut()`. Второй аргумент – это указатель на идентификатор таймера.

Отметим, что таймер автоматически уничтожается при вызове соответствующей процедуры. Поэтому, если необходимо, чтобы таймер функционировал и периодически проявлял себя каждые `nInterval`

миллисекунд, в вызываемой функции он должен пересоздаваться с помощью XtAppAddTimeOut().

Таймер может быть создан и зарегистрирован с помощью процедуры XtAddTimeOut(), которая отличается от описанной выше функции XtAppAddTimeOut() только отсутствием аргумента prAppContext.

Ниже приведен фрагмент кода, содержащий пример использования таймера.

```
void Count (Widget prWidget, XtIntervalId *Id)
{
   static int n = 0;

   if (n<10) {

XtAppAddTimeOut(XtWidgetToApplicationContext
(prWidget),
                1000, Count, prWidget);

    n++;
    printf ("%d seconds.\n", n);
 }
 else
  exit(0);
}

void main (int argc, char **argv)
{
  XtAppContext prAppContext;
  Widget    prTopLevelWidget;
  int       nId;

  prTopLevelWidget     =     XtVaAppInitialize
(&prAppContext,
                "Timer", NULL, 0,
                &argc, argv, NULL, NULL);
  XtAppAddTimeOut(prAppContext, 1000, Count,
                prTopLevelWidget);
```

```
  XtRealizeWidget(prTopLevelWidget);
  XtAppMainLoop(prAppContext);
}
```

В данном случае таймер работает, как секундомер, отсчитывая десять секунд, после чего программа завершается.

Мы использовали в приведенном фрагменте кода удобную процедуру `XtWidgetToApplicationContext()`, которая по данному виджету возвращает контекст программы.

2.3.3. Рабочие процедуры

«Рабочая» процедура (work-процедура) – это специальная функция, определяемая программой и вызываемая Xt, когда очередь событий пуста. Такие функции используются, как правило, для выполнения различных действий и вычислений в течение очень короткого времени. Как и таймер, каждая work-процедура должна быть зарегистрирована. Для этого используется, например, функция

```
XtWorkProcId XtAppAddWorkProc (XtAppContext
prAppContext,
                XtWorkProc pProcedure,
                XtPointer pUserData);
```

Здесь, как обычно, `prAppContext` – контекст приложения, `pProcedure` определяет вызываемую функцию, а `pUserData` задает передаваемые последней данные. Функция возвращает дескриптор рабочей процедуры, который, затем можно использовать для ее удаления.

Каждая work-процедура должна иметь следующий прототип:

```
Boolean WorkProc (XtPointer pUserData);
```

Здесь `pUserData` совпадает с третьим аргументом функции `XtAppAddWorkProc()`.

Если work-процедура возвращает `True`, то Xt

автоматически удаляет ее после того, как она завершит свою работу. В следующий раз, когда очередь событий приложения будет пуста, work-процедура не будет вызвана. Если же возвращаемое значение есть `False`, то процедура будет вызываться каждый раз, когда в очереди событий ничего нет. Удалить рабочую процедуру можно и явно, используя функцию `XtRemoveWorkProc(XtWorkProcId nId)`, задавая в качестве аргумента `nId` идентификатор функции, возвращенный процедурой `XtAppAddWorkProc()`.

Приложение может зарегистрировать несколько work-процедур.

Для регистрации рабочей процедуры можно использовать также и процедуру `XtAddWorkProc()`, которая отличается от описанной выше процедуры `XtAppAddWorkProc()` только отсутствием аргумента `prAppContext`.

2.3.4. Управление очередью событий

Как уже отмечалось выше, каждое приложение вызывает процедуру `XtAppMainLoop()` или `XtMainLoop()` для организации процесса получения и рассылки событий. Данные процедуры, в свою очередь, вызывают две процедуры `XtAppNextEvent()` (`XtNextEvent()`) и `XtDispatchEvent()`. Другими словами работа `XtAppMainLoop()` эквивалентна следующей последовательности операторов:

```
. . . . . .
XtAppContext prAppContext
. . . . . .
for ( ; ; ) {
  XEvent rEvent;
  XtAppNextEvent (prAppContext, &rEvent);
  XtDispatchEvent (&rEvent);
}
```

. . . . . .

Процедура `XtAppNextEvent()` имеет прототип:

```
void        XtAppNextEvent        (XtAppContext
prAppContext,
            XEvent *prEvent);
```

Здесь `prAppContext` – это контекст приложения, а второй аргумент используется для хранения информации о следующем событии в очереди событий программы. (`XtNextEvent()` отличается только тем, что отсутствует аргумент `prAppContext`).

Работает `XtAppNextEvent()` так: если в очереди первым идет событие о готовности данных в файле (внешнем устройстве), или там находится событие, инициирующее вызов соответствующей таймер-процедуры, то вызывается процедура (процедуры), зарегистрированные с помощью процедур `XtAppAddInput()` (`XtAddInput()`) или `XtAppAddTimeOut()` (`XtAddTimeOut()`). Если никаких событий в очереди нет, то вызываются work-процедуры, зарегистрированные процедурой `XtAppAddWorkProc()`. Если же в очереди есть событие от сервера, то `XtAppNextEvent()` удаляет его из очереди и переносит в структуру, на которую указывает ее второй аргумент.

Процедура `XtDispatchEvent()` имеет прототип:

```
void XtDispatchEvent (XEvent *prEvent);
```

Она анализирует событие и вызывает соответствующую callback-процедуру, action-процедуру или обработчик событий. Если таковых нет, то событие игнорируется. Если для события зарегистрировано несколько обработчиков событий или action-процедур, то порядок их выполнения не определен.

Каждое приложение может модифицировать описанный цикл получения и рассылки (обработки) событий. Ниже приводится фрагмент кода, показывающий, как это можно сделать:

```
void MyMainLoop (XtAppContext prAppContext)
{
  XEvent prEvent;
  . . . . . .
  for ( ; ; )
  {
    XtAppNextEvent (prAppContext, &prEvent);
    . . . . . .
  /* Предобработка события */
  . . . . . .
  XtDispatchEvent (&prEvent);
  . . . . . .
  /* Постобработка события */
  . . . . . .
  }
}
```

Xt предусматривает целый набор процедур для работы с очередью событий. Например, функция `XtAppPeekEvent()` позволяет получить информацию о следующем событии без удаления его из очереди. Процедура имеет следующий прототип:

```
Boolean      XtAppPeekEvent      (XtAppContext
prAppContext,
            XEvent *prEvent);
```

Здесь `prAppContext` – это контекст приложения; второй аргумент используется для хранения информации о следующем событии в очереди. Функция возвращает `True`, если событие в очереди есть X событие, в противном случае (т.е., если это событие, инициирующее вызов соответствующей таймер-процедуры, или какое-либо другое событие) возвращается `False`. Событие из очереди не удаляется.

Процедура `XtAppProcessEvent()` позволяет, задавая маску событий, выбирать из очереди нужные события и обрабатывать их. Функция имеет следующий прототип:

```
void      XtAppProcessEvent      (XtAppContext
prAppContext,
```

```
                XtInputMask nMask);
```

Здесь первый аргумент задает контекст приложения. Второй аргумент определяет маску событий, при этом последняя может состоять из одной или нескольких констант, разделенных оператором `OR` (`|`). Указанные константы следующие:

`XtIMXEvent` – X событие;

`XtIMTimer` – событие, инициирующее вызов таймер-процедуры;

`XtIMAlternateInput` – событие о готовности данных;

`XtIMAll` – все события.

Аналогичные описанным выше процедурам функции `XtPeekEvent()` и `XtProcessEvent()` отличаются только отсутствием аргумента `prAppContext`.

2.3.5. Акселераторы

Акселераторы в Xt похожи на action-процедуры с той лишь разницей, что событие (или последовательность событий), происходящее в одном виджете, инициирует вызов соответствующей action-процедуры для другого объекта.

Каждый виджете в Xt (кроме объектов классов `Object` и `RectObj`) имеет ресурс `XtNaccelerators`, наследуемый от класса `Core`. Данный ресурс – это таблица акселераторов, которая по формату аналогична таблице трансляции. Она содержит описание событий и соответствующих действий. Задается данный ресурс так же, как и ресурс `XtNtranslations`. Ниже приводится пример его определения в ресурсном файле:

```
justas*XmDialogShell*mywidget.acceleratos :\n\
              <KeyPress> : Set() Cancel()
```

Если значение ресурса `XtNaccelerators` задано для объекта `W`, то программа должна указать виджет, в котором будут происходить события, вызывающие выполнение соответствующих action-процедур, указанных в таблице

акселераторов. Для этого можно использовать процедуру `XtInstallAccelerators()` или `XtInstallAllAccelerators()`. Первая из указанных функций имеет следующий прототип:

```
void XtInstallAccelerators (Widget prDestination,
                Widget prSource);
```

Здесь `prDestination` – это виджет, в котором будут происходить события, которые приводят к выполнению action-процедур, заданных в таблице акселераторов для объекта, определяемого вторым аргументом.

Приведенная процедура может вызываться до и после реализации виджетов, указанных в качестве ее параметров.

Когда создается объект, Xt переводит таблицу акселераторов во внутренний формат и размещает ее в соответствующей структуре виджета. При вызове функции `XtInstallAccelerators()` таблица акселераторов объединяется с таблицей соответствия или замещает последнюю. Все зависит от директивы «`#replace`», «`#override`» или «`#augment`», указанной при описании таблицы акселераторов. По умолчанию действительным считается режим «`#augment`». Процедура `XtRemovelAccelerators()` имеет обратное действие – она восстанавливает таблицу соответствия для объекта.

Процедура `XtInstallAllAccelerators()` имеет такой же прототип, что и описанная выше функция `XtInstallAccelerators()`, и используется для задания акселераторов самого объекта и его потомков.

Порядок шагов при вызове акселератора описан ниже. Когда в объекте `prDestination` происходит какое-то событие, то для виджета `pSource` делается следующее:

- Xt проверяет для данного виджета таблицу трансляции; если в ней такого события нет, то ничего не делается;
- если событие есть, то по нему определяются

имена соответствующих действий;

- сканируется зарегистрированная action-таблица, и по именам действий находятся адреса процедур; когда адрес становится известен, соответствующая функция вызывается.

В заключении отметим несколько особенностей, связанных с применением акселераторов.

1. При определении таблицы акселераторов в файле ресурсов, не допускается применение сокращенной аббревиатуры события, кроме «`Key`» (в случае `KeyPress`). Необходимо использовать только полное описание события.
2. Не разрешены символы `None`, `Meta`, `Hyper`, `Super`, `Alt`.
3. Параметры, передаваемые в action-процедуры, должны быть заключены в двойные кавычки.
4. Для конвертирования таблицы акселераторов во внутреннее представление Xt используется процедура `XtParseAcceleratorTable(String sTable)`, где строка `sTable` определяет таблицу акселераторов.
5. Для задания всех акселераторов приложения можно использовать процедуру `XtInstallAllAccelerators(prTopLevel, prTopLevel)`, где виджет `prTopLevel` – родительский для всех остальных объектов программы.

2.3.6. Процедуры, предназначенные для работы с окнами объектов

Xt поддерживает несколько способов работы с окнами виджетов. Так, программа, используя процедуры, предоставляемые Xt, может показать окно, установить для него тот или иной режим работы, и закрыть окно, когда последнее становится не нужным. В основном эти процедуры используются для создания всплывающих (pop up) меню и диалогов. Рассмотрим кратко эти способы.

Первый способ. В Xt есть три предопределенные

процедуры для показа окна объекта и одновременного задания режима обработки событий (Xt grab mode). Может быть сохранен нормальный процесс рассылки событий между объектами приложения, но может быть сделано и так, что Xt будет направлять события, возникающие в результате действий пользователя, в виджет независимо от того, в каком объекте они происходили. Режим обработки событий называется в Xt «режимом захвата» (grab mode).

Захват может быть двух типов: исключительным (exclusive) и неисключительным (nonexclusive). Разница между ними проявляется, когда программа создает, например, многоуровневое меню. Рассмотрим следующий пример. Пусть мы открыли первое подменю, затем второе, а после него третье. Теперь пользователь нажал на кнопку мыши. Если режим захвата exclusive, соответствующее событие будет передано виджет, представляющему третье подменю. Если режим захвата nonexclusive, то оно будет послано тому подменю, в котором находится курсор мыши. Если же курсор не попал ни в одно подменю, то событие опять таки получит последний (третий) виджет.

Процедуры, отображающие объект и задающие режим обработки событий, следующие: `XtCallbackNone()`, `XtCallbackExclusive()`, `XtCallbackNonexclusive()`. Все они имеют стандартный набор параметров, определенный в Xt для процедур обратного вызова. Идентификатор объекта, окно которого должно быть показано на экране, передается через аргумент `pUserData`.

Первая процедура используется для показа окна виджета и не меняет способ обработки событий.

Вторая и третья процедуры используются для отображения окна виджета и одновременного задания режимов exclusive и nonexclusive соответственно.

Ниже приводится фрагмент кода, содержащий пример использования одной из перечисленных выше процедур. Здесь мы используем объект `PushButton` («нажимаемая

кнопка») из библиотеки OpenMotif.

```
. . . . . .
main (int argc, char **argv)
{
   Widget  prShellWidget;     /*  Показываемый
shell-объект */
   Widget  prPushButton;       /*  "Нажимаемая
кнопка"    */
   . . . . . .
   prShellWidget = XtCreatePopupShell ( . . .
);
   prPushButton = XtCreateWidget ( . . .);
   . . . . . .
   XtAddCallback                  (prPushButton,
XmNactivateCallback,
      XtCallbackExclusive, prShellWidget);
   . . . . . .
}
. . . . . .
```

Цель данного кода – показать всплывающее (pop up) окно при нажатии на кнопку. Для этого мы сначала создаем всплывающий shell-объект с помощью процедуры `XtCreatePopupShell()` и заносим стандартную процедуру `XtCallbackExclusive()` в список callback-процедур кнопки `prPushButton`. Этот список соответствует ресурсу `XmNactivateCallback`. Занесенные в него процедуры вызываются в ответ на нажатия на объект с помощью мыши или клавиши клавиатуры. Последний параметр функции `XtAddCallback()` позволяет нам сообщить стандартной процедуре `XtCallbackExclusive()`, окно какого виджета должно быть показано на экране.

Для закрытия окон виджетов используется callback-процедура `XtCallbackPopdown()`. При ее вызове параметр `pUserData` должен указывать на структуру типа `XtPopdownId`, определяемую следующим образом:

```
typedef struct {
```

```
   Widget shell_widget;
   Widget enable_widget;
} XtPopdownIdRec, *XtPopdownId;
```

Здесь первое поле определяет shell-объект, который будет закрыт, второе поле задает объект, которой инициировал его показ.

Ниже приводится фрагмент, содержащий пример использования описанной процедуры.

```
. . . . . .
Widget    prShellWidget, prPushButton;
XtPopdownIdRec  rRec;
. . . . . .
main (int argc, char **argv)
{
   . . . . . .
   prShellWidget = XtCreatePopupShell ( . . .
);
   prWidget      = XtCreateManageWidget ( . . .
);
   . . . . . .
   rRec.shell_widget  = prShellWidget;
   rRec.enable_widget = prPushButton;
   . . . . . .
   XtAddCallback                (prPushButton,
XmNactivateCallback,
     XtCallbackPopdown, &rRec);
   . . . . . .
}
. . . . . .
```

Второй способ. Xt предусматривает две специальных action-процедуры `XtMenuPopup()` и `XtMenuPopdown()`, которые применяются для отображения и закрытия окон объектов соответственно. Имена процедур могут быть использованы в таблицах трансляции программы. Функция `XtMenuPopup()` имеет следующий прототип:

```
void XtMenuPopup (String shellname);
```

Здесь аргумент процедуры используется для задания имени отображаемого shell-объекта. Данная action-

процедура может быть определена в таблице трансляции для следующих событий: `ButtonPress`, `KeyPress`, `EnterWindow`.

Процедура `XtMenuPopdown()` аналогична описанной выше функции только используется для закрытия указанного shell-объекта.

Третий способ. Xt предусматривает для непосредственного отображения на экране и закрытия окон виджетов также и отдельные процедуры `XtPopup()`, `XtPopupSpringLoaded()` и `XtPopdown()`.

Первая процедура используется для непосредственного размещения на экране окна виджета и имеет следующий прототип:

```
void XtPopup (Widget prShellWidget,
       XtGrabKind grabmode);
```

Здесь первый аргумент задает shell-объект. Второй аргумент определяет режим обработки событий. Возможные его значения следующие:

XtGrabNone – нормальный режим работы;

XtGrabExclusive – исключительный (exclusive) режим захвата;

XtGrabNonexclusive – неисключительный (nonexclusive) режим захвата.

Вторая функция – `XtPopupStringLoaded()` – используется также для отображения окон и отличается от `XtPopup()` в частности тем, что автоматически задает режим обработки событий exclusive. Функция имеет следующий прототип:

```
void XtPopupStringLoaded (Widget
prShellWidget);
```

Здесь аргумент аналогичен первому аргументу предыдущей процедуры.

Третья процедура используется для закрытия окна объекта и имеет прототип, аналогичный предыдущей функции.

Заметим, что, как правило, каждое множество виджетов (например, OpenMotif, Open Look и др.) имеет свои удобные процедуры для создания меню и диалогов, поэтому мы не будем более подробно останавливаться на данном вопросе.

2.3.7. Программы, имеющие много объектов (окон) верхнего уровня

Процедура инициализации `XtAppInitialize()` или аналогичная ей возвращает указатель на объект класса `ApplicationShell`, который непосредственно контактирует с менеджером окон и является началом дерева виджетов. Но приложение может иметь не одно, а несколько объектов верхнего уровня. Для их создания можно использовать процедуры `XtAppCreateShell()` (`XtVaAppCreateShell()`, `XtCreateApplicationShell()`). Создаваемые виджеты принадлежат, как правило, классу `TopLevelShell`.

В приводимом ниже примере приложение создает для своих нужд три окна верхнего уровня:

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>

int main (int argc, char **argv)
{
  Widget                topLevel1,  topLevel2,
topLevel3,
               CoreWidget1,        CoreWidget2,
CoreWidget3;
  XtAppContext prAppContext;

  topLevel1             =     XtVaAppInitialize
(&prAppContext, "Justas1",
                                   NULL,     0,
&argc, argv, NULL, NULL);
```

```
  topLevel2     = XtAppCreateShell  ("justas2",
"Justas2",

topLevelShellWidgetClass,
                                    XtDisplay
(topLevel1), NULL, 0);
  topLevel3     = XtAppCreateShell  ("justas3",
"Justas3",

topLevelShellWidgetClass,
                                    XtDisplay
(topLevel1), NULL, 0);

  CoreWidget1 = XtCreateManagedWidget  ("Core",
widgetClass,

topLevel1, NULL, 0);
  CoreWidget2 = XtCreateManagedWidget  ("Core",
widgetClass,

topLevel2, NULL, 0);
  CoreWidget3 = XtCreateManagedWidget  ("Core",
widgetClass,

topLevel3, NULL, 0);

  XtVaSetValues (CoreWidget1,
                 XtNwidth,    100,    XtNheight,
100, NULL);
  XtVaSetValues (CoreWidget2,
                 XtNwidth,    200,    XtNheight,
200, NULL);
  XtVaSetValues (CoreWidget3,
                 XtNwidth,    400,    XtNheight,
400, NULL);

  XtRealizeWidget (topLevel1);
  XtRealizeWidget (topLevel2);
  XtRealizeWidget (topLevel3);
```

```
  XtAppMainLoop (prAppContext);
  return 0;
}
```

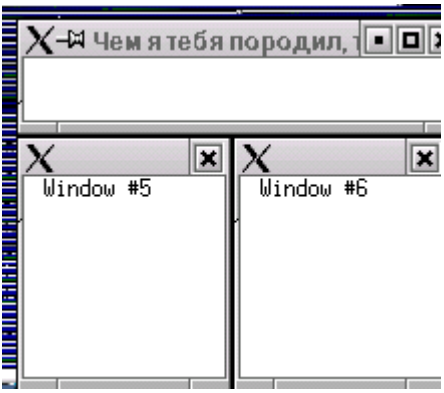
Заметим, что существует еще один способ создания в программе нескольких объектов верхнего уровня. В частности, можно поступать следующим образом: сначала создается shell-объект класса `ApplicationShell`, который никогда не показывается на экране, все остальные shell-объекты верхнего уровня создаются, как потомки данного родителя с помощью процедуры `XtCreatePopupShell()`.

2.3.8. Лабораторная работа №8 «Дополнительные возможности Xt»

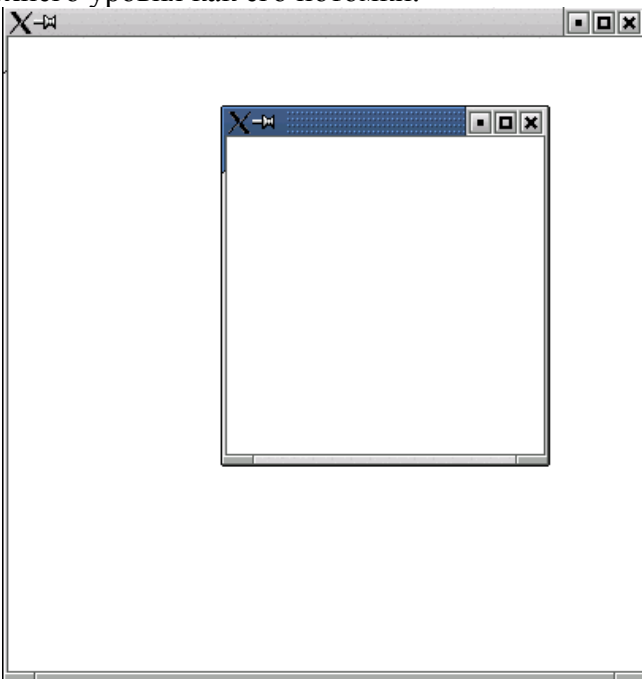
1. Создайте окно с «идущими часами», масштабируемыми при изменении размеров окна. При запуске программы без параметров должна выводится подсказка по используемым ключам: -e – часы электронные, -d – часы со стрелками

2. Используя work-процедуру, модифицируйте предыдущую программу таким образом, чтобы через 30 секунд после начала она завершала свою работу.

3. Создайте программу, которая по нажатию клавиши мыши в основном окне создает новое окно (не более 100 одновременно), а по нажатию клавиши мыши в дочернем окне удаляет его. Если дочернее окно существует более одной минуты, оно должно самоудаляться.



4. Создайте shell-объект класса ApplicationShell, который никогда не показывается на экране, и два shell-объекта верхнего уровня как его потомки.



2.4. Xt и ресурсы программ

2.4.1. Формат файла описания ресурсов

Подробно формат файла описания ресурсов (или просто файла ресурсов) был нами рассмотрен в предыдущем разделе. Но когда программист работает с Xt, то возникает вопрос, как конкретно определяется значение того или иного параметра виджета. Делается это следующим образом. Пусть объект с именем «WnameN» и класса «WclassN» имеет родителей с именами «WnameN-1», ..., «Wname1», принадлежащих классу «WclassN-1», ..., «Wclass1» соответственно, причем «Wname1» – корень дерева виджетов программы. Тогда, чтобы задать значение его атрибута XtNA, в ресурсном файле должна быть строка:

```
<имя программы>. "Wname1". . . . . "WnameN".A
: <значение>
```

Например, пусть в программе с именем «justas» объект «dialogBox» принадлежит классу TopLevelShell и имеет два родительских виджета «Core» класса Core и «appShell» класса ApplicationShell. Тогда значение его атрибута XtNheight (высота окна) равное 100 можно задать так:

```
justas.appShell.Core.dialogBox.height : 100
```

Вместо имени любого объекта иерархии можно указать его класс. В этом случае значение ресурса будет распространяться на все объекты, у которых в иерархии на соответствующем месте стоит виджет указанного класса. Так, например, строки

```
justas.appShell.Core1.dialogBox.height : 100
justas.appShell.Core2.dialogBox.height : 200
```

означают, что виджет с именем «dialogBox» и родителем «Core1» имеет высоту 100. А виджет с именем «dialogBox» и родительским виджетлм «Core2» имеет высоту 200. Строка же

```
justas.appShell.Core.dialogBox.height : 200
```

говорит, что для обоих объектов высота равна 200. В строке, задающей ресурс, можно употреблять символ «*».

2.4.2. Создание базы данных ресурсов программы

Загрузка ресурсов производится инициализационными процедурами, например `XtAppInitialize()`, которые конструируют базу данных из различных ресурсных файлов, опций командной строки и других источников. Ниже описывается последовательность шагов Xt, выполняемая при поиске и загрузке ресурсов. Если встречаются две одинаковые спецификации какого-то ресурса виджета, то будет использована первая встретившаяся (данная последовательность шагов принята в версии X11R5, в X11R4 поиск и загрузка ресурсов выполняется в обратном порядке).

Первый шаг. Загрузка ресурсов из командной строки. Xt поддерживает стандартное множество опций, комбинации которых могут указываться в командной строке при запуске программы. Они перечислены в таблице, приведенной ниже.

Опция	Имя ресурса	Тип	Описание и примеры

`-bg`
`-background`
	`*background`	`String`	цвет фона
			`-bg blue`
			`-background white`

`-bd`
`-bordercolor`
	`*borderColor`	`String`	цвет границы
			`-bd black`
			`-bordercolor red`

`-bw` `-borderwidth`	`.borderwidth`	`Integer`	ширина края окна в пикселях `-bw 2`
`-display`	`.display`	`String`	дисплей для связи с сервером

Опция	Имя ресурса	Тип	Описание и примеры
			-display odessa:0
-fg -foreground	*foreground	String	цвет переднего плана -fg red -foreground black
-fn -font	*font	String	имя шрифта -fn 9x15 -font 9x15
-geometry	.geometry	String	размер и положение окна -geometry = 80x24
-iconic	.iconic	None	если значение опции «on», то окно программы будет показано в минимизированном виде
-name	.name	String	имя программы -name Justas
-reverse -rv	*reverseVideo	None	если значение ресурса «on», то выводимое изображение появится в инверсном виде
+rv	*reverseVideo	None	изображение будет нормальным
-title	.title	String	заголовок окна программы

Опция	Имя ресурса	Тип	Описание и примеры
			`-title Justas`
`-xrm`	`значение аргумента`	`String`	ресурс и его значение задаются аргументом опции `-xrm "*.height: 100"`

В данной таблице точка, предшествующая имени ресурса, означает, что опции могут быть использованы для установки ресурса только в виджете, принадлежащих классу `TopLevelShell` или его подклассу. В свою очередь, «`*`» означает, что ресурс может быть установлен для произвольного виджета приложения.

Инициализационные процедуры понимают также любую уникальную аббревиатуру предопределенных опций, например:

```
./justas -back green
```

Xt предоставляет также возможность передавать в командной строке нестандартные опции. Для того, чтобы определить их и получить соответствующие значения, используются третий и четвертый аргументы процедуры `XtAppInitialize()` (`XtVaAppInitialize()`, `XtInitialize()`).

Ниже приведен фрагмент кода и пример командной строки, иллюстрирующие описанный выше механизм получения значений ресурсов.

```
. . . . . .
static XrmOptionDescRec arCommandOptions [ ] =
{
  { "-delay", "*delay", XrmoptionSepArg, NULL
},
  { "-debug", "*debug", XrmoptionNoArg, "True"
},
```

```
} ;
. . . . . .
main (int argc, char **argv)
{
  Widget prWidget;
  . . . . . .
  prWidget = XtInitialize (argv [0], "Justas",
arCommandOptions,
    XtNumber()    (arCommandOptions),    &argc,
argv);
  . . . . . .
}
```

Пример командной строки:

```
./justas -debug -delay 15
```

Отметим также, что если имя опции, данное пользователем, совпадает с предопределенной стандартной опцией, то из значений, указанных в командной строке, будут использоваться лишь ресурсы, соответствующие стандартным опциям.

Второй шаг. Загружается файл, на который указывает переменная среды `XENVIRONMENT` (если таковая задана). Данная переменная содержит полный путь к файлу, включая и его имя. Если же переменная `XENVIRONMENT` не установлена, то менеджер ресурсов будет пытаться подгрузить файл «`.Xdefaults-<hostname>`», находящийся в домашней (home) директории пользователя. Здесь `<hostname>` – это имя компьютера, на котором происходит запуск программы.

Третий шаг. Если корневое окно (root window) экрана имеет ресурсы, загруженные в свойство `RESOURSE_MANAGER` программой `xrdb`, то они также добавляются в базу данных ресурсов приложения. Если же корневое окно не имеет такого свойства, то подгружаются данные, определенные в файле «`.Xdefaults`», находящемся в домашней директории пользователя.

Четвертый шаг. Если задана переменная среды

XAPPLRESDIR, то менеджер ресурсов будет пытаться загрузить ресурсный файл с именем «<$XAPPLRESDIR>/<classname>», где <classname> – имя класса программы. Если файла нет, Xt пытается прочитать другой файл с именем «<$APPLRESDIR>/<$LANG>/<classname>» (здесь LANG – переменная среды, задающая язык системы). Если переменная XAPPLRESDIR не установлена, то соответствующие файлы ищутся в домашней директории. Отметим, что в системе могут быть предусмотрены и некоторые другие переменные среды, позволяющие задать путь к файлу ресурсов (например, XUSERFILESEARCHPATH и др.).

Пятый шаг. Xt ищет следующий файл:
"/usr/X11R6/lib/app-defaults/<classname>"

Если этот файл существует, он загружается в базу данных ресурсов.

Шестой шаг. Рассматриваются параметры, переданные с помощью пятого и шестого аргументов инициализационных процедур. Данная возможность может быть использована в случае, если значение какого-либо ресурса осталось неопределенным. Передаваемые через параметры процедур значения ресурсов представляются в виде массива указателей на строки. Каждая строка имеет вид: «<описание ресурса> : <значение ресурса>». Здесь <описание ресурса> такое же, как и в ресурсном файле. Ниже приведен пример этого способа определения параметров программы.

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

extern void DrawHellowString();

void main (int argc, char **argv)
{
  Widget  topLevel, fallback;
```

```
   XtAppContext prAppContext;

   static String Fallback [ ] = {
     "Fallback*background : blue",
     "Fallback.Core.width  : 200",
     "Fallback.Core.height : 200",
     NULL,
   } ;

   topLevel         =         XtVaAppInitialize
(&prAppContext, "Fallback",
    NULL, 0, &argc, argv, Fallback, NULL);

   fallback  =  XtCreateManagedWidget  ("Core",
widgetClass,
    topLevel, NULL, 0);

   XtAddEventHandler  (fallback,  ExposureMask,
False,
          DrawHellowString, NULL);
   XtRealizeWidget (topLevel);

   XtAppmainLoop (prAppContext);
}

void   DrawHellowString   (Widget   prWidget,
XtPointer pData,
      XEvent *prEvent, Boolean *pbContinue)
{
  Display *prDisplay = XtDisplay (prWidget);
  Window nWindow  = XtWindow (prWidget);
  GC   prGC;

  if (prEvent->type == Expose)
  {
    prGC  =  XCreateGC  (prDisplay,  nWindow,  0,
NULL);
    XDrawString (prDisplay, nWindow, prGC, 10,
50,
```

```
     "Hello, world!", strlen ("Hello, world!")
);
    XFreeGC (prDisplay, prGC);
   }
}
```

2.4.3. Получение ресурсов программы

Для получения ресурсов приложения из базы данных, в Xt предусмотрена процедура

```
void     XtGetApplicationResources     (Widget,
prWidget,
  XtPointer pBase, XtResourseList prResources,
  Cardinal  nItem,  ArgList  prArgs,  Cardinal
nArgs);
```

Здесь `prWidget` задает некоторый объект приложения. По нему определяется имя и класс программы. Второй аргумент – это адрес области памяти, где будут размещены получаемые значения ресурсов. Третий и четвертый аргументы задают массив и число структур типа `XtResource`, определяющих нужные ресурсы. Последние аргументы `prArgs` и `nArgs` используются для того, чтобы запретить модификацию некоторых ресурсов пользователем. Если ресурс присутствует в базе данных программы и в списке `prArgs`, то его значение берется из списка.

Структура `XtResource` имеет следующее определение:

```
typedef struct {
   String   resource_name;
   String   resource_class;
   String   resource_type;
   Cardinal resource_size;
   Cardinal resource_offset;
   String   default_type;
   XtPointer default_addr;
} XtResource, *XtResourceList;
```

Здесь

`resource_name` – имя ресурса. Где это возможно, для

указания имени ресурса следует использовать константы, определенные в файле «X11/StringDefs.h». Например, XtNforeground, XtNbackground и т.д.

resource_class – задает класс ресурса. Для многих классов его имя совпадает с именем ресурса, за исключением того, что используется префикс «XtC», и первая буква имени класса – заглавная (например, для ресурса XtNforeground соответствующий класс XtCForeground).

resource_type – определяет тип ресурса. Можно задавать произвольный тип данных, включая и любой тип, определенный программой. Определения общих типов ресурсов даны в файле «X11/StringDefs.h». Для удобства тип ресурса состоит из имени с префиксом «XtR». Если в системе зарегистрирована соответствующая процедура-конвертор, то при вызове XtGetApplicationResources() Xt преобразует значение ресурса из базы данных (строку) к типу, задаваемому полем resource_type.

resource_size – задает размер ресурса в байтах.

resource_offset – определяет смещение от адреса, задаваемого параметром pBase процедуры XtGetApplicationResources(), начиная с которого размещается значение ресурса, «вынимаемое» из базы данных.

default_type – задает для ресурса тип значения, принимаемый по умолчанию.

default_addr – задает (или указывает на) значения ресурса по умолчанию. Оно используется Xt, если на ресурс нет ссылок в базе данных программы.

В Xt предусмотрены два специальных типа ресурсов, которые используют только значения по умолчанию, задаваемые полями default_type и default_addr. Первый – XtRImmediate, который указывает, что значение

ресурса не надо искать в базе, поскольку оно определяется непосредственно полем `default_addr`. Второй – `XtRCallProc`, который указывает, что значение, заданное в поле `default_addr`, это указатель на процедуру, которая возвращает значение ресурса. Эта функция должна иметь прототип:

```
void ResourceDefaultProc (Widget prWidget,
  int nOffset, XrmValue *prValue);
```

Здесь `prWidget` задает объект, ресурс которого должен быть получен. `nOffset` задает местоположение ресурса в структуре виджета. `prValue` определяет адрес для сохранения полученного значения. Xt автоматически определяет нужные параметры и вызывает данную процедуру, сохраняя в `prValue` полученное значение.

В следующей таблице приведены основные типы ресурсов, принятые в Xt, соответствующие им типы данных и/или пояснения.

Тип ресурса	Тип данных
`XtRAcceleratorTable`	`XtRAccelerators` (таблица акселератов)
`XtRAtom`	`Atom`
`XtRBitmap`	`Pixmap`
`XtRBoolean`	`Boolean`
`XtRBool`	`Bool`
`XtRCallback`	`XtCallbackList` (список callback-процедур)
`XtRCallProc`	`XtCallbackList` (список callback-процедур)
`XtRCardinal`	`Cardinal`
`XtRColor`	`XColor`
`XtRColormap`	`Colormap`
`XtRCursor`	`Cursor`
`XtRDimension`	`Dimension`
`XtRDisplay`	`Display`
`XtREnum`	`XtEnum`
`XtRFile`	`File *`

Тип ресурса	Тип данных
XtRFloat	float
XtRFont	Font
XtRFontStruct	XFontStruct *
XtRFunction	(*) ()
XtRGeometry	String
XtRImmediate	см. выше
XtRInitialState	int
XtRInt	int
XtRLongBoolean	long
XtRObject	Object
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	XtPointer
XtRPosition	Position
XtRScreen	Screen *
XtRShort	short
XtRString	char *
XtRStringArray	String *
XtRStringTable	char **
XtRTranslationTable	XtRTranslations (таблица соответствия)
XtRUnsignedChar	unsigned char
XtRVisual	Visual *
XtRWidget	Widget
XtRWidgetClass	WidgetClass
XtRWidgetList	WidgetList
XtRWindow	Window

Ниже приводится пример использования описанного выше механизма получения ресурсов приложения.

```
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

typedef struct {
   Pixel   pColor;
   int     nLine;
```

```
   Boolean bFlag;
} trAppData, *prAppData;

static XtResource prResources[ ] = {
  {  XtNbackground,  XtCBackground,  XtRPixel,
sizeof (Pixel),
   XtOffset  (prAppData,  pColor),  XtRString,
"Green"},
  { "line", "Line", XtRInt, sizeof (int),
   XtOffset (prAppData, nLine), XtRImmediate,
(XtPointer) 20},
  {   "flag",   "Flag",   XtRBoolean,   sizeof
(Boolean),
   XtOffset (prAppData, bFlag), XtRImmediate,
(XtPointer) True}
} ;

void main (int argc, char **argv)
{
  Widget  topLevel, Core;
  trAppData  rData;
  XtAppContext prAppContext;

  memset (&rData, 0, sizeof (trAppData));

  topLevel = XtVaAppInitialize (&prAppContext,
"Justas",
          NULL, 0, &argc, argv, NULL, NULL);

  XtGetApplicationResources (topLevel, &rData,
prResources,
        3, NULL, 0);
  printf ("background = %d, Line = %d, Flag =
%d\n",
   rData.pColor, rData.nLine, rData.bFlag);

  Core   =   XtVaCreateManagedWidget   ("Core",
widgetClass,
       topLevel,   XtNwidth,   300,   XtNheight,
```

```
300, NULL);

  XtRealizeWidget (topLevel);
  XtAppMainLoop (prAppContext);
}
```

В программе используется макрос `XtOffset()`, который позволяет определить смещение в байтах от начала структуры до заданного поля. Определение макроса находится в файле «`X11/Intrinsic.h`». Формат его выглядит следующим образом:

```
Cardinal XtOffset (Type type, Field field);
```

Здесь первый аргумент – тип указателя, специфицирующего структуру данных пользователя, второй аргумент – имя устанавливаемого ресурса (поле) в данной структуре.

2.4.4. Процедуры, преобразующие значения ресурсов от одного типа к другому (конвертеры)

Значения параметров в файлах ресурсов представляются в виде строк. Перед использованием в программе их надо преобразовать к требуемому типу. Так, например, значение атрибута `XtNwidth` задается строкой «`100`», но должно быть превращено в целое число. Для этого Xt и программы регистрируют специальные процедуры, преобразующие значения из одной формы представления в другую (т.е. из одного типа в другой). Их называют «конвертеры» (converter) («процедуры-конверторы», «преобразователи»).

Механизм использования конвертеров следующий. Когда программа получает ресурс с помощью `XtGetApplicationResources()` или создает виджет, обращаясь к `XtCreateWidget()` или аналогичной процедуре, Xt достает значение ресурса из базы данных. Это значение представлено в виде строки. Xt знает тип атрибута (обозначим его `T`) и ищет зарегистрированный

конвертер «строка Т». Если таковой есть, то он вызывается и значение ресурса переводится в требуемую форму. Если же конвертера нет, параметр получает значение по умолчанию.

Мы описали работу преобразователя «строка произвольный тип». Однако можно создать и зарегистрировать конвертер «произвольный тип произвольный тип» и использовать его для своих нужд.

Ниже, в таблице, приведены преобразования типов данных, поддерживаемые самой Xt.

Откуда	Куда	Описание
XtRString	XtRAcceleratorTable	преобразует строку, задающую таблицу акселераторов, во внутреннюю форму Xt
XtRString	XtRAtom	преобразует строку, содержащую имя свойства, в соответствующий атом
XtRString	XtRBoolean	конвертирует строки «True», «False», «yes», «no», «on», «off» в соответствующее логическое значение
XtRString	XtRBool	аналогично предыдущему
XtRString	XtRCursor	преобразует стандартное название курсора для X Window в соответствующий идентификатор

Откуда	Куда	Описание
XtRString	XtRDimension	конвертирует ширину и высоту к типу Dimension
XtRString	XtRDisplay	преобразует имя дисплея и возвращает указатель на структуру типа Display
XtRString	XtRFile	преобразует имя файла и возвращает его дескриптор
XtRString	XtRFloat	конвертирует строку, задающую число, в формат float
XtRString	XtRFont	преобразует имя шрифта и возвращает его идентификатор
XtRString	XtRFontStruct	преобразует имя шрифта и возвращает указатель на структуру типа XFontStruct
XtRString	XtRInitialState	конвертирует строки «Normal» или «Iconic» в символы NormalState или IconicState соответственно
XtRString	XtRInt	конвертирует строку, содержащую число, в формат int
XtRString	XtRPixel	преобразует строку, содержащую имя цвета (например,

Откуда	Куда	Описание
		black или #FF0000), в значение пикселя
XtRString	XtRPosition	преобразует значения x и y координат к типу Position
XtRString	XtRShort	конвертирует строку, содержащую число, в форму short
XtRString	XtRTranslationTable	преобразует строку, задающую таблицу трансляции, во внутренний формат Xt
XtRString	XtRUnsignedChar	конвертирует строку, содержащую число, в формат unsigned char
XtRString	XtRVisual	преобразует строку, задающую тип палитры, и возвращает указатель на структуру типа Visual
XtRPixel	XtRColor	конвертирует значение пикселя в указатель на структуру типа XColor
XtRInt	XtRBool	конвертирует целое в логическое
XtRInt	XtRColor	конвертирует целое в XColor
XtRInt	XtRDimension	конвертирует целое в Dimension
XtRInt	XtRFloat	конвертирует целое в

Откуда	Куда	Описание
		плавающее
XtRInt	XtRFont	конвертирует целое в Font
XtRInt	XtRPixel	конвертирует целое в значение пикселя
XtRInt	XtRPixmap	конвертирует целое в Pixmap
XtRInt	XtRPosition	конвертирует целое в Position
XtRInt	XtRShort	конвертирует целое в short
XtRInt	XtRUnsignedChar	конвертирует целое в беззнаковое целое

Процедура-конвертор должна иметь следующий прототип:

```
void ConverterProc (XrmValue *prArgs, Cardinal
*nArgs,
    XrmValue *prFromVal, XrmValue *prToVal);
```

Здесь аргументы prArgs, prFromVal, prToVal – это указатели на структуры типа XrmValue.

Процедура-конвертор должна преобразовывать данные из структуры prFromVal и результат помещать в структуру prToVal. Любые дополнительные данные, требующиеся при конвертации, передаются через аргументы prArgs и nArgs.

До того, как менеджер ресурсов сможет использовать указанную процедуру-конвертор, последняя должна быть зарегистрирована. Для этого можно использовать одну из функций XtAddConverter(), XtAppAddConverter() или XtSetTypeConverter(). В коде программы конвертор должен быть зарегистрирован после инициализации Xt, но до обращения к XtGetApplicationResources().

Процедура XtAddConverter() имеет следующий прототип:

```
void XtAddConverter (String prFromType, String
psToType,
    XtConverter pConverter, XtConvertArgList
prConvArgs,
    Cardinal nNumArgs);
```

Первый и второй аргументы процедуры – это строки, задающие соответственно типы, из которого и в который надо преобразовывать. Это должны быть стандартные имена типов данных, определенные в файле «X11/StringDefs.h», или имена, определенные в файле-заголовке программы. Третий аргумент задает непосредственно саму процедуру-конвертор. Четвертый и пятый аргументы задают дополнительные данные, которые передаются конвертору и используются им. При этом prConvArgs – указатель на массив структур типа XtConvertArgRec, определяемых следующим образом:

```
typedef struct {
  XtAdddressMode address_mode;
  XtPointer    address_id;
} XtConvertArgRec, *XtConvertArgList;
```

Здесь address_mode определяет, как следует интерпретировать поле address_id. Возможные значения для поля address_mode даются перечисляемым типом XtAdddressMode:

```
typedef enum {
  XtAddress,     /* адрес                */
  XtBaseOffset,    /* смещение             */
  XtImmediate,    /* константа            */
  XtResourceString,   /* строчное имя ресурса
*/
  XtResourceQuark,      /*   внутренняя   форма
задания ресурса */
  XtWidgetBaseOffset,     /*     смещение     от
"родителя"      */
  XtProcedureArg,        /*   вызов   процедуры
*/
} XtAddressMode;
```

Здесь

XtAddress – указывает, что параметр address_id интерпретируется как адрес данных;

XtBaseOffset – address_id интерпретируется как смещение относительно базового адреса виджета;

XtImmediate – address_id является константой;

XtResourceString – address_id интерпретируется как имя ресурса, которое будет преобразовано в смещение относительно базового адреса виджета;

XtResourceQuark – address_id интерпретируется как имя ресурса, которое будет преобразовано во внутреннюю форму XtResourceString;

XtWidgetBaseOffset – аналогично XtBaseOffset, за исключением того, что смещение рассматривается относительно родителя, если последний не принадлежит подклассу класса Core;

XtProcedureArg – address_id интерпретируется как указатель на процедуру типа XtConvertArgProc, которая будет вызываться при завершении конвертации.

Процедура XtAppAddConverter() аналогична описанной выше процедуре, за исключением того, что добавлен еще один аргумент – контекст приложения. Процедура XtSetTypeConverter() также аналогична описанной процедуре, давая при этом дополнительные возможности по более эффективному конвертированию значений ресурсов.

Ниже приведен пример процедуры-конвертора, позволяющий преобразовывать строку в длинное целое.

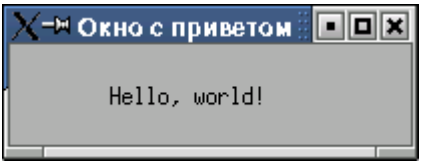
```
Void CnvStringToLong (XrmValue *prArgs,
Cardinal *pnNrgs,
          XrmValue     *prFromVal,     XrmValue
*prToVal)
{
  static long nResult;   /* result variable */
```

```
  if (*pnArgs! = 0)
    XtWarning ("String to Long conversion
needs no extra arguments!");

  if (sscanf ( (char *) prFromVal->addr,
"%ld", &nResult) = = 1)
  {
    prToVal->size = sizeof (long);
    prToVal->addr = (XtPointer) &nResult;
  }
   else
     XtStringConversionWarning ( (char *)
prFromVal->addr,
         XtRLong);
}
```

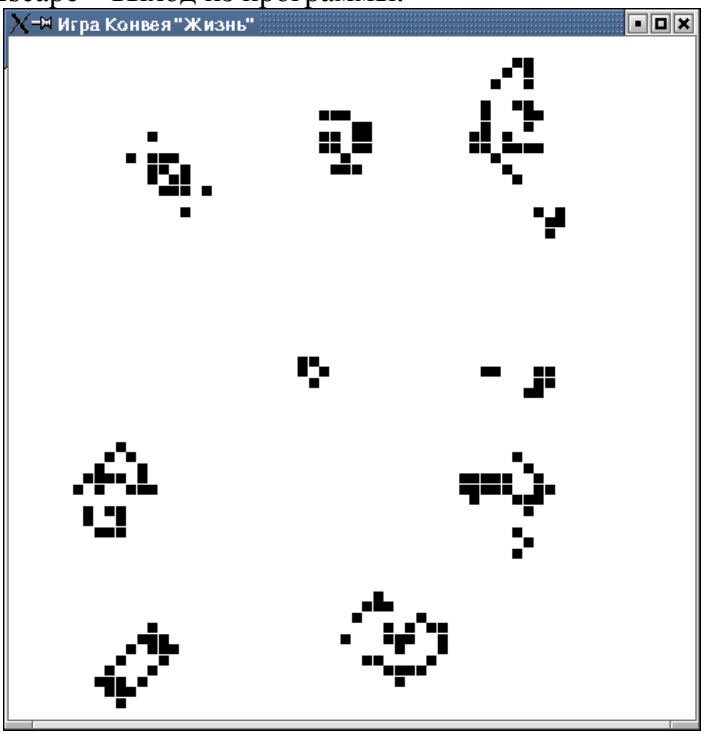
Данная процедура-конвертор использует функцию `XtWarning()` для печати предупреждающего сообщения, если число аргументов `args` больше нуля, для конвертирования строковой переменной в переменную типа `long` используется процедура `sscanf()`. Результат работы данной процедуры сохраняется в структуре `prToVal`. Процедура `XtStringConversionWarning()` предназначена для выдачи предупреждающего сообщения при неуспешном завершении `sscanf()`. Процедура имеет два аргумента типа `String`; первый – строка, задающая данные конвертируемого типа, второй – название типа данных, к которому не смогли быть преобразованы данные, определенные первым аргументом.

2.4.5. Лабораторная работа №9 «Xt и ресурсы программ»

1. Используя пример из раздела 2.4, создайте окно приветствия на основе программно определяемых ресурсов: заголовок окна, цвет фона, размеры окна.



2. Создайте программу моделирования эволюции клеточного автомата «Жизнь», ячейки которого имею два состояния: пусто и заполнено. Если рядом с пустой ячейкой три заполненных, она заполняется. Если рядом с заполненной ячейкой меньше двух или больше трех заполненных, ячейка становится пустой. Размеры модельного поля – 64x64 ячейки, вначале поле пустое. По нажатию любой кнопки мыши состояние ячейки меняется на противоположное, по нажатию пробела осуществляется один шаг эволюции, а по нажатии Escape – выход из программы.



2.4.6. Лабораторная работа №10 «Комплексные задания»

1. Составьте программу, которая принимает от пользователя параметрически заданные функции $x=x(t)$, $y=y(t)$, пределы изменения t и строит по ним график.
2. Создайте программу, отображающую свои окна на двух дисплеях, имена которых задаются в файле ресурсов.
3. Напишите программу, рисующую в дочернем окне перевернутую копию основного.
4. Разделив окно на прямоугольные области, отобразите в них все битовые изображения из каталога /usr/include/X11/bitmaps.

Литература

1. Полищук А.П., Семериков С.А. Событийно-ориентированное программирование. – Кривой Рог: КГПУ, 2001. – 336 с.
2. Робачевский А.М. Операционная система UNIX. – К.: БХВ, 2000. – 518 с.
3. David Flanagan. Volume 5: X Toolkit Intrinsics Reference Manual, 3rd Edition. – O'Reilly & Associates, 1992. – 913 p.
4. Antony Fountain, Jeremy Huxtable, Paula Ferguson and Dan Heller. Volume 6A: Motif Programming Manual for Motif 2.1, Open Source Edition. – O'Reilly & Associates, 2001. – 975 p.
5. Adrian Nye. Volume 0: X Protocol Reference Manual, 4rd Edition. – O'Reilly & Associates, 1990. – 446 p.
6. Adrian Nye. Volume 1: Xlib Programming Manual, 3rd Edition. – O'Reilly & Associates, 1992. – 821 p.
7. Adrian Nye. Volume 2: Xlib Reference Manual. – O'Reilly & Associates, 1992. – 935 p.
8. Tim O'Reilly, Adrian Nye. Volume 4M: X Toolkit Intrinsics Programming Manual. Motif Edition, 2nd Edition. – O'Reilly & Associates, 1992. – 712 p.
9. Robert W. Scheifler & James Gettys. X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD. X Version 11, Release 4. – Digital Press, 1992. – 711 p.

Учебное пособие

Александр Павлович Полищук

Сергей Алексеевич Семериков

Программирование в X Window

Подп. к печати 15.01.2003 Формат 80x84 1/16.
Бумага офсетная №1 Усл. печ. л. 10,01
Усл. кр.-от. 10,01 Уч.-изд. л. 12,14
Тираж 300 Зак. №01-1504

КГПУ, 50086, Кривой Рог-86, пр. Гагарина, 54

Криворожская городская типография
50050, Кривой Рог-50, пр. Металлургов, 28.