OpenGUI
The Fast Graphics Library.

The lightweight multi-platform library for GUI applications and games development

with GNU or Watcom C++

on MS-DOS, QNX and LINUX.

revision: 2.12 (2/23/2000)

1.1 The overview

OpenGUI is a very fast multi-platform 32-bit graphics library for **MS-DOS** (DPMI client), **QNX** and **LINUX**. It can be used to create graphics applications and games for these **Operating Systems**. The can be used with **GNU C++** (from ver. 2.7.2) or **Watcom C++** (from 9.5) and **NASM** assembler (0.97). The library is very stable and is ready for programs that require real-time drawing.

Features:

- **ultra-fast (asm kernel & MMX support)**
- **support for resolutions ranging from 320x200 to 1600x1200**
- **windowing system...**
- **object-oriented multi-platform API (DJGPP, WATCOM, GCC, LINUX, QNX)**
- **full application development environment (configuration file, file dialogs, etc.)**
- **professionally tested & actively used for one year (see examples)**
- **a tool for interactive code generation (draw & run)**
- **and much more**

The library consists of three layers. The first layer is a hand-coded and fast assembler kernel. This layer does the biggest piece of hard work. The second layer implements the API for drawing graphics primitives like lines, rectangles, circles etc. This layer is comparable to Borland BGI API. The third layer is written with C++ and offers a complete object set for the GUI developer. The third layer implements objects like input windows, buttons, menus, bitmaps etc, with addition of integrated mouse & keyboards support. To create a simple demonstration that draws on the screen a window captioned "HELLO WORLD", a button labelled "Finish" and shows a mouse pointer only a few lines (see below) are required.

```
#include <fastgl.h>
int main(int argc, char **argv )
{
        App MyApp(3,argc,argv,CDARK,APP_ENABLEALTX);
        Window *MyWindow = new Window(&MyWindow, X_width/2-100,Y_width/2-50,200,100, "Window");
        set_colors(CRED,PM);
        MyWindow.printf(54,15,"Hello World!\n");
        MyWindow.AddPushButton(60,45,80,24,"Finish",'F', App::AppDone);
        MyApp.Run();
        return 0;
}
```

1.2 The installation

To install the library, unpack **OpenGUI*.tgz and examples.tgz** to your hard disk (using tar xvzf FILENAME command). You will get the directory **OpenGUI** containing two subdirectories: **src** and **examples**. If you are compiling the library on LINUX, you will require **root** permissions. To build the library, cd to the **src** directory and run **make –f makefile.XXX** where XXX should match your OS of choice – "**lin**" for LINUX, **".dj**" for djgpp etc. If the compilation succeeds, it will also install libraries and headers to their appropriate locations. Next, you may wish to compile examples (in a similar way – in the **examples** dir).

The installation consists of three header files (**base.h**, **fastgl.h** and **widgets.h**) and library files (**libfgl.a**, **fgl.lib** or **libfgl.so**, depending on the platform) that will be linked with your programs. During the installation (which is performed automatically during the make) include files are copied to standard include directory (/usr/include, /gcc/include etc.), and library files to the library directory (/usr/lib, /gcc/lib etc.). For LINUX, a dynamic version of the library is built as well. Static library should be used for debugging purposes and the dynamic library can be used for the final version of your software. Use of dynamic library requires the code to be compiled with –FPIC flag, forcing position independent code generation.

NOTE: the library was tested under LINUX, with EGCS, PGCC and GCC compilers.

1.3 Troubleshooting

1.3.1 LINUX: the linker - undefined reference to `vga_setmode'

You forgot to link with the svgalib library. Add parameter-*lvga* into the Makefile for the linker

1.3.2 LINUX: my program crashes immediatelly at the start (shared linking)

You forgot to use the *–fPIC* compiler switch that is needed to create a dynamic executable (it forces the compiler to generate position independent code). You should not use the *–shared* switch at the same time! (I don't know why, but this will do the right thing).

1.3.3 LINUX: make produces absurd errors (I don't know how to reproduce this)

The GNU make utility doesn't work with makefile that contain **CRLF** instead **LF** only at the end of each line. You can convert the file using utility **textto** with parameter *–l*. The source code for this utility is included in the *src* directory.

1.3.4 LINUX: the mouse does not work (or works partially)

You should edit the MOUSE section in */etc/vga/libvga.config*,. If it does not solve the problem, try to check the permissions for the /dev/mouse or /dev/ttyS0 and adjust them.

1.3.5 QNX: my program crashes with signal SIGFPE

This problem occurs at the start-up, at the switch to graphics mode. The solution is to start the graphics daemon that emulates **int 10.** Run it by entering the command "int10 &". The best solution is to insert this line into the config file */etc/config/sysconfig.x*, where 'x' is the number of the node in the net.

1.3.6 QNX: the linker produces undefined symbols

Check that you use explicit compiler switch *–5s* (stack calling).

1.3.7 LINUX: the linker produces undefined symbol "pow" , "sin" etc.

This can happen when you link with dynamic librarie. Link with the math library using the **–lm** linker switch.

2 THE API.

2.1 Introduction to the OpenGUI programming

First thing you should do when you write a OpenGUI program, is to create the instance of APP (or a child of APP). The constructor of this instance will cause all initialisation actions that should be performed at the start-up. Next, add a couple of windows, add to them event handlers to process system messages and you already got a system that filters user INPUT (mouse, keyboard) and dispatches events to the appropriate place (the active window).

The system events are listed below:

Event name	Description
NOEVENT	dummy event
KEYEVENT	key press
MOUSEEVENT	anything from mouse
QUITEVENT	program termination
ACCELEVENT	activating the button, EditBox …
TERMINATEEVENT	window is deleted
RESIZEEVENT	window is resized
MOVEEVENT	mouse cursor has been moved
CLICKRIGHTEVENT	click with the right mouse button
CLICKLEFTEVENT	click with the left mouse button
DRAGLEFTEVENT	mouse drag with left mouse button
DRAGRIGHTEVENT	mouse drag with right mouse button
STARTDRAGLEFTEVENT	start mouse drag with left mouse button
STARTDRAGRIGHTEVENT	start mouse drag with right mouse button
BUTTONHOLDEVENT	mouse button is being pressed for some time
GETFOCUSEVENT	window became active
LOSTFOCUSEVENT	window is not active anymore
WINDOWMOVEEVENT	window has been moved to a new position
INITEVENT	window has been initialised (the right place to draw into it)
REPAINTEVENT	window repainting

If you want some additional events, you can use values from 256 and up. For example:
enum events { CHEAT_EVENT = 256, … };

An event is a class that has the following data members:

```
class GuiEvent {
        protected:
                static  char *eventName[];
                int             type;
                int             key;
                int             buttons, x,y, w, h;
                friend class BaseGui;
                friend class App;
                void AutoRepeatStart(Control *);
                void AutoRepeatDo(void);
        public:
                static void AutoRepeatEnd(void);
                Window  *wnd;   // address of destination
                int             guiType;
                Control *accel;
                GuiEvent(int a=NOEVENT,       int     b=0, int c=0, int d=0)
                {
```

```
                type= a;
                key    = b;
                x      = c;
                y      = d;
                accel = 0;
        }
        int     Key(void)  const { return key;       }
        char    *Name(void);
        int     GetX(void)  const { return    x; }
        int     GetY(void)  const { return    y; }
        int     GetW(void)  const { return    w; }
        int     GetH(void)  const { return    h; }
        int     Type(void)  const { return    type; }
        int     Buttons(void)  const { return buttons;       }
        GuiEvent * GetUserEvent(int);
        void   TranslateUserEvent(void);
        static void    InitUserEvent(void);
        static void    CloseUserEvent(void);
        static void    GetDragVector(int &a, int &b, int &c, int &d);
        static void    SetDragShape(void(*a)(int,int,int,int)=0);
};
```

The detailed description of the variables and the methods of this class can be found in the chapter about GuiEvent. Events can be received at two locations. The first is the event processing function. The prototype of such function is: *int MainProc(GuiEvent *);* The system itself here sends event about finished post routine - QUITEVENT. Also you can send this event to application procedure by *App::AppDone();*, to terminate the main loop. Other events are CLICKRIGHTEVENT, CLICKLEFTEVENT, DRAGLEFTEVENT and DRAGRIGHTEVENT. I should just think so don't any problem with send application procedure any user's defined events by App::SendToApp(GuiEvent *). The other destination where events are going, is a window procedure. The first event that is sent to any window procedure is INITEVENT. The most frequent event is MOVEEVENT that is generated on every mouse movement.

NOTE: All events (except KEYEVENT) can only be sent when there's some window!

2.2 Miscellaneous global functions

In this chapter I describe all features that are not related to any of the described GUI objects. The API described here includes functions that get the mouse position, create error dialogs and provide colour management functionality. The colour management API is essential and large because of 256-colour limit of the library. Colours at position 0 to 15 are used by system, but is possible they're reassigned. The list of the symbolic names for the colours is in the chapter 4.1.32 as type *enum* ***Colors***

2.2.1 "C" void IError(char *s, int flag);

This function pops a dialog box with an error message and waits for the <ENTER> key to be pressed, or for a mouse click on the 'OK' button. The *flag* argument describes an additional action: if *flag* is 0 then only warning occurs, and when *flag* equals to 1 the program will be aborted by calling the *AppDone()* action that will terminate the application loop started with *App::Run()*. The code that is located after the call to *App::Run()* will be executed correctly.

2.2.2 "C" int GetMouseX(void);

2.2.3 "C" int GetMouseY(void);

2.2.4 "C" void RemoveMousePointer(void);

The first two functions are used to get mouse position (in the screen co-ordinates). The procedure *RemoveMousePointer()* allows you to remove the mouse pointer from the screen. This is a helper function for any asynchronous events.

2.2.5 void SetRepeatDelay(int first, int repeat_delay);

This function will set the system parameter that specifies how long the system will wait when a button is pressed before sending repeated **BUTTONHOLDEVENT** (auto-repeat setting).

2.2.6 void SaveScreen(void);

The call to this function saves the whole screen to a bitmap file xxxxxx.bmp, where xxxxxx, is the hexadecimal number generated by the clock() system call.

2.2.7 void SendEvent(Window *w, int event, int key=0, int x=0, int y=0);

This is the same as:

```
void SendEvent(Window *w, int event, int key, int x, int y)
{
       GuiEvent e(event, key,x,y);
       w->SendToWindow(&e);
}
```

2.2.8 void cleanup(void);

Funkcia je veľmi dôležitá. Umožňuje vrátiť konzolu naspäť do textového režimu a reinicializovať klávesnicu, myš tak aby opäť pracovali normálne ako operačný systém očakáva. Za normálnych okolností ju volať nemusíš pretože je volaná automaticky po ukončení funkcie *main().* (je zaregistrovaná cez funkciu *atexit()*).

2.2.9 void FlushInput(void);

Ukončí prípadne otvorené editačné pole v okne.

2.2.10 unsigned int CalculateCRC(unsigned StartCRC, void *Addr, unsigned size);

Vypočíta kontrolný súčet bloku pamäte od adresy *addr* s veľkosťou *size*. Argument *StartCRC* je možné nastaviť na 0 pre normálny kontrolný súčet. V prípade že potrebuješ urobiť kontrolný súčet bloku rozdeleného na viac častí, potom naplníš tento argument posledne vrátenou hodnotou CRC.

2.3 Color Scheme

2.3.1 struct ColorScheme;

```
typedef struct
{
        int             window_back, window_fore;
        int             active_title;
        int             inactive_title;
        int             wnd_bord1;
        int             wnd_bord2;
        int             wnd_bord3;
        int             statusbar;
        int             menu_back;
        int             menu_fore;
        int             menu_back_active;
        int             menu_fore_active;
        int             button_fore;
        int             button_back;
        int             button_fore_pushed;
        int             button_back_pushed;
        int             button_bord1;
        int             button_bord2;
        int             button_bord3;
        int             edit_back;
        int             edit_fore;
        int             edit_bord1;
        int             edit_bord2;
        int             slider;
        int             menuwindow_back;
        int             menuwindow_fore;
        int             pdmenu_back_active;
        int             pdmenu_fore_active;
        int             pdmenu_gray;
        int             edit_disable;
} ColorScheme;
```

2.3.2 extern ColorScheme * CScheme;

This is a global pointer to the *ColorScheme* structure that contains all current color settings for windows, buttons, edit boxes etc. For example, the code:

CScheme->active_title = CBLACK; Cscheme->unactive_title = CWHITE;

will cause the active window title bar will be coloured BLACK and inactive title bar WHITE.

2.4 App

2.4.1 App::App(int videoMode, int & argc, char ** &argv, int bkCol, int appFlags=0);

The instance of this class encapsulates the application. You have to declare this construct at the start of your program (See the “Hello World” example). The first parameter is a number of the default graphics mode (valid mode number values are described bellow, look for *graph_set_mode* function), argv and argc parameters are standard ”C” command line parameters, bkColor defines the initial colour of the screen background. The last parameter, *appFlags* is very important for you. It is a set of bit flags that define the behaviour of your application. The the meaning of bit flags is described in the table below.

The following steps are executed at the start-up:

- if possible, the window database that contains last sizes and positions for registered windows is accessed.
- if possible, the application configuration is loaded from **appname.rc** file.
- command line parameters are parsed, and arguments from the list of system parameters are extracted (consult the ”reserved parameters” chapter for details)
- test for the availability of MMX processor and type of graphics card is performed
- the screen is switched to graphics mode
- colour palette management system is initialized
- screen is cleared
- mouse is checked and, if present, is passed to the system
- call to *SetColorFuzzy (3);* is performed
- the ROOTWINDOW is created if needed
- global variables *cApp* and *cfg* are set.

AppFlags values – should be OR-ed together.

FLAG	DESCRIPTION
APP_WINDOWDATABASE	enable saving the size and position to the window database
APP_CFG	enable configuration file management
APP_MAGNIFIER	enable mouse drags with rubber-band rectangle selection
APP_ROOTWINDOW	enable drawing to the root window – the whole screen
APP_ENABLEALTX	enable termination by **<ALT+X>** key press

Table 1.

2.4.2 App::~App();

This is the destructor method, called when the application ends. It performs the following steps:
- resets color palette
- closes windows database
- closes config file
- resets the windowing system

2.4.3 Window * App::GetRootWindow(void);

Be sure before using this function, that you add switch **APP_ROOTWINDOW** to *App* constructor else program will be aborted. Returned value is a pointer to whole screen window ROOT. You can draw using this pointer with standard window stuff (*WindowLine (), WindowText ()* etc.) NOTE: You can't use any Controls into this window, only drawing!

2.4.4 static void App::SetDelayProc(void (*fnc)(void));

When the system is waiting to user input, it waits into inner loop. Because this time, I add to the library little hack. You can define some very little bit of code, procedure that will be called many times per second. Call with parameter 0 switch this feature off.

2.4.5 static void App::SetTimerProc(void (*fnc)(int));

This is another call-back procedure that will be call per each second. Your user-defined procedure expects one **int** parameter – number of second from the start of your program. Call with parameter 0 switch this feature off.

2.4.6 void App::Run(MainHwnd hwnd=0);

When you created the instance of *App* class, sets call-back for "DelayProc" , it is a right time to call main application loop. System starts processing of user input (keyboard & mouse) till you call *App::AppDone ()*. The mouse movement is automatically converted to move the arrows cursor at the screen and you don't care about it. When you press left button at the controls, system detects it, and sends right event to the appropriate window. When you press button an empty area of window, window got event **CLICKLEFTEVENT.** The events that aren't assigned to any of exist windows, are sends to the application procedure, when one is defined. The pointer at this mysterious procedure is parameter *hwnd* of *App::Run ()*.

2.4.7 void App::Yield(void);

Táto rutina je obzvlašť užitočná pre linux. Pokiaľ sa v niektorom handleri program zdrží tým že musí vykonávať nejakú náročnú činnosť (napr.viac ne 0.1 sec.), je to poznať na trhanom pohybe, prípadne na zmazaní kurzoru myši. Je to spôsobené tým že program nieje v hlavnej slučke, ktorá by mu umožnila spracovávať udalosti od klávesnice a myšky. V linuxe to napriklad spôsobi nemožnosť prepnúť sa na inú konzolu či ukončiť predčasne program klávesom BREAK. Táto funkcia umožňuje potlačiť práve tento nežiadúci efekt. Stačí ju občas (niekoľko krát za sekundu) zavolať a klávesnica bude opäť živá.

2.4.8 static void App::SendToApp(GuiEvent *x);

When you defined the application procedure also, you can send to this procedure any events by this method.

2.4.9 static void App::AppDone(void);

This method you can assign, for example, to menu item "EXIT". When it is once called, it will cause jumping out from the main application loop (return from *App::Run ()*)

2.4.10 int App::Argc;

2.4.11 static char *App::homedir;

2.4.12 char **App::Argv;

Globals for you:

- argc & argv – the values from standard "C" variables

2.4.13 App::BroadcastMessage(GuiEvent *);

Pošle rovnakú správu naraz do všetkých okien. Veľmi vhodné pre vzájomnú komunikáciu medzi jednotlivými oknami.

2.5 GUIEVENT

2.5.1 GuiEvent::GuiEvent(int typ = NOEVENT, int key=0, int x=0, int y=0);

Create the event of appropriate type. In the most cases, you will parse system events only. Creating of the events by the hand and sending them to other windows is less frequently. The response to the system events is adjusted into separate window procedures, each per window. This procedure is fact a one big **switch** statement with many **cases**. For example:

2.5.2 Window * GuiEvent::wnd;

This is a public member of GuiEvent class and it is initialised for window procedures only. It contains the pointer at the target window. It is useful when one procedure is assigned to more windows. Using of this member is generally better, than global pointers.

2.5.3 Window * GuiEvent::accel;

This is a public member of GuiEvent class and it is initialised for window procedures only and **ACCELEVENT** only. It contains the pointer at the Controls that emit this event.

2.5.4 int GuiEvent::guiType;

This is a public member of GuiEvent class and it is assigned for **ACCELEVENT** only. It contains the type of controls. Normally it is unnecessary. The types are: *enum Type {WINDOW, CHECKBUTTON, PUSHBUTTON, POINTBUTTON, EDITBOX, MENUBUTTON, MENUBUTTON2, MENUWINDOW, SLIDEBAR, PSEUDOWINDOW, and LISTBOX};*

2.5.5 int GuiEvent::Key(void);

Returns the value of *key* member of class *GuiEvent*. As for event **KEYEVENT,** it return a key code.

2.5.6 char * GuiEvent::Name(void) ;

Return a text string, which identified the name of events, or string ”NONSENSE”.

2.5.7 int GuiEvent::GetX(void) ;

2.5.8 int GuiEvent::GetY(void);

Returns (x or y) co-ordination for events that initialise this member (**MOVEEVENT, CLICKLEFTEVENT**).

2.5.9 int GuiEvent::GetW(void);

2.5.10 int GuiEvent::GetH(void);

Returns (x or y) co-ordination for events that initialise this member (**DRAGLEFTEVENT ..**).

2.5.11 int GuiEvent::Type(void);

Returns the **type** of the events. The table is listed below.

2.5.12 static void GuiEvent::GetDragVector(int &x, int &y, int &a, int &b);

Set variable *x* and *y* with position of top left corner, *a* & *b* right bottom of the rectangle that is showed when you drag mouse with button pushed at the same time. It is dependent at the switch **APP_MAGNIFIFIER**.

2.5.13 static void GuiEvent::SetDragShape(void(*a)(int x, int y, int offset_x, int offset_y)=0);

Set procedure that will be called when you drag&drop to draw out dragging object. This procedures has four argument *x,y* for position in the current window and *offset_y, offset_x are current offset from [x,y].* To restore default, use as argument 0. For more see the RAD project – source *rad_prj.cc*.

2.6 The events & their parameters

Table of all event and it placement:

EVENT	members using for one[1]						
	Typ()	Key()	GetX()	GetY()	GetW()	GetW()	placement [2]
NOEVENT	*						App, Wnd
KEYEVENT	*	*					Wnd
MOVEEVENT	*	*	*	*			App, Wnd
QUITEVENT	*						App
ACCELEVENT	*	*					Wnd
TERMINATEEVENT	*						Wnd
RESIZEEVENT	*						Wnd
INPUTEVENT1	*	*					Wnd
INPUTEVENT2	*	*					Wnd
CLICKRIGHTEVENT	*		*	*			App, Wnd
CLICKLEFTEVENT	*		*	*			App, Wnd
DRAGLEFTEVENT	*		*	*	*	*	App, Wnd
DRAGRIGHTEVENT	*		*	*	*	*	App, Wnd
STARTDRAGLEFTEVENT	*		*	*			App, Wnd
STARTDRAGRIGHTEVENT	*		*	*			App, Wnd
GETFOCUSEVENT	*						Wnd
LOSTFOCUSEVENT	*						Wnd
BUTTONHOLDEVENT	*	*					Wnd
WINDOWMOVEEVENT	*						Wnd
INITEVENT	*						Wnd
REPAINTEVENT	*						Wnd

2.6.1 EVENT: KEYEVENT

This events indicates key pressing (the keyboard input is going to the active window only). System has some reserved key combination (**ALT+X, CTRL+TAB** etc), and these aren't possible to testing. These reserved keys are converted to appropriate events (**QUITEVENT**, **LOSTFOCUSEVENT** & **GETFOCUSEVENT**)

[1] The member *guiEvent* is **public** and is assigned for events **ACCELEVENT** only. The member *wnd* is **public** and is assigned for window procedures, but not for application handler. It contains a pointer to the target window.

[2] Where this type of event is sent. App = application handler, Wnd = window handler. The events are sent to one destination at a time only!

2.6.2 EVENT: MOVEEVENT

Indicates any movement of the mouse (including button clicks). The members are assigned as follows: If cursor is out of any window, member *key* is 0, and items *x* & *y* contains the position of mouse cursor in screen co-ordination ([0,0] is top left corner). When the cursor is over some window, then event is sent only to the active window. The *key* member is set to 0 if the window is active, -1 when window with cursor is not active, and if the member *key* is greater than 0, then value is an **id** of controls in window. The co-ordinations for movement in case of window are relative to the window workspace (if cursor is at the title of window, then y co-ordination will be negative!). For a good overview of this, look at the **events** example.

2.6.3 EVENT: CLICKLEFTEVENT, CLICKRIGHTEVENT

This event is sent when a simple mouse click is detected. You must use window switch **WCLICKABLE** to enable this feature. Position reported in the event is in the screen co-ordinates, or relative to windows, according to placement.

2.6.4 EVENT: INITEVENT

The first event that is sent to the window when a window is created and is drawn at the screen. It is a right place for drawing controls.

2.6.5 EVENT: QUITEVENT

The last event, that is sent only to the application procedure. This event can generated in one of the two ways: by user pressing **<ALT+X>** or by calling *App::AppDone()*

2.6.6 EVENT: ACCELEVENT

This is a very important event that is generated by system when you activate some Controls (Button, EditBox, ..). The member *key* contains the **id** of current control. The callback handler of the object [you can assign to each Control your callback that will be called at the right time] is called **after** this event parsing into your procedure. The test of some controls:

```
        Void WindowHandler(GuiEvent *p)
        {
                switch(p->Type())
                {
                        case ACCELEVENT:
                        if (p->Key() == button1->GetId())
                        {
                                // some code
                        }
                }
        }
```

NOTE: this event is generated include EditBox activating, but no suggestions to parse it. For it, are intended nest two events.

2.6.7 EVENT: TERMINATEEVENT

Last event. You can delete your stuff. After this, will be window deleted.

2.6.8 EVENT: RESIZEEVENT

If the window is resize-able (switch **WSIZEABLE**), then you can by dragging bottom right corner of windows, resize one. After each size change, this event is generated. You may redraw window at this point (Controls are draws automatically). The current size (of window workspace) is available by calling *GetWW() a GetHW().*

2.6.9 EVENT: REPAINTEVENT

If window is repainted.

2.6.10 EVENT: WINDOWMOVEEVENT

Event is generated when window is moved at new position.

2.6.11 EVENT: DRAGLEFTEVENT, DRAGRIGHTEVENT, STARTDRAGLEFTEVENT, STARTDRAGRIGHTEVENT

If you use switch **APP_MAGNIFIER,** you can select a rectangle area by the mouse dragging. The members are assigned with [x,y] position of top left corner and [w,h] the size of one. If you want exactly first and end point position of dragging, you must use *GetDragVector()*.

2.6.12 EVENT: BUTTONHOLDEVENT

If you hold the mouse button at the Controls **PUSHBUTTON** a while, system will repeatedly emit this event. See also the *SetRepeatDelay*() function.

2.7 Base class: BaseGui

This class is base parent of all graphics objects into OpenGUI library and encapsulates all basic methods. These methods are used in Window, EditBox, PointButton and other GUI classes.

2.7.1 void BaseGui::SetColors(void);

This method is very useful. Sets the global Ink & Paper from local ink & paper from the window. For example:

```
Wnd->SetColors();      //      set colors to window default
Wnd->printf("Hello world\n");
```

2.7.2 void BaseGui::SetParam(int a);

2.7.3 int BaseGui::GetParam(void);

This is an trivial way to solve problem with unavailable related user defined parameter. You can use method *SetParam ()* to any Window to assign some parameter of **int** type. Since you can read this parameter at other place. If returned value is –1, value is not assigned.

2.7.4 char * BaseGui::GetName(void);

2.7.5 void BaseGui::SetName(char *s);

Gets or Sets the text strings to identifying of graphic object. Its length is limited with free memory only.

2.7.6 char BaseGui::GetInk(void);

2.7.7 char BaseGui::GetPaper(void);

2.7.8 char BaseGui::GetX(void);

2.7.9 char BaseGui::GetY(void);

2.7.10 char BaseGui::GetW(void);

2.7.11 char BaseGui::GetH(void);

2.7.12 char BaseGui::GetType(void);

2.7.13 int BaseGui::GetId(void);

Returns the value of ink or paper attribute of instance.

2.8 The configuration object: Config

2.8.1 Config::Config(char *name);

2.8.2 Config::~Config();

Open a configuration file with *name* (without the extension ".rc"). If any not exists, system creates a new empty into the RAM. If you use switch **APP_CFG,** system automatically opens in the *App* constructor. Default config file is *appname.rc*. The pointer at this config saves to global variable *cApp->cfg* and its destructed (and saved) in *App* destructor. The maximal number of items into the config file is up to 500. The number of at once opened config files is any. The maximal size of name of variable is up to 32 chars. The maximal size of saved text string is up to 256 chars.

2.8.3 int Config::ReadInt(char *name);

2.8.4 char const * Config::ReadString(char *name);

2.8.5 double Config::ReadDouble(char *name);

Reads the variable with *name* from config. For types **int** or **double** is value returned directly. For a text strings will be returned a pointer at one. In the case that the follow variable isn't found, it returns 0. If you want alternative effect, you must use other version of this methods...

2.8.6 int Config::ReadInt(char *name, int &p);

2.8.7 int Config::ReadString(char *name, char *p, int maxlen);

2.8.8 int Config::ReadString(char *name, char **p);

2.8.9 int Config::ReadDouble(char * name, double &p);

These functions are almost equivalent of previous, but if an error occurred, its returns 0, and don't overwrite variable. You can set this before calling, and if variable is not found, the value keeps old value. For a strings are two alternative. First, work with pointer and size of pointed place. The string will be copied here. The next version is pointer at pointer at char. This is rewritten with the address of string (no copy).

2.8.10 void   Config::WriteInt(char * name, int value);

2.8.11 void   Config::WriteString(char * name, char *string);

2.8.12 void   Config::WriteDouble(char * name, double value);

Write the variable with adequate type, name and value to config.

2.9 DrawBuffer

2.9.1 DrawBuffer(int ww=0, int hh=0, int t=BMP_MEM, int color=CBLACK, FGPixel *buf=0);

2.9.2 DrawBuffer(int ww, int hh, FGPixel *buf);

2.9.3 Virtual ~DrawBuffer();

2.9.4 void DrawBuffer::BitmapDraw(int a=0, int b=0, int c=-1, int d=-1);

Umožňuje kresliť bitmapu na zakladnú obrazovku ako pozadie (teda nie do okna). Táto funkcia je skôr mimo účel a patrí do rodiny low-level funkcii. Uspešne ju nahrádza **Window::WindowPutBitmap()** v kombinácii z RootWindow.

2.9.5 int DrawBuffer::GetW(void);

Returns the width of image. The real may be about little bit greater (from 1 to 3 bytes) because image width must be rounded to the 32-bit. In a use, if you calculate the position in the bitmap, you must use the real width (next function), but if you want draw bitmap, you must use the logic width of one.

2.9.6 int DrawBuffer::GetH(void);

Returns a height of object.

2.9.7 FGPixel * DrawBuffer::GetArray(void);

Returns the address of image. It is a memory array with dimensions height * width. Pixels are arranged from left to right and from top to bottom.

2.9.8 void DrawBuffer::clear(FGPixel color);

Redraw the all Bitmap image with *color.*

2.10 Bitmap : public DrawBuffer

2.10.1 Bitmap::Bitmap(char *name);

2.10.2 Bitmap::~Bitmap();

This creates ”in memory Bitmap” object from the bitmap file with *name*. The file **must be** Windows BITMAP 8-bit uncompressed. Whether you use or no the **.bmp** extensions is not important. If an operation is OK, then methods data member *type* is not equal **BMP_NONE,** else file doesn't exist or isn't a valid BMP file.

2.10.3 Bitmap::Bitmap(void *image);

You can create a Bitmap object is link one straight to the executable file. You need the utility file2asm that converts file to the assembler file (it is included into the OpenGUI package) and assembler nasm min. version 0.97 - http://www.cryogen.com/nasm. The steps are:

1. transform image file to the assembler file: **file2asm image.bmp** (for the Watcom assembler **wasm** you must use switch **-wasm**)
2. transform the assembler file to the object file: **nasm –f coff image.nsm –o image.o** (for LINUX **–f elf**)

3. add the object file to the executable (standard executable linker)

2.10.4 Bitmap::Bitmap(Window *wndPtr);

You can create the Bitmap object from Window object also. This one has dimensions and contents the same as source window. The image has been copied from, that you can't expect that any changes are appears in it. The applications of one are when you want save the contents of a window to the BMP file.

2.10.5 Bitmap::Bitmap(int width, int height, int Colors = CBLACK);

You can create an empty Bitmap object with ability color layout. This construction is highly usable with the constructors *Window::Window(Bitmap*).* At that *Bitmap* you can apply the all graphics methods of the *Window* class. This is a very power to use. You can draw to the memory buffer ... and later copy all this at the screen by the one command.

NOTE: the number of color/bits per pixel is 256/8 bit, the dimensions of the bitmap is up to free system memory.

2.10.6 int Bitmap::BitmapSave(char *name);

Save the contents of object to the file with **name** and format WINDOWS BITMAP 8-BIT UNCOMPRESSED. The name is allowed without the *.bmp extension only.

2.11 Control : public BaseGui

This family of graphics controls contains the class *PushButton, CheckButton, EditBox and Base menu*. It is child objects of class *Controls*, which is a child object of base class *BaseGui*. There is a more information.

```
void call_back_procedure(CallBack control_object)
{
}
```

NOTE: All of the Control class objects and its parents are on the create time automatically attached with its Window. When a Window is destroyed, Controls are destroyed either automatically. Don't delete it explicitly!

2.11.1 void Control::SetName(char name);

Change the name of the object.

2.11.2 void Control::SetKey(int key);

Change the accelerator key.

2.11.3 int Control::GetKey(void);

Returns a *key* that is assigned to the object.

2.11.4 void Control::SetHandler(ControlCall);

2.11.5 ControlCall Control::GetHandler(void);

Slúži na zisťovanie a nastavovanie obslužnej procedúry ovládacieho prvku.

2.11.6 Window *Control::GetOwner(void);

Vráti ukazateľ na rodičovské okno.

2.11.7 void Control::draw(void);

Refresh the object at the screen.

2.11.8 void Control::Disable(void);

Change the state of the object (draws it as grey and don't activate one).

2.11.9 void Control::Enable(void);

Change the state of the object (draws it as normal and allow activate one).

2.11.10 void Control::ClickUp(void);

Change the state of the object to state "**active"** (as you as click with mouse at one). It is a good choice if you want input to the EditBox immediately without clicking at one.

2.11.11 void Control::SetTrigger(int a);

2.11.12 int Control::GetTrigger(void);

With the procedure *SetTrigger (int)* you can change the state of ON/OFF switches. The function *GetTrigger ()* returns the state of one (**TRUE** or **FALSE**).

2.11.13 int Control::GetLocalId(void);

The all controls have local **id**. This value goes from 0 to .., for each window and its control items. For example: when window contains 5 buttons, then these buttons have local id values from 0 to 4,according to the order of its creating. This is useful for fast and easy testing which button (when you use one call-back procedure for more buttons i.e.) especially with using C*ontrol::GetLastActive();*

2.12 CheckButton, PointButton : public Control

2.12.1 CheckButton::CheckButton(int xs, int ys, char *nm, int key, int flag, Window *w, int fg, int bg, ControlCall f=0, int *var=0);

2.12.2 PointButton::PointButton(int xs, int ys, char *nm, int key, int flag, Window *w, int fg, int bg, ControlCall f=0, int *var=0);

The using of these objects is with addition to the Window class. You can see at the chapter 2.18.49 a 2.18.50

2.12.3 PointButton::ChangeItem(int *variable);

2.12.4 CheckButton::ChangeItem(int *variable);

These are very useful procedures. You can by one, change the data item assigned to the *EditBox.* Other alternative is if you want only refresh data on the screen.

2.13 Label : public Control

2.13.1 Label ::Label(int xs, int ys, char name, Window *parent, int key=0, ControlCall f, int ink, ink paper);

2.13.2 void Label::SetTransparent(void);

Ide o textový objekt, ktorý je možné používať ako ktorýkoľvek iný riadiaci prvok. Tiež je na rozdiel od normálneho textu vždy pri zmene veľkosti okna vykresľovaný odznova. Metoda *Transparent()* slúži na zmenu zobrazovania – text objektu sa bude zobrazovať transparentne.

2.14 BaseMenu : public Control

2.14.1 BaseMenu::BaseMenu(int x, int y, int ww, int hh, char *nm, int key, Window *w, int bg, ControlCall f=0);

The using of these objects is with addition to the Window class. You can see at the chapter 2.18.57.

2.15 PushButton : public Control

2.15.1 PushButton::PushButton(int xs, int ys, int ws, int hs, char *nm, int key, int flag, Window *w, ControlCall f=0);

2.15.2 PushButton::PushButton(int xs, int ys, int key, int flag, Window *w, Bitmap *bm, ControlCall f=0)

The using of these objects is with addition to the Window class. You can see at the chapter 2.18.47 and 2.18.48.

2.15.3 PushButton::Push(void);

2.15.4 PushButton::Release();

Change the visual state of *PushButton().*

2.16 EditBox : public Control

You can use this object to get any text or number from user. When you activate such object, you can use keyboard to edit the text (HOME, DEL, BACKSPACE, ARROWS and ESC work properly). The input is passed to your program when the user hits the **<ENTER>** key. The **ESC** key terminates the input without change.

2.16.1 EditBox::EditBox(int xs, int ys, int ws1, int ws2, char *nm, int key, Window *w, int *p, int ink, int paper, ControlCall f, int mn, int mx);

2.16.2 EditBox::EditBox(int xs, int ys, int ws1, int ws2, char *nm, int key, Window *w, int *p, int ink, int paper, ControlCall f=0);

2.16.3 EditBox::EditBox(int xs, int ys, int ws1, int ws2, char *nm, int key, Window *w, char *p, int ink, int paper, ControlCall f=0);

2.16.4 EditBox::EditBox(int size, int xs, int ys, int ws1, int ws2, char *nm, int key, Window *w, char *p, int ink, int paper, ControlCall f=0);

2.16.5 EditBox::EditBox(int xs, int ys, int ws1, int ws2, char *nm, int key, Window *w, double *p, int ink, int paper, ControlCall f, double mn, double mx);

2.16.6 EditBox::EditBox(int xs, int ys, int ws1, int ws2, char *nm, int key, Window *w, double *p, int ink, int paper, ControlCall f=0);

The using of these objects is with addition to the Window class. You can see at the chapter from 2.18.51 to 2.18.53.

2.16.7 void EditBox::PasswdMode(int m);

You can set the *EditBox* to be in the **password** mode. Once you switched the EditBox to this mode, all characters are printed as '*'. The argument is **bool**, **TRUE** or **FALSE**.

2.16.8 void EditBox::HexMode(int m);

You can set the *EditBox* to be in the **hexadecimal** mode. The argument is **bool**, **TRUE** or **FALSE**.

2.16.9 void EditBox::ChangeItem(int *p);

2.16.10 void EditBox::ChangeItem(char *p);

2.16.11 void EditBox::ChangeItem(double *p);

These are very useful procedures. You can by one, change the data item assigned to the *EditBox.* Other alternative is if you want only refresh data on the screen.
NOTE: You can't use as parameter a character literal (instance->ChangeItem("hello world");, because the C compiler store to the read-only sections, and you may see the **Segmentation Fault** error). The right choice is: *char *s=new char[32]; strcpy(s,"hello world"); instance->ChangeItem(s);*

2.16.12 void EditBox::SetSize(int width);

The length of editable space in the EditBox is sizeable from 1 to 127 chars. When you set the size greater than visible area, text will roll if needed.

2.17 ButtonGroup

When you write the code to manage some menus, sometimes you need program the group of buttons, that would worked as welded. When you choose one, previous goes to the inactive state – this is called a *ButtonGroup* – and make possible to you easy aggregate items like *CheckButtons* and *PointButtons*, or *PushButton* (the word **or** is at a right place, because you can't mix these types together!). The maximal used items at once are 32.

2.17.1 ButtonGroup::ButtonGroup();

2.17.2 ButtonGroup::~ButtonGroup();

Create an empty object for aggregate Controls. It is a first step before using *ButtonGroups**.***

*This object you must destroy **before** destroying Controls or Window. The best place for do it, is Window handler: case TERMINATEEVENT.*

2.17.3 AddToGroup(Control *button, int active=0);

Add *Controls* object to the *ButtonGroup*. The argument *active* is not mandatory (default is **false**) and if you set this to **true**, then object will have the active state at the start. If you mix no acceptable items together, or if you add other than allowed types, an error dialog will be showed.

2.18 Window

2.18.1 Window::Window(Window **itself, int x, int y, int w, int h, char *name, GuiHwnd=0, int ink=IM, int paper=PM, int flag=WTITLED | WFRAMED | WCLICKABLE);

2.18.2 Window::Window(Bitmap *bmpPtr);

2.18.3 Window::~Window();

It is a rectangle space at the screen that is intended to the communication with the user. Its look & feel is derived from the flags assigned on the create time. The maximal size of the window is limited by current video mode resolutions only. It's have own fore and back color, title (or no). You can use many windows at once (it is limited with free memory only), but only and just only may be in the active state. You can draw your window everywhere, but if you go out of bounds of the screen, then graphics system adjust the size and position at the right values. You can set that windows remember your position and size. Each window is active immediate after it is created. To the active window goes user input, key pressing, mouse events and other system events. You can serve these events by defining your own **Window Handler**. It is a single procedure with type *GuiHwnd* and is described below. Because using *Windows* methods to an no-existing object, library has implemented safety mechanism (it is not mandatory but it is a good choice). You can crate Window from Bitmap also. This is a constructor parameter list. First param is pointer at the pointer at the window. It is an address to store window pointer. When window is destroyed, to this pointer is assigned NULL. You can basically test this pointer to NULL to prevent manipulating with destroyed object. Next parameters are standard [x,y] and [width:height]. The parameter name is standard C string, which can be with any length. This name will be displayed it the title of the window (if title is present). *GuiHwnd* is pointer to the window procedure. It is a big **switch** command with many **cases**. You can set this to NULL by default. This an example of little window handler:

```
void WindowProcedure(GuiEvent *p)
{
        switch(p->Type())
        {
                case INITEVENT:
                        // at initialization time
                        break;
                case TERMINATEEVENT:
                        // at end of life
                        break;
                case KEYEVENT:
                        // if you press any key (that is not assigned to some controls)
                        break;
                case CLICKLEFTEVENT:
                        // click at the window workspace
                        break;
                default: // any more
        }
}
```

The events **INITEVENT** and **TERMINATEEVENT** are sends to the window everywhere and automatically. Parameters *ink* and *paper* are fore and back colors. You create your own colors and use it here. The last parameter is *flags.* These define the look of the window. Default is **WSTANDARD** (**WCLICKABLE|WTITLED|WFRAMED**). There are possible values.

The table of window switches:

NAME	DESCRIPTION
WFRAMED	Window has frame
WTITLED	Window has title
WMODAL	Window is a modal
WSOLID	The colors are predefined (CBLACK, CWHITED)
WUNMOVED	You can't move with window

WMENU	Window has menu space
WNOPICTO	No pictograms for window closing – ALT+F4 don't work
WCLICKABLE	You can click to the window
WLASTFOCUS	Window is not active after creating
WSIZEABLE	You can resize window by mouse
WUSELAST	The size and the position will be rather reads from Window database – if any
WSTATUSBAR	You can use *WindowStatusBar()*
WCENTRED	Window has been positioned at the center of the screen
WESCAPE	Window will be closed on ESCAPE pressing
WALIGNED	Window will be alligned in [x,y,w,h] to 32 bit
WFASTMOVE	Override full redrawing on move

These switches you can combine. Also you can create a window from *Bitmap* class. This is useful when you must, for example, draw picture with size 4000x4000 points. The size of normal window is limited by physical screen (or VRAM) size. But you can create window from any bitmap in RAM with size, which is limited with free memory only! All standard methods like WindowLine() etc. is possibly to use and in the last step, you copy this window (or its part) to the screen using WindowPutBitmap().

Destructor of *Window* class execute these steps:
- send **TERMINATEEVENT** to window handler
- delete all controls in itself
- assign NULL to the pointer at the instance
- restore screen contents
- deallocate memory used for one

And at this place, I want notice some lines about using *Window* methods. The co-ordinates [0,0] are in the top left corner of window. You can draw to window "workspace" only, no to borders, title or statusbar. You can use any from all 256 colors (*set_fcolor()* and *set_bcolor()*) and pixel-mode (*set_ppop()*).

2.18.4 void Window::WindowFocus(void);

Activates the window, and gives the focus to it.

2.18.5 void Window::WindowText(int x, int y, char *s);

2.18.6 void Window::WindowText(int x, int y, char *s, int color);

2.18.7 void Window::WindowText(int x, int y, char *s, int color, int bkcol);

Draws the text string at [x,y] position. If colour arguments are not used, the default window colours are used. The maximum allowed string length is 256 chars.

2.18.8 void Window::WindowBox(int x, int y, int w, int h);

2.18.9 void Window::WindowBox(int x, int y, int w, int h, int color);

Draws a filled rectangle. If the colour argument is not used, the last used colour is used. The size is clipped to fit the window workspace size.

2.18.10 void Window::WindowPixel(int x, int y, FGPixel c);

2.18.11 void Window::WindowPixel(int x, int y);

2.18.12 unsigned Window::WindowGetPixel(int x, int y);

WindowPixel() draws a pixel at the [x,y] position with color *c*. *WindowGetPixel()* is the opposite of the latter. It returns values from 0 to 255. If the co-ordinates are invalid, it returns 256 to indicate an error.

2.18.13 void Window::WindowRect(int x, int y, int w, int h);

2.18.14 void Window::WindowRect(int x, int y, int w, int h, int color);

Draws a rectangle in the window. If you don’t use colour, then the last used colour is selected. The rectangle position and size is clipped to the window ‘workspace’.

2.18.15 void Window::WindowLine(int x1, int y1, int x2, int y2);

2.18.16 void Window::WindowLine(int x1, int y1, int x2, int y2, int col);

Draws a line from (x1,y1) to (x2,y2) in the current window, colour usage is like described above.

2.18.17 void Window::WindowDrawCircle(int x, int y, int r, int c);

2.18.18 void Window::WindowDrawCircle(int x, int y, int r);

2.18.19 void Window::WindowFillCircle(int x, int y, int r, int c);

2.18.20 void Window::WindowFillCircle(int x, int y, int r);

Draws empty circle (WindowDrawCircle) or filled circle (WindowFillCircle). The radius must be greater than zero. Color setting is like in previous cases.

2.18.21 void Window::WindowScrollDown(int x, int y, int h, int w, int n);

2.18.22 void Window::WindowScrollUp(int x, int y, int h, int w, int n);

Roll the rectangle about *n* pixels up or down.

2.18.23 void Window::WindowStatusBar(int x, char *s, int c = IM);

When you create a window with **WSTATUSBAR,** then your window will have in bottom part reserved some space named – Status Bar. This part of the window isn’t in the window workspace. That only one method is for printing to this place – *WindowStatusBar()*. The string *s* is printed at the position *x*, (y co-ordinate is constant) with color *c*.

2.18.24 int Window::printf(const char *, ...);

2.18.25 int Window::printf(int, int, const char *, ...);

These are a full featured traditional *printf()* functions. Here is a differences from *WindowText()*. Text is buffered (buffer size is 256) until ‘\n’ is not occurred or to the next *printf()* call. NOTE: If you use [x,y] start of printed text, then everything is alright, else text is printed at next position past the last output (with ‘\n’ case at the next line, column 0). The line height is set from font size at the window create time. When the last line is reached, a content of window (entire workspace) is roll up. The colors are last used (no window default).

2.18.26 void Window::SendToWindow(GuiEvent *p);

This procedure is used to communication between window handler and your code. Also you can send events that is not parsed (default case) to other window. For example, this is key pressing simulation:

```
...
GuiEvent event(KEYEVENT, ESC);  // ESC key
MainWnd->SendToWindow(&event);  // window MainWnd receive the event ‘KEYEVENT; with value Key() == ESC
```

2.18.27 void Window::WindowLock(void);

2.18.28 void Window::WindowUnLock(void);

When you draws to the window first of all you modify memory image in RAM, step two is copying adequate parts of this image at the screen. The operation of testing which parts will be updated at the screen is relatively CPU difficult and very increase with numbers of windows in system. When you draw too many small objects, or some complicated, you can ‘lock’ window (disable screen update) and after this call *WindowUnLock()* for update entire window at the screen.

2.18.29 void Window::WindowPutBitmap(int x, int y, int xs, int ys, int w, int h, DrawBuffer *p);

This procedure copies any part of a bitmap from RAM to the window. Position [x,y] is in window,, position [xs,ys] is source position in the bitmap *p* relative to its left corner. [w,h] are dimensions of copied rectangle. When is bitmap size greater than window, it will be clipped. When is Bitmap smaller than [w,h], it will be copied many times repeatly.

2.18.30 void Window::WindowGetBitmap(int x, int y, int xs, int ys, int w, int h, DrawBuffer *p);

Copy rectangle described by [x,y,w,h] in the window at position [xs,ys] in the bitmap *p*. When *w* or *h* is greater than bitmap size, it will be clipped to this.

2.18.31 void Window::SetName(char *s);

Set name in the title bar of window.

2.18.32 void Window::WindowFillPolygon(int nodes, int array[][2], int col);

2.18.33 void Window::WindowDrawPolygon(int nodes, int array[][2], int col);

Draw any polygon to the window. Number of nodes is unlimited. When you not define last parameter *col*, color will be last used. Parameter *array* is two-dimensional array of **int** – it contains [x,y] positions of all nodes of the drawing polygon.

2.18.34 void Window::WindowSpline(int points[8]);

Draws the **bezier spline** between four points (two integer for each [x,y]).

2.18.35 void Window::FlushInput(void);

Switch editbox from edit to static mode (all changes are saved – if valid).

2.18.36 void Window::WindowSetWorkRect(int,int,int,int);

This allow dynamically change the size of window workspace (the rectangle where you can draw). For example: *Wnd->WindowSetWorkRect(0, 0, Wnd->GetW(),Wnd->GetH());* allows drawing to the entire window include title or borders.

2.18.37 void Window::WindowGetWorkRect(int &,int &,int &,int &);

Returns the position and size of window workspace.

2.18.38 FGPixel * Window::GetArray(void);

Returns address of IN-RAM window image.

2.18.39 int Window::GetW(void);

2.18.40 int Window::GetH(void);

2.18.41 int Window::GetWW(void);

2.18.42 int Window::GetHW(void);

- GetW() – absolute width of window
- GetH() – absolute height of window
- GetWW() – width of workspace in the window
- GetHW()– height of workspace in the window

2.18.43 static Window * Window::GetCurrent(void);

Returns the pointer to the current window.

2.18.44 void Window::WindowRepaint(int x, int y, int w, int h);

2.18.45 void Window::WindowRepaintUser(int x, int y, int w, int h);

2.18.46 void Window::WindowClip(int &x, int &y, int &w, int &h);

Refresh section [x,y,w,h] of the window at the screen. The source is IN-RAM image of window. Window can’t be locked.

2.18.47 PushButton * Window::AddPushButton(int xs, int ys, int ws, int hs, char *name, int key=0, ControlCall f=0, int flag =BNORMAL);

Add PushButton to the window. Parameters [x,y,w,h] are traditional. Argument *name* is the title of button. Argument *key* is the code of the activate key. Argument *f* describes call-back procedure, which will be called when button is activated (but before call procedure will be sent event **ACCELEVENT** to the window procedure).

2.18.48 PushButton * Window::AddPushButton(int xs, int ys, int key, Bitmap *b, ControlCall f=0, int flag =BNORMAL);

Add PushButton to the window. Parameters [x,y] are traditional. The dimensions of button are calculated from the bitmap size. Bitmap *p* is standard 256 colors bitmap. Argument *key* is the code of the activate key. Argument *f* describes callback procedure, which will be called when button is activated (but before call procedure will be sent event **ACCELEVENT** to the window procedure).

2.18.49 CheckButton * Window::AddCheckButton(int xs, int ys, char *name, int key=0, int * variable=0, ControlCall f=0, int flag =BNORMAL);

2.18.50 PointButton * Window::AddPointButton(int xs, int ys, char *name, int key=0, int * variable=0, ControlCall f=0, int flag =BNORMAL);

This object is two-state switch button. Its state is controlled by value (int* *variable*). Look & feel of button is set immediately after drawing by appropriate value.

2.18.51 EditBox * Window::AddEditBox(int x, int y, int ws1, int ws2, char *name, int key, int *p, ControlCall f, int min_val, int max_val);

2.18.52 EditBox * Window::AddEditBox(int x, int y, int ws1, int ws2, char *name, int key, int *p, ControlCall f=0);

2.18.53 EditBox * Window::AddEditBox(int x, int y, int ws1, int ws2, char *name, int key, char *p, ControlCall f =0);

2.18.54 EditBox * Window::AddEditBox(int size, int x, int y, int ws1, int ws2, char *name, int key, char *p, ControlCall f =0);

2.18.55 EditBox * Window::AddEditBox(int x, int y, int ws1, int ws2, char *name, int key, double *p, ControlCall f , double min_val, double max_val);

2.18.56 EditBox * Window::AddEditBox(int x, int y, int ws1, int ws2, char *name, int key, double *p, ControlCall f =0);

All of these items allow you an interactive input data from user (int, double or string). For numbers you can set the maximal and minimal values. When you enter data value out of this range, old values not replaced by this. Arguments [x,y] are position of EditBox in the window, *ws1 and ws2* are widths of the title part of editbox and edited values. The value of argument *ws2* is computed by formula *ws2=length_in_char*8+8.* Pointer *p* is pointer to the edited data item.

2.18.57 BaseMenu * Window::AddBaseMenu(char *name, int key=0, ControlCall f=0);

Add to the window menu item of *BaseMenu* type. You must set the switch **WMENU** when you create window with the menu bar. Arguments are like previous case.

2.18.58 SlideBarH *Window::AddSlideBarH(int x, int y, int min, int max, int step, int *val, ControlCall f=0);

Add to the window controls – slidebar. You can use this object to change int varible with any steps and in any range. Arguments are: position [x,y], *min & max* are minimal and maximal used values of controlled variable, *step* is minimal step to modification variable and *f* is procedure that is called whenever values is changed.

2.18.59 ListBox * Window::AddListBox(int xs, int ys, int wpol, int hpol, int w, int h, int *cnt, int *pos, void (*drawone)(int,int,int,int,void *), void *data=0);

2.18.60 ListBox * Window::AddListBox(int xs, int ys, int wpol, int hpol, int w, int h, int cnt, void (*drawone)(int,int,int,int,void *), void *data);

Viď kapitolu **Chyba! Nenalezen zdroj odkazů.** ListBox.

2.18.61 Label * Window::AddLabel(int xs, int ys, char *name, int key=0, ControlCall f=0, int ink=PM, ink paper=IM);

Pridá do okna textové návestie *name* na pozíciu [x,y], ďalej je možne zadať obslužnú rutinu, ktorá bude zavolaná pri aktivácii myškou alebo priradenou klávesou. Farby sú default zdedené z rodičovskeho okna.

2.18.62 void * Window::SetWindowMoveStyle(int);

Určuje ako sa bude chovať okno pri presune na inú pozíciu. TRUE = bude sa prekresľovať celé okno, FALSE = nebude sa prekresľovať celé okno (vhodné pre pomalšie stroje). Volanie tejto funkcie dokáže predefinovať nastavenie prepínačom **–nofull**. Tiež je možne chovanie okna ovplyvniť pri jeho vytváraní a to prepínačom **WFASTMOVE**.

2.19 MenuWindow

This class is derived from class *Window*. It is designed to using as context menu, tool bars, etc. At once can exist one instance only. Object is temporary only and is automatically destroyed on:

- event *LOSTFOCUSEVENT* (you click at other window)
- select menu item of types *BaseMenu or PushButton from this window*
- **ESC** key is pressed

The position of added controls is calculated automatically.

NOTE: don't use operator delete to this class!

2.19.1 MenuWindow::MenuWindow(int x, int y, int w, int h, char *name ="Menu Window", GuiHwnd proc=0, int bg = CGRAY3);

Parameters *x,y,w and h* are standard position and size of window. The name of window is no relevant. The window handler (optional) is *proc*. The last parameter is background color. Certainly, you can use **all** drawing and other methods of Window class.

*NOTE: don't assign pointer of object MenuWindow to type Window *, because overloaded methods crash your program!*

2.19.2 int MenuWindow::GetXM(void);

2.19.3 int MenuWindow::GetYM(int a);

2.19.4 BaseMenu *MenuWindow::AddMenu(char *name, int key=0, ControlCall proc=0);

Add to the menu item of type *Menu* (pull-down menu) with name *nm*, assign *key* with them and assign call-back procedure (type *ControlCall* – void proc(void);). This procedure will be executed after select this Controls. The side effect is menu destroying. More information is in the chapter 2.19.4.

2.19.5 CheckButton *MenuWindow::AddCheckButton(char *name, int key=0, int * variable=0, ControlCall proc=0, int flag=BNORMAL);

2.19.6 PointButton *MenuWindow::AddPointButton(char *name, int key=0, int * variable=0, ControlCall proc=0, int flag=BNORMAL);

Add to the menu switch button. More information is in the chapter 2.18.50 *Window::AddPointButton()* and *2.18.49 Window::AddCheckButton()*.

2.19.7 EditBox *MenuWindow::AddEditBox(int ws1, int ws2, char *name, int key, int *p, ControlCall proc, int min, int max);

2.19.8 EditBox *MenuWindow::AddEditBox(int ws1, int ws2, char *name, int key, int *p, ControlCall proc=0);

2.19.9 EditBox *MenuWindow::AddEditBox(int ws1, int ws2, char *name, int key, char *p, ControlCall proc=0);

2.19.10 EditBox *MenuWindow::AddEditBox(int size, int ws1, int ws2, char *name, int key, char *p, ControlCall proc=0);

2.19.11 EditBox *MenuWindow::AddEditBox(int ws1, int ws2, char *name, int key, double *p,ControlCall proc, double min, double max);

2.19.12 EditBox *MenuWindow::AddEditBox(int ws1, int ws2, char *name, int key, double *p,ControlCall proc=0);

Add to the menu EditBox. More info in chapter 2.18.51 *Window::AddEditBox()*.

2.19.13 void Separator(void);

Adds a separator to the menu.

3 WIDGETS

3.1 Intro

Windows and buttons are not enough to create a full-featured application. For example, when you want to save or open a file, you need to use many objects at once. The library provides a set of convenient widgets for you.

3.2 FileDialog

3.2.1 FileDialog(void (*filesel)(char *, FileDialog *), char *dir=0, char *flt=0, char *namewnd="File Dialog", int m=FDIALOG_OPEN, int ink=0, int paper=PM);

3.2.2 FileDialog(void (*filesel)(char *), char *dir=0, char *flt=0, char *namewnd="File Dialog", int m=FDIALOG_OPEN, int ink=0, int paper=PM);

3.2.3 char * FileDialog::GetDir(void);

3.2.4 char * FileDialog::GetName(void);

3.2.5 void FileDialog::SetDir(char *d);

3.2.6 void FileDialog::SetFilter(char *f);

3.2.7 void FileDialog::SetMode(int m);

3.2.8 void SetParam(int p);

3.2.9 int GetParam(void);

You will use this widget (or dialog) frequently – always when you will work with disk files. For example:

*FileDialog *fd = new FileDialog(file_select, ".", ".cc .c .asm .nsm", "Save file", FDIALOG_SAVE* | *FDIALOG_MODAL* | *FDIALOG_SAVEDIR);*

This example shows dialog with title 'Save file'. Shows files with extensions '*.cc .c .asm .nsm'* only, window will be modal and remember position in the file system tree (starts with the current directory). At the end, mysterious parameter *file_select* is a pointer to procedure that will be called, when you select right file. This procedure has only one parameter – pointer to string – file name with the full path. Parameter *filter* is a list of the file extensions (with dot) separates with space. Some other methods are here: *GetDir()* – returns current directory, *GetName()* – returns filename only, *SetFilter(char *)* set filter to the string .., *SetDir(char *)* – change current directory, *SetMode(int)* – change mode (values are in the table below).

File dialog control:

The window contains two push buttons – for select and for cancel the file (eventually **<ENTER> or <ESC>**). In the top left corner is EditBox with current highlighted file or directory. When you press **<SPACE>** or click at this with mouse, you can edit this field. By entering other directory or drive, you can slightly change the current position in the file system. You can move about your disks using traditional methods – keys **<ARROW KEYS>, <HOME>, <END>,<PGUP>, <PGDOWN>.**

SWITCH	DESCRIPTION
FDIALOG_OPEN	Dialog for file open
FDIALOG_SAVE	Dialog for saving (new name allowed)
FDIALOG_MODAL	No switching to other window allowed
FDIALOG_SAVEDIR	Save last position

NOTE: At the time you can have one copy of the file dialog. This is always deleted when you create next instance.

3.3 ColorDialog

3.3.1 ColorDialog(char *capture=0, void (*FNC)(FGPixel), FGPixel paper=PM);

3.4 ListBox

Is class intended for browse and select items from list (i.e. directory contents) as you can see in the file dialog. The number of the items and position in the list you can change dynamically.

3.4.1 ListBox::ListBox(int xs, int ys, int wpol, int hpol, int w, int h, int cnt, void (*drawone)(int, int, int, int, void *), Window *wnd, void *data=0);

This construction creates in window *wnd* instance of *ListBox* class. Its top left corner is at position [xs,ys]. The size of ListBox will be *wpol* items abreast in *hpol* lines (matrix with this size at this position). Each items of this table will have dimensions [w*h] pixels. Number of lines of table can be many greater than *wpol* parameter. Items that are not visible, you can roll with methods *Up() and Down()*. To paint each items, you must define your own paint procedure - *void (*drawone)(int x, int y, int index, int mode, void *data).* Parameters are: position (in window) [x,y], index of displayed items, *mode* is TRUE (current item=selected) or FALSE (others), parameter *data* is generic user defined pointer (obviously **this** pointer, because *drawone* can be static method only). For more information see at source *src/widgets.cc* or example *widget*.

3.4.2 void ListBox::draw(void);

Redraws entire ListBox in the window.

3.4.3 int ListBox::GetCurrent(void);

3.4.4 int ListBox::GetFirstVisible(void);

Returns index of current items or first displayed items.

3.4.5 int ListBox::GetSize(void);

3.4.6 void ListBox::SetSize(int);

3.4.7 void ListBox::Resize(int amount);

GetSize() returns number of all items in *ListBox. SetSize(int)* change the number of items in the object. *Resize(int)* resize ListBox by offset. For example: *Resize(-3)* reduce in size by 3 items.

3.4.8 void ListBox::SetToItem(int item);

3.4.9 void ListBox::SetToItemRel(int item);

Sets item as current absolutely or relative by offset.

3.4.10 void ListBox::RedrawItem(void);

Redraw current item.

3.4.11 int ListBox::Test(int mousex, int mousey);

Returns index of mouse clicked items (co-operative with CLICK*****EVENT) or –1 if no items has been selected.

3.4.12 void ListBox::Up(void);

3.4.13 void ListBox::Down(void);

3.4.14 void ListBox::Right(void);

3.4.15 void ListBox::Left(void);

Move current position in the ListBox to the corresponding direction by one (binding with keys).

3.5 PROGRESSBAR

3.5.1 ProgressBar::ProgressBar(Window *w, int x, int y, int w, int h, int step);

3.5.2 void ProgressBar::setProgress(int a);

3.5.3 int ProgressBar::progress(void);

You can use this simple widget to show the state of some working job. Parameters are: pointer at parent window, position in the window, objects dimensions and especially *step*. This is number of steps, in which operation will be executed. For example: for *step*=80, call *setProgress(40)* shows 50%. Method *progress()* returns current value.

NOTE: this object is no connected with window – you must destroy it explicitly on TERMINATEEVENT !

3.6 TextEditor

3.6.1 TextEditor::TextEditor(TextEditor **myself, char *name, int font=FONT0816, int ink=CGRAY3, int paper=CBLACK);

Parametre: myself je ukazateľ na ukazateľ na vytváraný objekt. Ide o obdobu mechanizmu pre zabezpečenie použitého pri triede *Window*. Po zrušení objektu užívateľom (File->Close) a teda nie volaním metódy *Close().* Je takto zabezpečené asynchrónne znulovanie ukazateľa. Vytvorí okno z jednoduchým textovým editorom vhodným na vytváranie a editáciu obyčajných textových súborov. Menu obsahuje položky **File, Search, Options** (ALT+F, ALT+S, ALT+O). Čo sa pod týmito položkami skrýva nebudem zbytočne popisovať. Okrem týchto kláves je možné používať klávesy PgUp, PgDown atď., úplne klasickým spôsobom. Menej bežné je CTRL+PgUp pre skok na začiatok dokumentu, CTRL+PgDown na koniec dokumentu.

Je možné vytvoriť viac okien pre rôzne súbory a striedavo pracovať so všetkými. Poloha a veľkosť okna sa uchováva v konfiguračnom default súbore. Vzhľadom nato že všetky inštancie zdieľajú tie isté premenné, budú pri každom otvorenom okne platné údaje posledne zatvoreného. Aby sa z dokumentom vo vnútri okna dali prevádzať základné úkony, poskytuje objekt nasledovné metódy.

3.6.2 void NewBuffer(void);

3.6.3 void OpenBuffer(char *s);

3.6.4 void ReopenBuffer(void);

3.6.5 void SaveBuffer(void);

3.6.6 void SaveAsBuffer(char *s);

3.6.7 void Close(void);

3.6.8 void Goto(int);

3.6.9 void ReadOnlyMode(void);

NewBuffer() zmaže obsah okna a vytvorí nový prázdny dokument. Všetky predchádzajúce zmeny budú nenávratne prepísané. *OpenBuffer(char *name)* načíta do editora nový textový súbor, predchádzajúci dokument bude takisto bez výstrahy zmazaný (názov môže samozrejme obsahovať okrem mena aj relatívnu alebo absolútnu cestu), *Reopen()* znovu načíta posledne otvorený súbor. *SaveBuffer(void)* uloži rozrobený dokument pod aktuálnym názvom. *SaveAsBuffer(char *name)* uloží súbor pod zadaným názvom. *Close()* zatvorí okno a zruší objekt. V prípade že v texte boli neuložené zmeny, bude najprv zobrazený dialog pre potvrdenie ich uloženia. V prípade že si zadal aj ukazateľ na na ukazateľ na inštanciu (viď konštruktor) bude tento po deštrukcii automaticky zrušený. *Goto()* sa umožňuje programovo premiestniť na iný riadok v texte. *ReadOnlyMode()* nenávratne nastaví režim prezerania (nieje možné dokument meniť).

4 LOW LEVEL GRAPHICS

4.1 The PHILOSOPHY

If you no need windows etc., you can write code with simple low level graphics primitives. This branch is highly optimised and fast. Be enough quickly and compose first layer for window extension routines. It is alike as Borland BGI and other basic graphics library. As far as don't alias initiate accordingly (x,y) designate elementary screen position where position (0,0) is top-left corner of screen. Parameters because dimensions (w,w) are all the time past parameters fences. Colour setting is permanent to next change. Accordingly all operation affects current pixel mode (AND, OR ..) , clipping and Font.

4.1.1 long long _rdtsc(void);

Returns a 64 bit value contains the number of processor ticks from start of the computer.

4.1.2 int test_mmx(void);

Returns 1, if your processor is MMX positive. Some routines are optimised for using MMX instructions (chars printing).

4.1.3 void set_mmx(void);

Turn processor to MMX mode.

4.1.4 void reset_mmx(void);

Turn processor back to the normal mode.

4.1.5 int mmx;

Global variable – it contain state of processor MMX extension.

4.1.6 int get_key(void);

Returns the last pressed key code or 0 if none is pressed. The codes of the key you can find in a headers file **base.h**.

4.1.7 int set_ppop(int mode);

Set the pixel operation mode for the all routines. Return value is previous draw mode. The table describes one.

Mode	Operation
_GSET	Pixel = Color
_GAND	Pixel = Pixel & Color
_GOR	B = Pixel | Color
_GXOR	Pixel = Pixel ^ Color
_GPLUS	Pixel = Pixel + Color
_GMINUS	Pixel = Pixel – Color
_GNOT	Pixel = ~ Pixel

_GTRANSP	Pixel = (src!=0)src?dst:

4.1.8 void draw_box(int x, int y, int w, int h);

4.1.9 void fill_box(int x, int y, int w, int h);

Draw the rectangle with current color (draw=shape, fill=filled)

4.1.10 void draw_point(int x, int y);

4.1.11 int get_point(int x, int y);

Draw pixel with current color and mode at position [x,y]. The function *get_point()* get the value of color at position [x,y]. Value is in range 0 .. 255 for valid position or 256 for out of screen.

4.1.12 int graph_set_mode(int mode);

This is about cardinal action at library. Switch graphic to graphic and back to text mode. In most cases but you uses this only to graphics mode. Text mode switching on terminating your program is automatic across *atexit()*. Function returns TRUE when all is OK, or FALSE.

Number	Resolution
GTEXT	Text mode
G320x200	320x200 (no MODEX)
G640x480	640x480
G800x600	800x600
G1024x768	1024x768
G1280x1024	1280x1024 (from 2 MB VRAM)
G1600x1200	1600x1200 (from 4 MB VRAM)

These commands will be executed by entering graph mode ...

```
        set_clip_rect(X_width, Y_width, 0, 0);
        set_font(FONT0816);
        set_colors(CBLACK, CWHITE);
        set_ppop(_GSET);
```

4.1.13 void draw_line(int x, int y, int x1, int y1);

4.1.14 void draw_hline(int x, int y, int w);

4.1.15 void drawto_line(int x, int y);

Draw line from [x,y] to [x1,y2].
Draw horizontal segment from [x,y] with [w] width.
Draw line from last used pixel to [x,y].

4.1.16 void clear_frame_buffer(FGPixel c);

Fill the screen with color.

4.1.17 void _palette(unsigned register, unsigned int RGBvalue);

Set the palette entry (0..255) to parameter RGBvalue. The value is computed as (RED<<16)+(GREEN<<8)+BLUE. The color values are 6 bit. (0 .. 63).

4.1.18 int X_max, Y_max, X_width, Y_width;

These globals describes the size of the screen in the pixels. X_max=X_width-1. Y_max=Y_width-1.

4.1.19 void draw_spline(int points[8]);

Draw the bezier spline. Spline is described with four vertices.

4.1.20 int get_colordepth(void);

Returns the color depth of screen. On now its 8 (256 colors), or 32 (16M) for linux linear framebuffer.

4.1.21 void set_fcolor(FGPixel c) ;

4.1.22 FGPixel get_fcolor(void);

Set or get the current color for foreground.

4.1.23 void set_bcolor(FGPixel c) ;

4.1.24 FGPixel get_bcolor(void);

Set or get the current color for background.

4.1.25 int GetFreeColors(void);

Returns the number of the free palette entries. At the start is it returns 256-16=240 colours.

4.1.26 int CreateColor8(int red, int green, int blue);

4.1.27 int CreateColor(int red, int green, int blue, int index);

At the start-up the system allocates 16 predefined/reserved colours (CBLACK, CDARK, CGRAYED, CGRAY1, CGRAY2, CGRAY3, CBLUE, CBLUELIGHT, CGREEN, CGREENLIGHT, CRED, CREDLIGHT, CBROWN, CYELLOW, CWHITED and CWHITE). It is possible to add colours using the *CreateColor (red,green,blue)* function. The RED, GREEN and BLUE parameters are values of individual colour components (like in RGB signal). Their values range from 0 to 63. The difference between the colour allocated by the system and the colour it was asked to allocate can be set with the *void SetColorFuzzy (int)* function mentioned below. The CreateColor() function returns a value of the index of the colour in the palette (0 to 255), or a negative value (–1 or -2) if an error occurred. CreateColor() will allocate the first available colour entry in the palette. If you want to allocate an exact colour at the exact location in the palette you should use four arguments version of *CreateColor ()*. The last parameter is the number of the palette cell you want to allocate in the palette. The four-arguments version of *CreateColor()* does not work with *SetColorFuzzy ()*.

4.1.28 void SetColorFuzzy(int);

Since the number of colours in the palette is limited it helps to share close colours. The logic of colour sharing is slightly intelligent. When you want to create a new colour, the system will check all allocated colours, and if your colour is already allocated the system will simply return the index of the palette entry of the allocated colour, instead of allocating a new palette entry for the colour. At this place the colour allocation algorithm is looking for the closest possible choice of colour. The procedure *SetColorFuzzy ()* has one parameter. This parameter is the maximal skew from allocated colours to the desired colour. For example, if you will use *SetColorFuzzy (3); CreateColor (13,8,36);* and a colour with value (10,8,39) is already allocated, then this colour will be returned as the best match, and the system will save one palette entry. This is a compromise, but for most applications it is good enough. If you call this function with parameter 0, then returned colour will match exactly. In practice, when you want to display bitmap pictures, the reasonable values are between 1 and 6.

4.1.29 void GetColor(char &red, char &green, char &blue, int index);

This function returns the RGB values for colour at the palette entry located at *index*.

4.1.30 void DeleteColor(int);

This function deletes the colour from the palette if it is not shared by few colour allocations, or decreases the reference count of the colour cell otherwise.

4.1.31 void set_colors(FGPixel foreground, FGPixel background);

4.1.32 const FGPixel CBLACK, CDARK, CGRAYED, CGRAY1, CGRAY2, CGRAY3, CBLUE,CBLUELIGHT,CGREEN,CGREENLIGHT,CRED,CREDLIGHT,CBROWN,CYELLOW,CWHITED, CWHITE;

Procedure *set_colors()* set both colors at once. *Enum Colors* are basic systém colors (16). These are set on the startup.

4.1.33 int text_out(int x, int y, char *text);

4.1.34 int gprintf(int x, int y, const char *format,...);

Function *text_out()* is general to the text output to screen. Print a standard C string at position [x,y], uses current draw mode, colors and font. Return value is the next *x* co-ordination. Function *gprintf()* is traditional printf() version. Return value is number of printed characters. NOTE: the maximal string length for all text output functions will be wrapped to 256.

4.1.35 void text_to_buffer(int x, int y, char *s, int w, int h, char *p);

Classic text output, but to the buffer in RAM with size [w*h] bytes from address *p*.

4.1.36 void fill_convex(int c, int a[][2]);

4.1.37 void draw_convex(int c, int a[][2]);

Fill or draw convex polygon with *c* nodes. Nodes are into two-dimensional array of **int**. The number of the nodes is not relevant.

4.1.38 void get_block(int x, int y, int w, int h, FGPixel *ptr);

4.1.39 void put_block(int x, int y, int w, int h, FGPixel *ptr);

4.1.40 unsigned int areasize(int w, int h);

Basic bitblit functions for copying sprites to/from VRAM/RAM. The draw mode is significant for *put_block()* only. Function *areasize()* returns the size of the memory block with width w and height h.

4.1.41 void set_clip_rect(int w, int h, int x, int y);

Set clipping rectangle for the **all library functions** (include Window stuff).

4.1.42 extern void draw_circle(int x, int y, int r);

4.1.43 extern void fill_circle(int xs, int ys, int r);

Draw or fill the circle with radius *r* at position [x,y].

4.1.44 int set_font(int font);

4.1.45 enum fonts { FONT0406, FONT0808=1, FONT0816, FONT1220, FONT16254 };

Set current font. On now are the five fonts in this library. Return value is code for previous font.

5 OPENGUI SOURCER

5.1 Čo to je a načo je to dobré ?

Táto utilitka slúži na vizuálne navrhovanie základnej kostry budúcej aplikácie používajúcej OpenGUI API. Pomocou myši a menu si vytvoríš prázdne okná do ktorý umiestniš ovládacie prvky (tlačítka, menu editačné okná …) a geometrické prvky (čiary, štvorce, bitmapy …). Vlastnosti (farba, pozícia, veľkosť …) každého prvku je možné kedykoľvek dodatočne zmeniť. U ovládacích prvkov sa zadáva ešte názov handleru, prípadne riadiacej premennej. V editore farieb je možné namiešať akúkoľvek z viac než 264000 možných farieb. Zá. Niesu to žiadne DELPHI, ale na vygenerovanie prvotnej kostry budúceho programu by ti to môhlo stačiť.

5.1.1 File – new, open, save, save as, about, exit

Klasika. Práca zo súbormi a tak ďalej.

5.1.2 Editor

Položka je pristupná až po prvom prekompilovaní práve editovaného projektu. Otvorí okno s jednoduchým ale účelným textovým editorom umožňujúcim bežné operácie.

5.1.3 Compile

Spustí generovanie zdrojáku v C++. Pokiaľ chýbajú niektoré informácie – napr. názov premennej pri objektoch, ktoré to vyžadujú – systém ohlási chybu s označením miesta výskytu chyby. Pokiaľ všetko prejde bez chýb oznámi systém info o tom čo ďalej – kostra budúceho programu je vygenerovaná, ďalej je to už tvrdá drina.

5.1.4 Arrange – forward, backward, clone, delete

Premiestňovanie, duplikovanie a mazanie objektov pre aktuálne okno.

5.1.5 Options

5.1.6 Window – add, remove, duplicate

5.1.7 Control

5.1.8 Shape

5.1.9 Colors

5.1.10 Bitmaps

5.1.11 Schemes

5.1.12 Defaults

Editor pre premenné použité v projekte a ich hodnoty.

6 APPENDIX

6.1 Command line parameters

6.1.1 –help

Shows help page.

6.1.2 –res number

Number	Rozlíšenie – Resolution
1	320x200 (no MODEX)
2	640x480
3	800x600
4	1024x768
5	1280x1024 (min 2 MB VRAM)
6	1600x1200 (min 4 MB VRAM)

Change default video mode resolution (in the App constructor).

6.1.3 –driver number

Explicit driver setting (not for LINUX).

NUMBER	DRIVER
0	VESA
1	CIRRUS LOGIC
2	TRIDENT
3	S3 OLD
4	S3 NEW
5	MX
6	TSENG 3000
7	ATI
8	TSENG 4000
9	CHIPS & TECHNOLOGIES
10	WESTERN DIGITAL
11	BANSHEE
12	NVIDIA RIVA 128, TNT, TNT2
13	MATROX
14	PERMEDIA II
15	INTEL 740

6.1.4 -linear

System will use linear framebuffer (this is not LINUX FRAMEBUFFER) instead of banking mode. This works for some chipset for now and is for LINUX only!

6.1.5 –nommx

Switch off MMX support and testing (does not work with Watcom C compiled version of library).

6.1.6 –nofull

Switch on full redrawing of the window contents.

6.1.7 –svga

Force link with SVGALIB = disable framebuffer

6.1.8 –verbose

Verbose diagnostics output

6.1.9 –safe

Vypne autodekciu grafickej karty, MMX, nastaví rozlíšenie 640x480, nastaví "ukecaný" režim atď. Užitočná funkcia keď sa vyskytnú nejaké problémy.

6.2 Utility

6.2.1 file2asm

This utility converts any file to the assembler text representation and is mainly used to converting ***.bmp** to assembler source. The file name is first argument. Target file has the same file name, nut with extension **nsm** for nasm assembler or **wsm** for Watcom assembler (Watcom assembler format requires second parameter: **-wasm**). NOTE: at the start each assembler file is saved the original file length as **unsigned long**.

6.2.2 textto

Utility for converting between MS-DOS and UNIX CR/LF and LF line formats. Syntax is:

textto [-l –c] file1 file2 ...

Wildcards are possible. Switch **–l** remove all carriage returns from file, **-c** add to the each LF char CR.

6.3 Hot keys

KEY			OPERATION
DOS	QNX	LINUX	
ALT+F4	ALT+F4	SHIFT+F4	Close window(if it is possible)
CTRL+TAB	CTRL+TAB	CTRL+TAB	Switch to the next window
ALT+X	ALT+X	ALT+X	Terminate the application
None	PRTSCR	PRTSCR	Save screen contents to the file *xxxxxxxx.bmp*

6.4 The table of the name predefined keys

Symbol	Description

ESC	Escape
F01	F1 key
F02	F2 key
F03	F3 key
F04	F4 key
F05	F5 key
F06	F6 key
F07	F7 key
F08	F8 key
F09	F9 key
F10	F10 key
F11	F11 key
F12	F12 key
HOME	HOME key
END	END key
CR	ENTER
BACKSP	BACKSPACE key
DEL	DELETE key
KLEFT	LEFT ARROW
KRIGHT	RIGHT ARROW
KUP	UP ARROW
KDOWN	DOWN ARROW
PGUP	PAGE UP key
PGDOWN	PAGE DOWN key
INSERT	INSERT key
TAB	TAB key