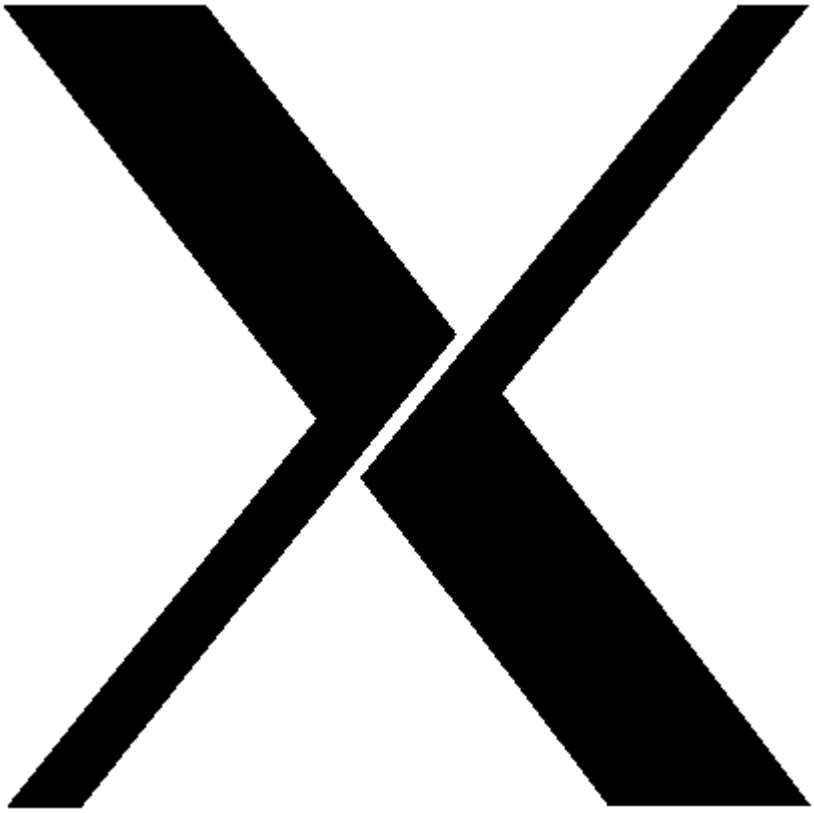
А.П. Полищук, С.А. Семериков

ПРОГРАММИРОВАНИЕ В



WINDOW

средствами Free Pascal

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.

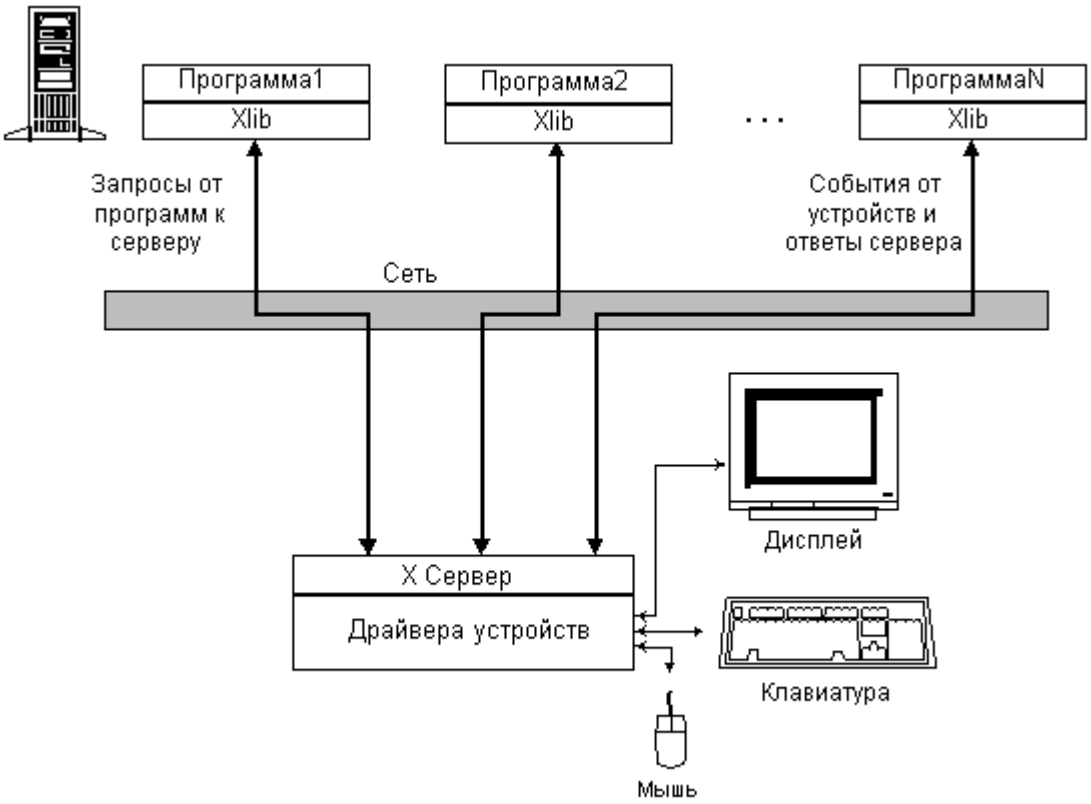


Рис. 1.1. Общая схема общения программ-клиентов и X-сервера

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.

```
uses x,xlib,x11,xutil,strings;

const
     WND_X=0;
     WND_Y=0;
     WND_WDT=100;
     WND_HGH=100;
     WND_MIN_WDT=50;
     WND_MIN_HGH=50;
     WND_BORDER_WDT=5;
     WND_TITLE='Hello!';
     WND_ICON_TITLE='Hello!';
     PRG_CLASS='Hello!';

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

procedure SetWindowManagerHints (
 prDisplay : PDisplay; (*Указатель на структуру TDisplay *)
 psPrgClass : PChar; (*Класс программы *)
 argv : PPChar;   (*Аргументы программы *)
 argc : integer;    (*Число аргументов *)
 nWnd : TWindow;    (*Идентификатор окна *)
         x,     (*Координаты левого верхнего *)
         y,     (*угла окна *)
         nWidth,
         nHeight,  (*Ширина и высота окна *)
         nMinWidth,
         nMinHeight:integer; (*Минимальные ширина и высота окна *)
 psTitle     : PChar;  (*Заголовок окна *)
 psIconTitle : PChar;    (*Заголовок пиктограммы окна *)
 nIconPixmap : TPixmap    (*Рисунок пиктограммы *)
);
var
 rSizeHints : TXSizeHints ; (*Рекомендации о размерах окна*)
 rWMHints : TXWMHints ;
 rClassHint : TXClassHint ;
 prWindowName, prIconName : TXTextProperty ;
begin

 if ( XStringListToTextProperty (@psTitle, 1, @prWindowName )=0) or
    (XStringListToTextProperty (@psIconTitle, 1, @prIconName )=0 ) then
 begin
  writeln('No memory!');
  halt(1);
 end;

rSizeHints.flags := PPosition OR PSize OR PMinSize;
rSizeHints.min_width := nMinWidth;
rSizeHints.min_height := nMinHeight;
rWMHints.flags := StateHint OR IconPixmapHint OR 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 );
end;

(*
*main - основная процедура программы
*)

//void main(int argc, char *argv[])
var
 prDisplay: PDisplay;  (* Указатель на структуру Display *)
 nScreenNum: integer;    (* Номер экрана *)
 prGC: TGC;
 rEvent: TXEvent;
 nWnd: TWindow;
begin

 (* Устанавливаем связь с сервером *)
 prDisplay := XOpenDisplay ( nil );
 if prDisplay = nil  then begin
  writeln('Can not connect to the X server!');
  halt ( 1 );
 end;

 (* Получаем номер основного экрана *)
 nScreenNum := XDefaultScreen ( prDisplay );

 (* Создаем окно *)
 nWnd := XCreateSimpleWindow ( prDisplay,
     XRootWindow ( prDisplay, nScreenNum ),
     WND_X, WND_Y, WND_WDT, WND_HGH, WND_BORDER_WDT,
     XBlackPixel ( prDisplay, nScreenNum ),
     XWhitePixel ( 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 OR KeyPressMask );

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

 (* Цикл получения и обработки событий *)
 while ( true ) do begin
  XNextEvent ( prDisplay, @rEvent );

  case ( rEvent.eventtype ) of
```

```
    Expose :
    begin
     (* Запрос на перерисовку *)
     if ( rEvent.xexpose.count <> 0 ) then
      break;

     prGC := XCreateGC ( prDisplay, nWnd, 0 , nil );

     XSetForeground ( prDisplay, prGC,
       XBlackPixel ( prDisplay, 0) );
     XDrawString ( prDisplay, nWnd, prGC, 10, 50,
       'Hello, world!', strlen ( 'Hello, world!' ) );
     XFreeGC ( prDisplay, prGC );
    end;

    KeyPress :
    begin
     (* Нажатие клавиши клавиатуры *)
     XCloseDisplay ( prDisplay );
     halt ( 0 );
    end;
  end;
 end;
end.
```

Для сборки программы используется команда:

```
fpc hello.pas
```

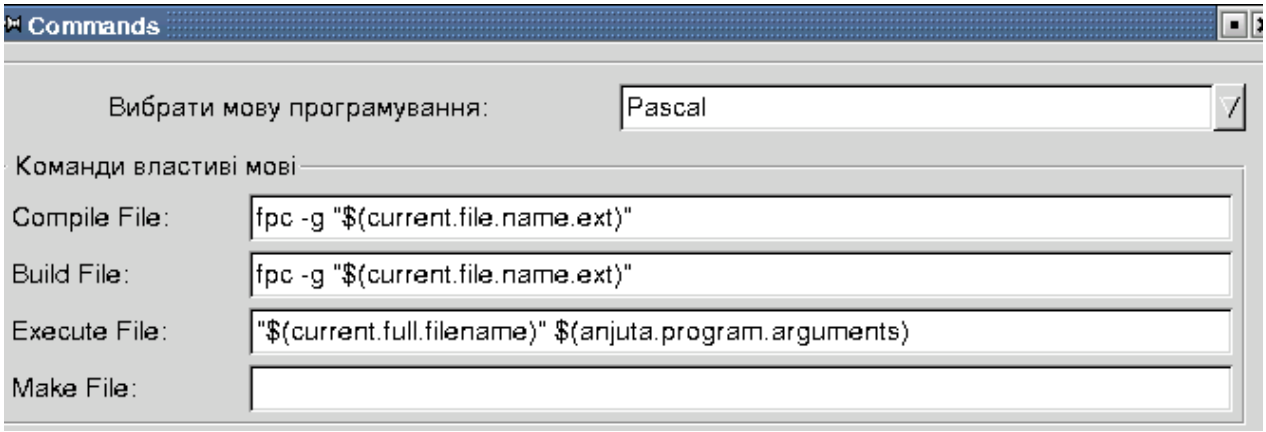
Здесь `fpc` – имя исполняемого файла компилятора. Как правило, это символическая ссылка на реальное имя компилятора (например, `ppc386`).

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 компилятора FreePascal необходимо добавить в диалог «Команды» следующие установки:



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-модулях: `Xlib`, `Xutil`, `X`, `X11`.

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 `nil`, 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 `TDisplay`. 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 `TDisplay`, которая определяется в `Xlib.pp`. Если же установить соединение не удалось, то `XOpenDisplay()` возвращает `NIL`. После успешного вызова `XOpenDisplay()` клиентской программой могут использоваться все экраны дисплея. Номер экрана возвращается функцией `XDefaultScreen()`. Доступ к полям структур `TDisplay` и `TScreen` возможен только посредством использования макроопределений и функций.

После того, как связь с сервером установлена, программа «Hello» определяет номер экрана. Для этого используется функция `XDefaultScreen()`, возвращающий номер основного экрана. Переменная `nScreenNum` может иметь значение от 0 до величины `(ScreenCount(prDisplay)-1)`. Макрос `XScreenCount()` 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 TDisplay,
          описывающую сервер *)
    XRootWindow (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 *)
    XBlackPixel ( prDisplay, nScreenNum ),
        (* цвет переднего плана окна *)
    XWhitePixel ( prDisplay, nScreenNum )
        (* цвет фона окна *)
  );
```

Для задания цветов окна используются функции `XBlackPixel()` и `XWhitePixel()`. Они возвращают значения пикселей, которые считаются на данном дисплее и экране соответствующими «черному» и «белому» цветам. Функция `XCreateSimpleWindow()` (`XCreateWindow()`) возвращает значение типа `TWindow`. 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 программе;

–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 `TXTextProperty`. Это выполняется процедурой `XStringListToTextProperty()`.

Для передачи информации о желаемой геометрии окна используется структура `TXSizeHints`.

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 or 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, NIL);
```

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 `halt()`.

`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.pp`. 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 `Function XNextEvent(prDisplay : PDisplay; prEvent : PXEvent) : longint; cdecl;external;`. 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 `TXSetWindowAttributes` и `TXWindowAttributes`. 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:

```
Function XChangeWindowAttributes(prDisplay : PDisplay;
```

```
  nWnd : TWindow; nValueMask : cardinal;
  prWinAttr : PXSetWindowAttributes) : longint; cdecl;external;
```

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

Структура `TXSetWindowAttributes` 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:

```
.......
var
   prDisplay : PDisplay;
   prWnd     : TWindow;
   rWndAttr  : TXSetWindowAttributes;
   nValMask  : cardinal;
const
   nPixmap   : TPixmap =0;
.......
nValMask := CWBorderPixmap or 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:

```
uses x,xlib,xutil,crt,dos;

(*
 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.
*)
function create_simple_window(display : PDisplay;
              width, height, x, y : integer): TWindow;
var
  screen_num, win_border_width: integer;
  win: TWindow;
begin
 screen_num := XDefaultScreen(display);
 win_border_width := 2;

(*
 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, XRootWindow(display, screen_num),
              x, y, width, height, win_border_width,
              XBlackPixel(display, screen_num),
              XWhitePixel(display, screen_num));

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

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

 create_simple_window:=win;
end;
```

```
//void main(int argc, char* argv[])
var
 display: PDisplay; (* указатель на структуру дисплея X *)
 screen_num: integer; (* количество экранов для размещения окон *)
 win: TWindow; (* дескриптор создаваемого окна *)
 display_width, display_height: word; (* высота и ширина X дисплея *)
 win_width, win_height: word; (* высота и ширина нового окна *)
 display_name: array [0..30] of Char;
 name: string;
 i: integer;
 win_attr: TXWindowAttributes;
 xx, y, scr_x, scr_y: integer;
 child_win: TWindow;
 (* переменная для хранения дескриптора родительского окна *)
 parent_win: TWindow;
 (* эта переменная будет хранить дескриптор корневого окна *)
 (* экрана, на котором отображено наше окно *)
 root_win: TWindow;
 (* эта переменная будет хранить массив дескрипторов *)
 (* дочерних окон нашего окна, *)
 child_windows: PWindow;
 (* а эта - их количество *)
 num_child_windows: integer;

begin
 name := getenv('DISPLAY'); (* имя X дисплея *)
 for i:=1 to byte(name[0]) do
   display_name[i-1]:=name[i];
 display_name[byte(name[0])]:=#0;
 (* устанавливаем соединение с X сервером *)
 display := XOpenDisplay(display_name);
 if (display = NIL) then begin
  writeln(paramstr(0),': не могу соединиться с X сервером ',
      display_name);
  halt(1);
 end;

 (* получаем геометрию экрана по умолчанию для нашего дисплея *)
 screen_num := XDefaultScreen(display);
 display_width := XDisplayWidth(display, screen_num);
 display_height := XDisplayHeight(display, screen_num);

 (* создаем новое окно в 1/9 площади экрана *)
 win_width := (display_width div 3);
 win_height := (display_height div 3);
 (* отладочная печать в стандартный вывод *)
 writeln('ширина окна - ', win_width, '; высота - ', 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 *)
 delay(3000);
```

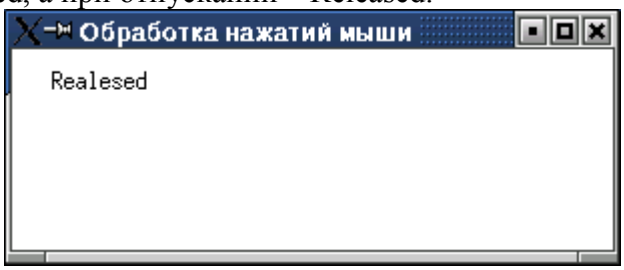
```
(* пример изменения размеров окна *)
begin

 (* в цикле уменьшаем окно *)
 for i:=0 to 39 do begin
  dec(win_width,3);
  dec(win_height,3);
  XResizeWindow(display, win, win_width, win_height);
  XFlush(display);
  delay(20);
 end;

 (* в цикле увеличиваем окно *)
 for i:=0 to 39 do begin
  inc(win_width,3);
  inc(win_height,3);
  XResizeWindow(display, win, win_width, win_height);
  XFlush(display);
  delay(20);
 end;
end;

delay(1000);

(* пример перемещения окна *)
begin

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

 xx := win_attr.x;
 y := win_attr.y;

 (* затем находим окно родителя *)
 begin

  (* 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);
 end;

 (* 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,
            xx, y,
            @scr_x, @scr_y,
            @child_win);
```

```
  (* перемещаем окно влево *)
  for i:=0 to 39 do begin
   dec(scr_x,3);
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   delay(20);
  end;

  (* перемещаем окно вниз *)
  for i:=0 to 39 do  begin
   inc(scr_y,3);
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   delay(20);
  end;

  (* перемещаем окно вправо *)
  for i:=0 to 39 do  begin
   inc(scr_x,3);
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   delay(20);
  end;

  (* перемещаем окно вверх *)
  for i:=0 to 39 do  begin
   dec(scr_y,3);
   XMoveWindow(display, win, scr_x, scr_y);
   XFlush(display);
   delay(20);
  end;
 end;

 delay(1000);

 (* пример сворачивания и восстановления окна *)
 begin
  (* сворачиваем окно *)
  XIconifyWindow(display, win, XDefaultScreen(display));
  XFlush(display);
  delay(2000);
  (* восстанавливаем окно *)
  XMapWindow(display, win);
  XFlush(display);
  delay(2000);
 end;

 XFlush(display);

 (* короткая передышка *)
 delay(2000);

 (* закрываем соединение с X сервером *)
 XCloseDisplay(display);
end.
```

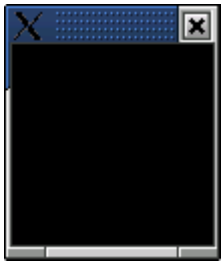
1.1.10. Лабораторная работа №1 «Основные понятия Xlib»

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 delay(), 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 `TPixmap`. 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:

```
Function XCreateGC(prDisplay : PDisplay; nDrawable : TDrawable;
   nValueMask : cardinal; prValues : PXGCValues) : TGC; cdecl;external;
```

Первый аргумент – это указатель на структуру типа `TDisplay`, 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 `TXGCValues`, определяемая следующим образом:

```
TXGCValues = record
     Xfunction : longint; { Renamed function to Xfunction }
     plane_mask : cardinal;
     foreground : cardinal;
     background : cardinal;
     line_width : longint;
     line_style : longint;
     cap_style : longint;
     join_style : longint;
     fill_style : longint;
     fill_rule : longint;
     arc_mode : longint;
     tile : TPixmap;
     stipple : TPixmap;
     ts_x_origin : longint;
     ts_y_origin : longint;
     font : TFont;
     subwindow_mode : longint;
     graphics_exposures : TBool;
     clip_x_origin : longint;
     clip_y_origin : longint;
     clip_mask : TPixmap;
     dash_offset : longint;
     dashes : char;
  end;
PXGCValues = ^TXGCValues;
```

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 модуле `X`. Если бит установлен, то значение соответствующего атрибута должно быть взято из переданной `XCreateGC()` структуры `TXGCValues`. 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.

```
. . . . . .
var
  prGC : TGC;
  rValues : TXGCValues;
  prDisplay : PDisplay;
  nScreenNum : integer;
. . . . . .
  rValues.foreground := XBlackPixel (prDisplay, nScreenNum);
  rValues.background := XWhitePixel (prDisplay, nScreenNum);
. . . . . .
  prGC := XCreateGC (prDisplay, XRootWindow (prDisplay, nScreenNum),
  (GCForeground OE 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 `Xfunction` в структуре `TXGCValues`) 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 `Xfunction` 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 функции `XDisplayPlanes()`. 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.

1. 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.

2. 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.

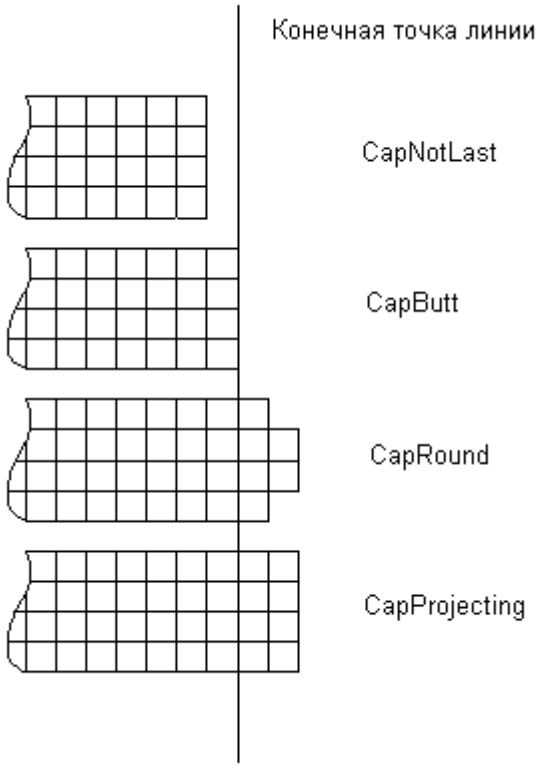


Рис. 1.4. Значения параметра `cap_style` графического контекста

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.

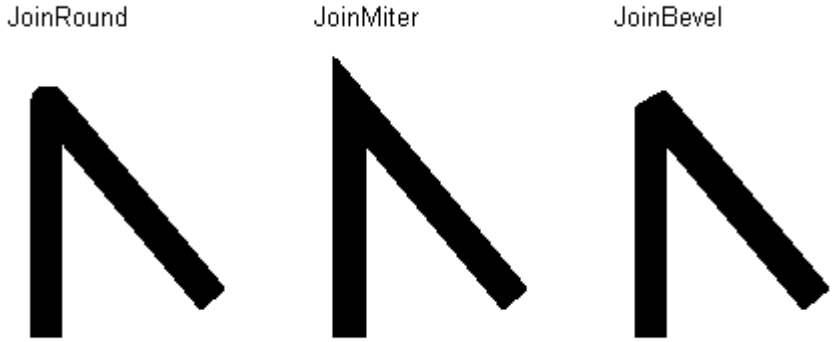


Рис. 1.5. Значения параметра `join_style` графического контекста

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 TXTextItem.

Процедура XDrawText16() использует структуру TXDrawText16.

Поле font в приведенных структурах (TXTextItem и TXDrawText16) 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 TXCharStruct. Сам шрифт описывается структурой TXFontStruct.

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!».

```
var
  prDisplay : PDisplay;
  prGC : TGC;
  nWnd : TWindow;
  prFontInfo : PXFontStruct;
. . . . . .
(* Загружаем шрифт *)
prFontInfo := XLoadQueryFont(prDisplay, "*-courier-*" );
if ( prFontInfo = NIL) then
begin
  writeln('Font not found!');
  halt(1);
end;
. . . . . .
(* Создаем GC и рисуем строку *)
prGC := XCreateGC(prDisplay, nWnd, 0, NIL);
XSetForeground (prDisplay, prGC, XBlackPixel(prDisplay, 0));
XSetFont (prDisplay, prGC, prFontInfo^.fid);
XDrawString (prDisplay, nWnd, prGC, 10, 50, 'Hello, world!',
      strlen ('Hello, world!') );
XFreeGC (prDisplay, prGC);
. . . . . .
(* "Сбрасываем" шрифт *)
XFreeFont (prDisplay, prFontInfo);
. . . . . .
```

Для 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, определена функция `XDefaultColormap`, возвращающая 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).

```
var
  screen_colormap : TColormap;

  screen_colormap := XDefaultColormap(display, XDefaultScreen(display));
```

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

```
var
  default_visual : PVisual;
  my_colormap : TColormap;

  default_visual := XDefaultVisual(display, XDefaultScreen(display));
  (* Создаем новую палитру, количество цветов в которой       *)
  (* определяется количеством цветов, поддерживаемых данным экраном. *)
  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 `TXColor`, содержащую следующие поля:

`pixel : cardinal` – индекс палитры, используемый, для рисования данным цветом.

`red : word` – красная составляющая RGB-значения цвета.

`green : word` – зеленая составляющая RGB-значения цвета.

`blue : word` – синяя составляющая RGB-значения цвета.

Пример использования этих функций:

```
var
  (* Эта структура будет содержать выделенные цветовые данные *)
  system_color_1, system_color_2 : TXColor;
  (* Эта структура будет содержать точные RGB-значения именованных *)
```

```
  (* цветов, которые могут отличаться от выделенных        *)
  exact_color : TXColor;
  rc : TStatus;

  (* Выделяем "красный" элемент палитры *)
  rc := XAllocNamedColor(display,
              screen_colormap,
              'red',
              @system_color_1,
              @exact_color);
  (* проверяем успешность выделения *)
  if (rc = 0) then begin
    writeln('XAllocNamedColor - выделить "красный" цвет не удалось.');
  end
  else begin
    writeln('Элемент палитры "красный" выделен как (',
            system_color_1.red, ', ', system_color_1.green, ', ',
            system_color_1.blue, ') в RGB-значениях.');
  end;

  (* выделяем цвет со значениями (30000, 10000, 0) в RGB. *)
  system_color_2.red := 30000;
  system_color_2.green := 10000;
  system_color_2.blue := 0;
  rc := XAllocColor(display,
            screen_colormap,
            @system_color_2);
  (* проверяем успешность выделения *)
  if (rc = 0) then begin
     writeln('XAllocColor - цвет (30000,10000,0) выделить не удалось.');
  end
  else begin
    (* что-то делаем с выделенным цветом... *)
  .
  .
  end;
```

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 X 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 `TDrawable`, не `TWindow`, 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:

```
var
  (* эта переменная будет содержать дескриптор новой пиксельной карты *)
  bitmap : TPixmap;

  (* эти переменные будут содержать размер загружаемой битовой карты *)
  bitmap_width, bitmap_height : word;

  (* эти переменные будут содержать положение горячей точки *)
  ((* загружаемой битовой карты                *)
  hotspot_x, hotspot_y : integer;

  (* эта переменная будет содержать дескриптор корневого окна экрана, *)
  (* для которого мы хотим создать пиксельную карту          *)
  root_win : TWindow;
  rc : longint;

  root_win := XDefaultRootWindow(display);

  (* загружаем битовую карту из файла "icon.bmp", создаем *)
  (* пиксельную карту, содержащую свои данные в сервере, *)
  (* и сохраняем ее дескриптор в переменной bitmap    *)
  rc := XReadBitmapFile(display, root_win,
             'icon.bmp',
             @bitmap_width, @bitmap_height,
             @bitmap,
             @hotspot_x, @hotspot_y);
  (* проверяем, удалось ли создать пиксельную карту *)
  case (rc) of
    BitmapOpenFailed:
      writeln('XReadBitmapFile - не могу открыть файл "icon.bmp"');
    BitmapFileInvalid:
      writeln('XReadBitmapFile - файл "icon.bmp" не содержит корректного
битового изображения.');
    BitmapNoMemory:
      writeln('XReadBitmapFile - не хватает памяти.');
    BitmapSuccess:
      (* битовая карта успешно загружена - что-то делаем с ней... *)
```

```
    .
    .
  end;
```

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 хотим 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.

```
var
  (* эта переменная будет содержать дескриптор новой пиксельной карты *)
  pixmap : TPixmap;

  (* эта переменная будет содержать дескриптор корневого окна экрана, *)
  (* для которого мы хотим создать пиксельную карту          *)
  root_win : TWindow;

(* эта переменная будет содержать глубину цвета создаваемой *)
(* пиксельной карты - количество бит, используемых для   *)
(* представления индекса цвета в палитре (количество цветов *)
(* равно степени двойки глубины)              *)
  depth : longint;

  root_win := XDefaultRootWindow(display);
  depth := XDefaultDepth(display, XDefaultScreen(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 хотим 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 `/usr/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. Из файла `/usr/include/X11/cursorfont.h` узнаем номера необходимых идентификаторов и опеределяем их:

```
const
  XC_watch=150;

var
  (* эта переменная содержит дескриптор создаваемого курсора *)
  watch_cursor : TCursor;
```

```
  (* 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.

```
var
  (* эта переменная содержит дескриптор создаваемого курсора *)
  icon_cursor : TCursor;

  (* вначале необходимо определить основной и фоновый цвета курсора *)
  cursor_fg, cursor_bg : TXColor;

  screen_colormap : TColormap;
  rc : TStatus;

  (* получаем доступ к палитре экрана по умолчанию *)
  screen_colormap := XDefaultColormap(display, XDefaultScreen(display));

  (* выделяем черный и белый цвета *)
  rc := XAllocNamedColor(display,
              screen_colormap,
              'black',
              @cursor_fg,
              @cursor_fg);
  if (rc = 0) then begin
    writeln('XAllocNamedColor - невозможно распределить цвет "black"');
    halt(1);
  end;
  rc := XAllocNamedColor(display,
              screen_colormap,
              'white',
              @cursor_bg,
              @cursor_bg);
  if (rc = 0) then begin
    writeln('XAllocNamedColor - невозможно распределить цвет "white"');
    halt(1);
  end;

  (* 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

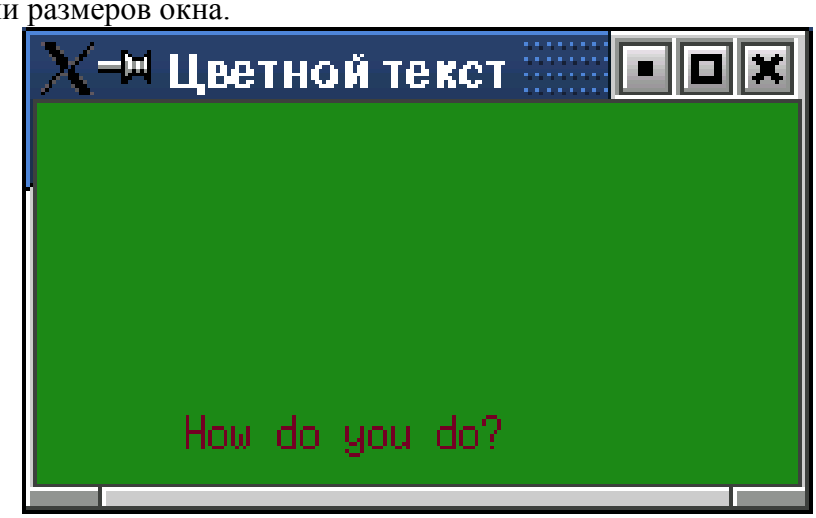
```
XFreeCursor():
XFreeCursor(display, icon_cursor);
```

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.

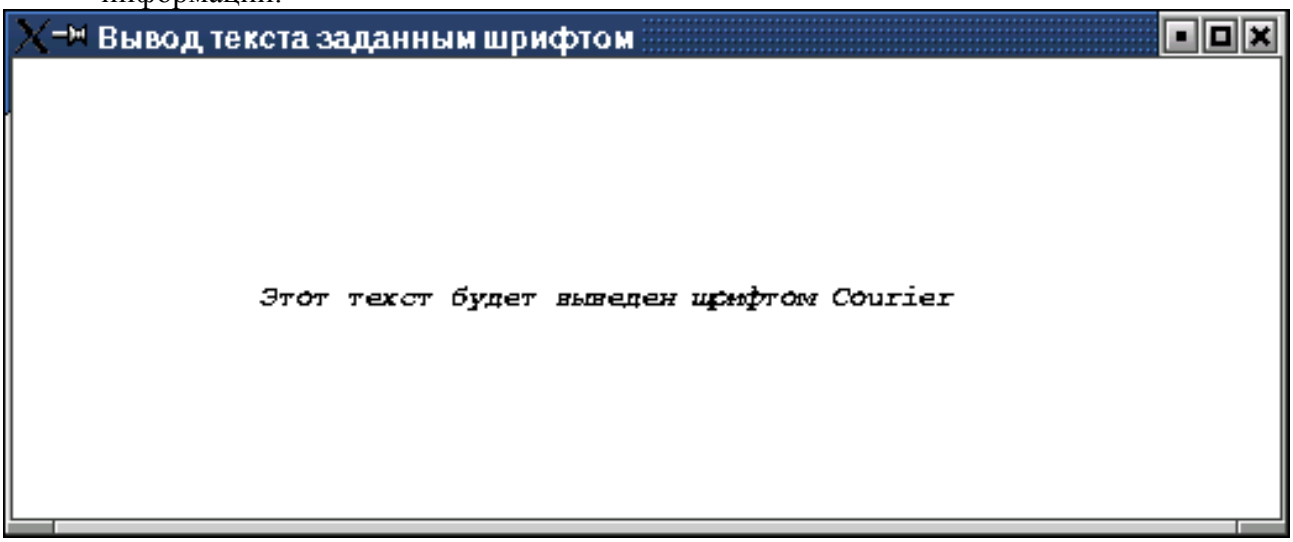
```
(* прикрепить курсор к окну *)
XDefineCursor(display, win, icon_cursor);

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

1.2.7. Лабораторная работа №2 «Текст и графика»

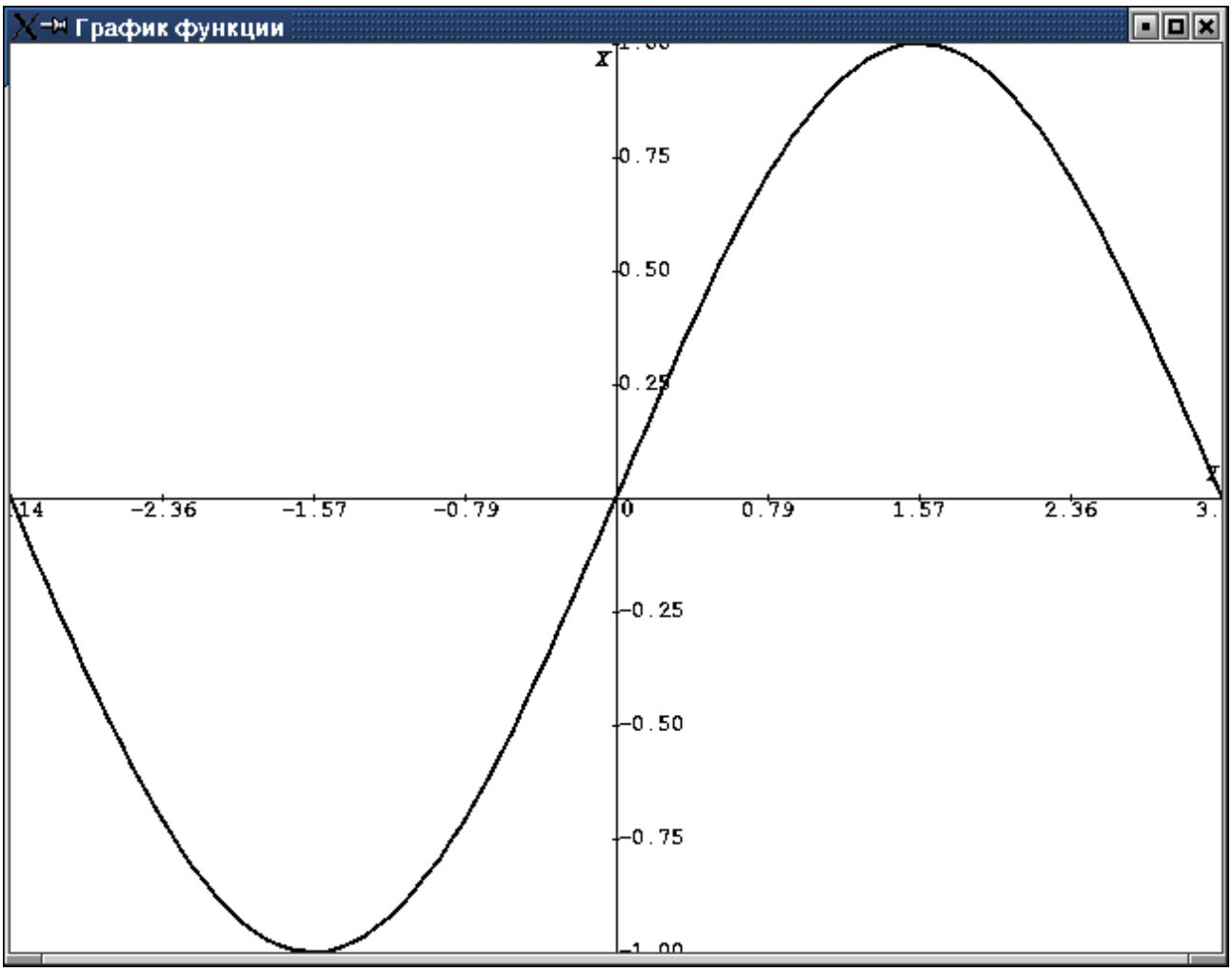
1. Напишите программу, выводящую текстовое сообщение в произвольную позицию (в пределах окна) произвольным цветом. Цвет и координаты должны меняться при изменении размеров окна.



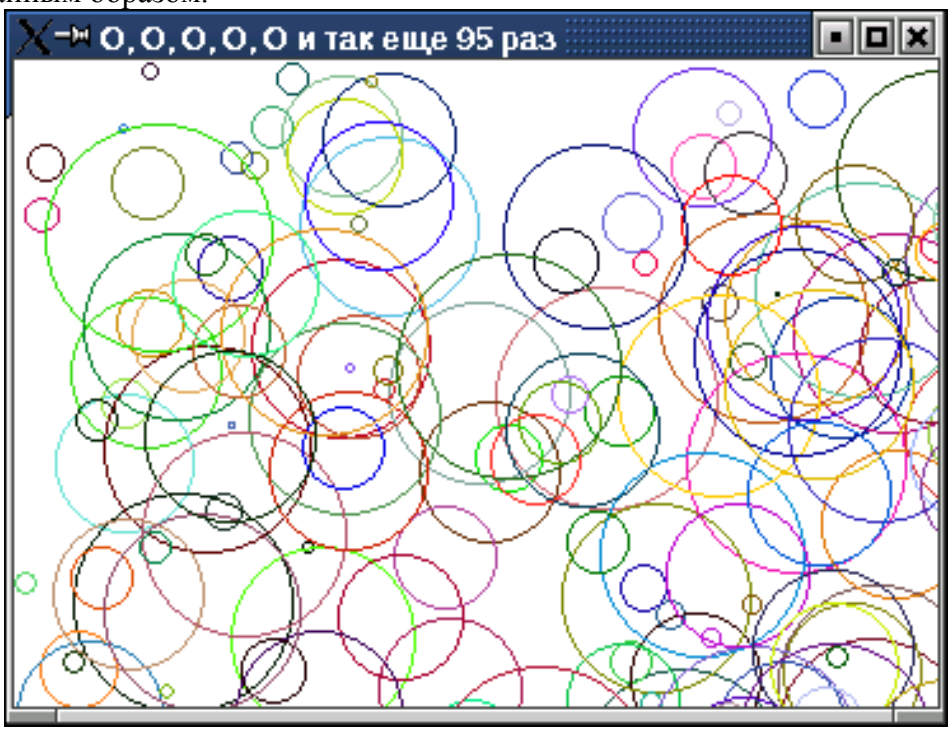
2. Составьте программу, принимающую со стандартного ввода маску шрифта, выводимую строку, координаты *x*, *y* и отображающую окно с текстом согласно введенной информации.



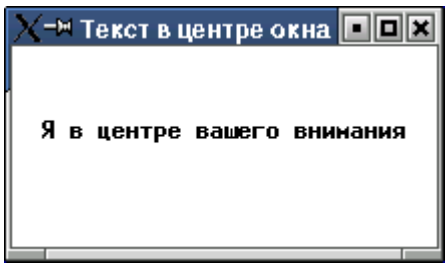
3. Нарисуйте в окне график функции sin(*x*) на отрезке [–π; π]. Оси подпишите курсивом, метки по осям – обычным шрифтом, начало координат (0) выделите жирным шрифтом.



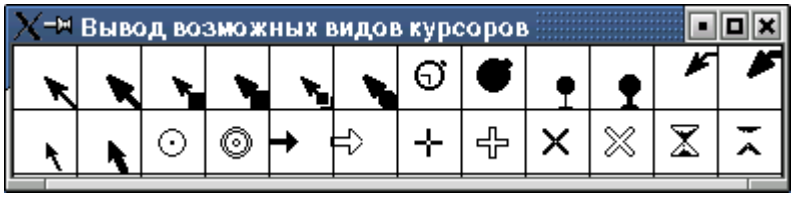
4.Нарисуйте в окне 100 окружностей. Цвет, координаты центра и радиус выбирать случайным образом.



5.Используя StructureNotifyMask и русский шрифт, модифицируйте программу из первого задания лабораторной работы №1 таким образом, чтобы сообщение всегда отображалось в центре окна.



6.Составьте программу, выводящую в окно все символы стандартного курсорного шрифта.



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 `TXKeyEvent`. 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 `TXKeyEvent`, 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 `/usr/include/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 `/usr/include/X11/keysym.h`.

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

```
Function XKeycodeToKeysym(prDisplay : PDisplay;nKeycode : TKeyCode;
                          nIndex : longint) : TKeySym; cdecl;external;
```

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`.

```
........
var
  nF2Sym, nF3Sym : TKeysym;
  nF2Keycode : TKeyCode;
  prDisplay : PDisplay;
........
  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.

```
........
var
  prDisplay : PDisplay;
  nScreenNum : integer;
  prGC : TGC;
  rEvent : TXEvent;
  nWnd : TWindow;
  sKeyStr : array [0..19] of char;
  nKeySym : TKeySym;
  naModList : array [0..1] of TKeySym;
  n : integer;
  r: char;

const
  XK_Control_L=$FFE3; (* Left control *)
  XK_Shift_L=$FFE1;   (* Left shift *)
  XK_F1=$FFBE;
  XK_F2=$FFBF;
  XK_F3=$FFC0;
  XK_F4=$FFC1;
  XK_F5=$FFC2;
  XK_F6=$FFC3;


  (* 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 true do begin
    XNextEvent (prDisplay, @rEvent);
    case (rEvent.eventtype) of
       ......
      KeyPress :
      begin
        (* Очищаем строку *)
        for n:=0 to 19 do
          sKeyStr[n]:=#0;

        (* Получаем строку, соответствующую событию *)
        XLookupString (@rEvent.xkey, sKeyStr, 20, @nKeySym, NIL);
        if ( strcomp (sKeyStr, 'EXIT')=0 ) then
        begin
          XFreeGC (prDisplay, prGC);
          XCloseDisplay (prDisplay);
          halt (0);
        end;

        case nKeySym of
          XK_F1: r:='1';
          XK_F2: r:='2';
          XK_F3: r:='3';
          XK_F4: r:='4';
```

```
          XK_F5: r:='5';
          else r:='0';
        end;

        if (n<>0) then begin
          sKeyStr[0]:='F';
          sKeyStr[1]:=r;
          sKeyStr[2]:=#0;
          strcat(sKeyStr, ' pressed.');
          XClearWindow (prDisplay, nWnd);
          XDrawString (prDisplay, nWnd, prGC, 10, 50,
                       sKeyStr, strlen (sKeyStr));
        end;
       end;
  end;
end;
........
```

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

```
TXKeyboardState = record
     key_click_percent : longint;
     bell_percent : longint;
     bell_pitch : cardinal;
     bell_duration : cardinal;
     led_mask : cardinal;
     global_auto_repeat : longint;
     auto_repeats : array[0..(32)-1] of char;
  end;
PXKeyboardState = ^TXKeyboardState;
```

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 `TXKeyboardControl`, определяемую следующим образом:

```
TXKeyboardControl = record
     key_click_percent : longint;
     bell_percent : longint;
     bell_pitch : longint;
     bell_duration : longint;
```

```
      led : longint;
      led_mode : longint;
      key : longint;
      auto_repeat_mode : longint;
  end;
PXKeyboardControl = ^TXKeyboardControl;
```

Первые четыре поля совпадают с аналогичными полями структуры `TXKeyboardState`. 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 `TXEvent` и содержит, в частности, такие поля:

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

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

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

`time : TTime` – 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.

```
. . . . . .
ButtonPress:
```

```
begin
  (* 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       *)
  case (an_event.xbutton.button) of
    Button1:
      (* рисуем пиксель в позиции мыши *)
      XDrawPoint(display, the_win, gc_draw, x, y);
    Button2:
      (* стираем пиксель в позиции мыши *)
      XDrawPoint(display, the_win, gc_erase, x, y);
    else (* возможно, третья кнопка - игнорируем *)
      ;
  end;
end;
. . . . . .
```

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 `TXEvent` и содержит, в частности, такие поля:

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

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

`state : cardinal` – 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 : TTime` – 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.

```
. . . . . .
MotionNotify:
begin
  (* 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 AND Button1Mask) then begin
    XDrawPoint(display, the_win, gc_draw, x, y);
  end;
end;
. . . . . .
```

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 `TXEvent` и содержит, в частности, такие поля:

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

`subwindow : TWindow` – 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.

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

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

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

`state : cardinal` – 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).

focus : TBool – 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

```
Function XGrabPointer(prDisplay : PDisplay; nGrabWnd : TWindow;
                      nOwnerEvents : TBool; nEventMask : cardinal;
                      nPointerMode : longint; nKeyboardMode : longint;
                      nConfineTo : TWindow; nCursor : TCursor;
                      nTime : TTime) : longint; cdecl;external;
```

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.

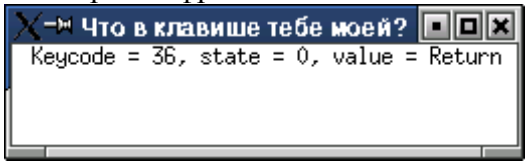
Если nOwnerEvents равен False, то параметр nEventMask указывает, какие события следует передавать окну nGrabWnd.

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().

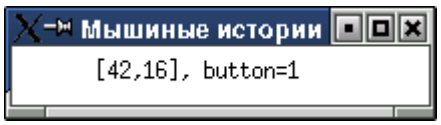
Параметр nCursor задает форму курсора во время того, как мышь захвачена. Аргумент nTime указывает, когда система должна активизировать режим захвата.

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

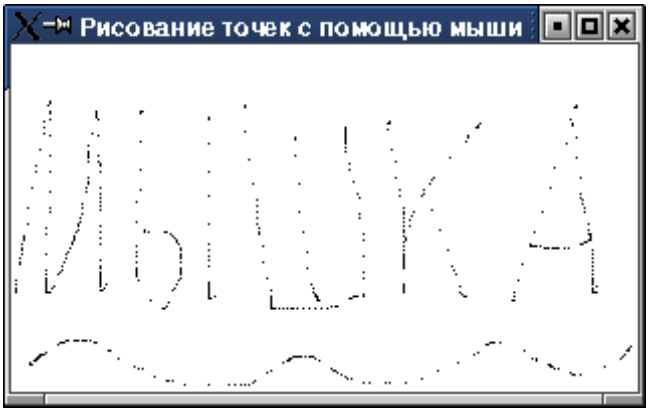
1. Используя функции XKeysymToString() и XKeycodeToKeysym(), напишите программу, которая реагирует на нажатие клавиш в окне выдачей в него кода символа, состояния модификаторов и символьной расшифровки нажатой клавиши.



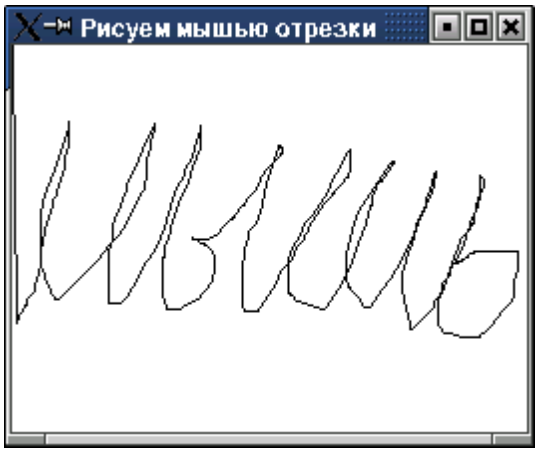
2. Напишите программу, определяющую координаты мыши в момент нажатия кнопки и печатающую в позицию мышиного курсора координаты мыши и номер нажатой кнопки.



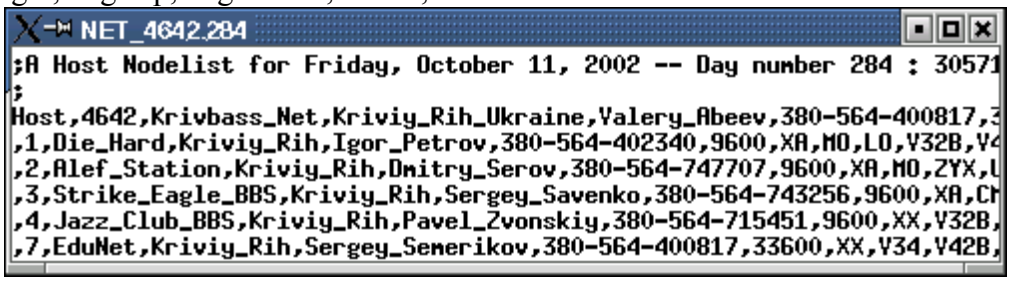
3. Модифицируйте предыдущую программу для рисования точек в местах нажатий мыши и при ее движении с нажатой кнопкой.



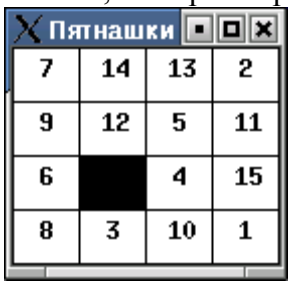
4. Модифицируйте предыдущую программу для рисования отрезков между нажатиями мыши и при ее движении с нажатой кнопкой.



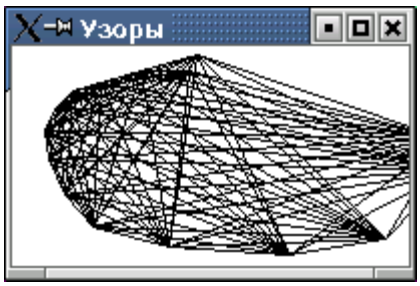
5. Создайте программу, отображающую в окне содержимое текстового файла, имя которого задается в командной строке. Для скроллинга текста используйте клавиши Up, Down, Left, Right, PageUp, PageDown, Home, End.



6. Составьте программу игры «Пятнашки», выбирая перемещаемую ячейку мышью.



7. 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.



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:

```
  . . . . . .
prDisplay : PDisplay;
prGC : TGC;
nWnd : TWindow;
psString : PChar;
. . . . . .
(* Устанавливаем связь с сервером, получаем номер экрана. . .*)
. . . . . .
(* Выбираем события, обрабатываемые программой *)
XSelectInput (prDisplay, nWnd, ExposureMask OR 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.

```
Procedure XrmParseCommand(
    prDB : TXrmDatabase   { database };
    prOptRec : TXrmOptionDescList     { table };
    nOptRecNum : integer                        { table_count };
    psProgName : pchar  { name };
    argc : Pointer                { argc_in_out };
    argv : ppchar         { argv_in_out }
);cdecl;external;
```

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.

```
. . . . . .
const
  rOptRec : array [0..2] of TXrmOptionDescRec = (
   ( '-bg', '*background', XrmoptionSepArg, 'Red'  ),
   ( '-fg', '*foreground', XrmoptionSepArg, 'White' ),
   ( '-xrm', NIL,     XrmoptionResArg, NIL  ),
);
var
  rDB : TXrmDatabase;
. . . . . .
//void main (int argc, char **argv)
begin
  . . . . . .
  XrmInitialize( );
  XrmParseCommand (rDB, rOptRec,
             sizeof (rOptRec) / sizeof (rOptRec[0]),
             argv[0], @argc, argv);
  . . . . . .
end.
```

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

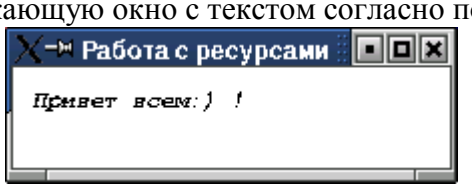
```
Function XrmGetResource(
    prDB : TXrmDatabase           { database };
    psResName : pchar  { str_name };
    psResClass : pchar  { str_class };
    psResType : ppchar         { str_type_return };
    psResVal : PXrmValue              { value_return }
) : Tbool;cdecl;external;
```

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. Полный список реализуемых X протоколом атомов можно найти в файле `/usr/include/X11/Xatom.h`.

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 `TXTextProperty`. Процедура `XStringListToTextProperty()` переводит список строк в набор данных типа `TXTextProperty`:

```
(* Эта переменная будет хранить созданное свойство. *)
var
  window_title_property : TXTextProperty ;
  rc : TStatus;

  (* Строка, преобразуемая в свойство. *)
const
  window_title : PChar = 'hello, world';

(* перевод строки в свойство X. *)
rc := XStringListToTextProperty(@window_title,
                                1,
                                @window_title_property);
(* проверка успешности преобразования. *)
if (rc = 0) then begin
  writeln( 'XStringListToTextProperty - нет памяти');
  halt(1);
end;
```

`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 `TXTextProperty`. 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 TXTextProperty. 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 TXSizeHints. 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. *)
var
  win_size_hints : PXSizeHints;

win_size_hints := XAllocSizeHints();
if (win_size_hints=nil) then begin
  writeln('XAllocSizeHints - нет памяти');
  halt(1);
end;

(* Инициализация структуры *)
(* Вначале укажем, что передаются пожелания о размерах: *)
(* установим минимальный и начальный размеры. *)
win_size_hints^.flags := PSize OR PMinSize;
(* Затем указываем требуемые границы.              *)
(* в нашем случае - создаем окно минимальным размером 300x200 *)
(* пикселей 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 TXWMHints. Для задания свойства используется процедура XSetWMHints(). Структура типа XWMHints, передаваемая функции XSetWMHints(), должна быть подготовлена с помощью XAllocWMHints():

```
var
  win_hints : PXWMHints;
  icon_pixmap : TPixmap;

const
  icon_bitmap_width=20;
  icon_bitmap_height=20;
  (* Определим битовое изображение в формате X - *)
```

```
  (* оно может быть создано программой xpaint  *)
  icon_bitmap_bits : array [0..59] of byte = (
 $60, $00, $01, $b0, $00, $07, $0c, $03, $00, $04, $04, $00,
 $c2, $18, $00, $03, $30, $00, $01, $60, $00, $f1, $df, $00,
 $c1, $f0, $01, $82, $01, $00, $02, $03, $00, $02, $0c, $00,
 $02, $38, $00, $04, $60, $00, $04, $e0, $00, $04, $38, $00,
 $84, $06, $00, $14, $14, $00, $0c, $34, $00, $00, $00, $00
);


win_hints := XAllocWMHints();
if (win_hints=nil) then begin
  writeln('XAllocWMHints - нет памяти');
  halt(1);
end;

(* установим пожелания о состоянии окна, позиции его иконки *)
(* и ее виде                                 *)
win_hints^.flags = StateHint OR IconPositionHint OR IconPixmapHint;

(* Загрузим заданное битовое изображение *)
(* и создадим из него пиксельную карту X. *)
Pixmap icon_pixmap = XCreateBitmapFromData(display,
                                           win,
                                           PChar(icon_bitmap_bits),
                                           icon_bitmap_width,
                                           icon_bitmap_height);
if (icon_pixmap=nil) then begin
  writeln('XCreateBitmapFromData: ошибка создания пиксмапа');
  halt(1);
end;


(* 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 `TXClassHints`. 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.

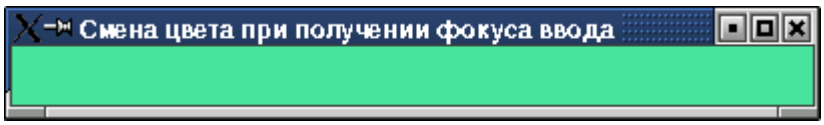
```
. . . . . .
var
  prDisplay : PDisplay;
  nScreenNum : integer;
  prGC : TGC;
  rEvent : TXEvent;
  nWnd : TWindow;
  pnProtocol : array [0..1] of TAtom;
  nWMProtocols : TAtom;

(*
 *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 true do
begin
  XNextEvent (prDisplay, @rEvent);

  case (rEvent.type) of
    . . . . . .
    ClientMessage :
    begin
      if (rEvent.xclient.message_type = nWMProtocols) then
      begin
        if (rEvent.xclient.data.l[0] = pnProtocol[0]) then
          writeln('Receiving the input focus.')
        else
          if (rEvent.xclient.data.l[0] = pnProtocol[1]) then
          begin
            XCloseDisplay (prDisplay);
            halt(0);
          end;
      end;
    end;
    . . . . . .
  end;
end;
. . . . . .
```

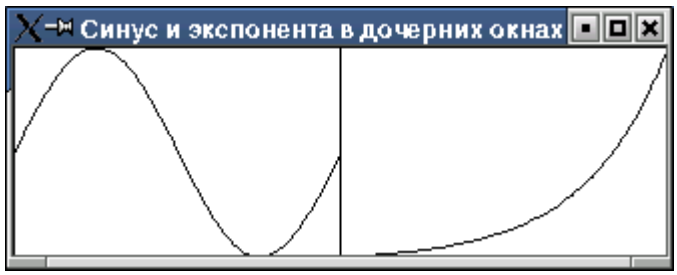
Заказывается реакция на два события: получение фокуса ввода (`WM_TAKE_FOCUS`) и завершение программы (`WM_SAVE_YOURSELF`). Когда сервер посылает событие первого типа, задача печатает соответствующее сообщение на устройства вывода. При приходе события второго типа, программа закрывает связь с сервером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

1.5.3. Лабораторная работа №5 «Межклиентское взаимодействие»

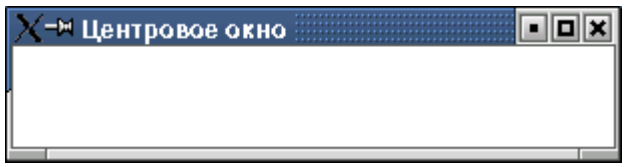
1. Составьте программу, которая при получении фокуса ввода перекрашивает свое окно в другой цвет.



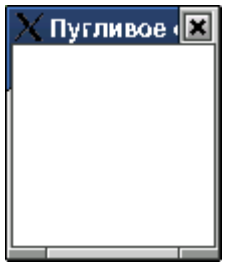
2. Составьте программу, порождающую два расположенных рядом дочерних окна, в которых отображаются графики функций $\sin(x)$ на отрезке $[0; 2\pi]$ и $\exp(x)$ на отрезке $[-2; 2]$. Графики масштабировать по размеру окон.



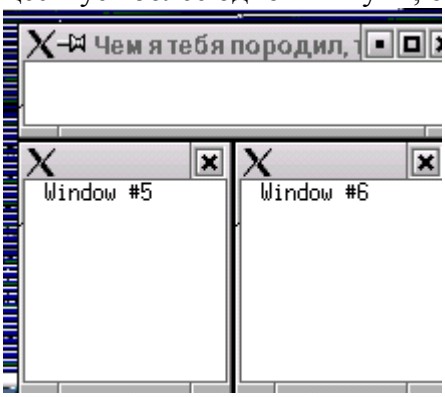
3. Создайте окно, изменяющее свои размеры таким образом, чтобы мышь всегда была в его центре.



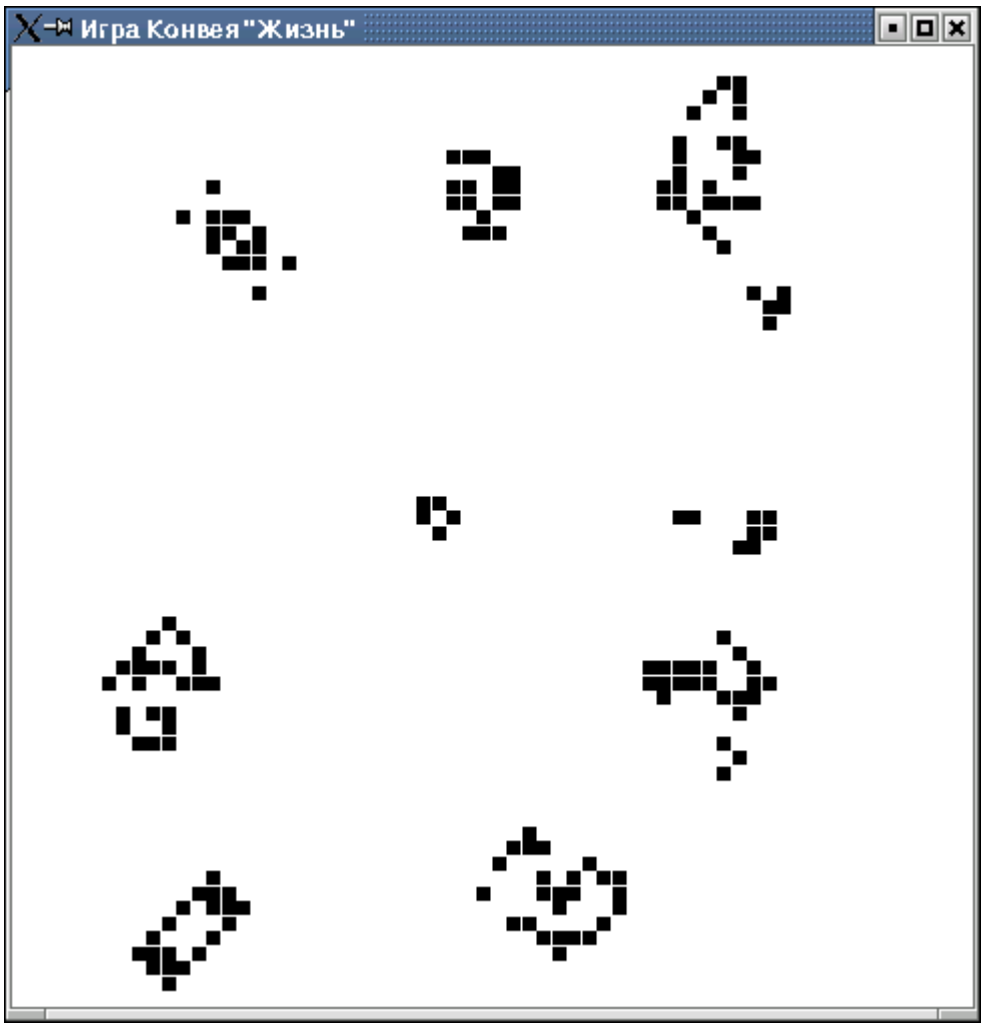
4. Создайте окно, «убегающее» от указателя мыши.



5. 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.



6. 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.


Игра Конвея "Жизнь"

Литература

1. Полищук А.П., Семериков С.А. Программированиев X Window. – Кривой Рог: Издательский отдел КГПУ, 2003. – 192 с.
2. Полищук А.П., Семериков С.А. Событийно-ориентированное программирование. – Кривой Рог: КГПУ, 2001. – 336 с.
3. Робачевский А.М. Операционная система UNIX. – К.: БХВ, 2000. – 518 с.
4. Adrian Nye. Volume 0: X Protocol Reference Manual, 4rd Edition. – O'Reilly & Associates, 1990. – 446 p.
5. Adrian Nye. Volume 1: Xlib Programming Manual, 3rd Edition. – O'Reilly & Associates, 1992. – 821 p.
6. Adrian Nye. Volume 2: Xlib Reference Manual. – O'Reilly & Associates, 1992. – 935 p.
7. 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.