

Graphics System (GS)

Copyright © 1997-2017 Ericsson AB. All Rights Reserved.

Graphics System (GS) 1.6
December 1, 2017

Copyright © 1997-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 GS User's Guide

The Graphics System application, *GS*, is a library of routines for writing graphical user interfaces. Programs written using GS work on all Erlang platforms and do not depend upon the underlying windowing system.

1.1 GS - The Graphics System

1.1.1 Introduction

> **Warning:**
>
> GS is not recommended for use in new applications. Instead we recommend WX for applications that need a graphical user interface.
>
> GS is not maintained and we plan to deprecate and remove it from the distribution as soon as possible, maybe already in the next major release (R15).

This section describes the general graphics interface to Erlang. This system was designed with the following requirements in mind:

- a graphics system which is easy to learn
- a graphics system which is portable to many different platforms.

Erlang has been implemented on a wide range of platforms and the graphics system works on all these platforms. Erlang applications can be written towards the same graphics API and the application can run on all supported platforms without modification.

Figure 1.1: Graphics Interface for Erlang

1.1.2 Basic Architecture of GS

The basic building block in the graphics system is the graphical object. Objects are created in a hierarchical fashion where each object has a parent. The most common object types are:

- window
- button
- label
- list box
- frame.

Whenever a new object is created, a unique object identifier is returned. This object identifier makes it possible to configure the object by changing its appearance and behaviour. This configuration of the object is controlled by the *Options*, also known as attributes or properties. These include width and height. Most options have a value of a specified type, but not all.

Whenever an Erlang process creates a graphical object, it is said to own the object. The graphics system must keep track of the owner of every graphical object in order to forward incoming events to the owner-process and kill the appropriate graphics window if the owner process suddenly dies.

Figure 1.2: Owner Process

Events are messages which are sent from the graphical object to the owner-process. The events the owner-process is informed about may include:

- the user has clicked on a button
- the user has entered text into an entry field
- the user has taken some action on the object, like moving the window.

Figure 1.3: Events Delivered to Owner Process

1.2 Interface Functions

1.2.1 Overview

The following interface functions are included with the graphics system:

- `gs:start().` This function starts the graphics server and returns its object identifier. If the graphics server has already been started, it returns its original identifier.
- `gs:stop().` This function stops the graphics server and closes all windows which `gs` has launched. This function is not the opposite of `gs:start/0` because `gs:stop/1` causes all applications to lose the graphics server and the objects created with the `gs` system.
- `gs:create(Objtype, Parent, Options).` This function creates a new object of specified `Objtype` as a child to the specified `Parent`. It configures the object with `Options` and returns the identifier for the object, or `{error,Reason}`.
- `gs:create(Objtype, Name, Parent, Options).` This function is identical to the previously listed function, except that a `Name` is specified to reference the object. `Name` is an atom.
- `gs:destroy(IdOrName).` This function destroys an object and all its children.
- `gs:config(IdOrNeme, Options).` This function configures an object with `Options`. It returns `ok`, or `{error,Reason}`.
- `gs:read(Id_or_Name, OptionKey).` This function reads the value of an object option. It returns the value, or `{error,Reason}`.

The above list contains all the function which are *needed* with the graphics system. For convenience, the following aliases also exist:

- `gs:create(Obttype, Parent).`
- `gs:create(Objtype, Parent, Options).`
- `gs:create(Objtype, Parent, Option).`
- `gs:create(Objtype, Name, Parent, Options).`
- `gs:create(Objtype, Name, Parent, Option).`
- `gs:Objecttype(Parent).`
- `gs:Objecttype(Parent,Options).`
- `gs:Objecttype(Parent, Option).`
- `gs:Objecttype(Name, Parent, Options).`

- `gs:Objecttype(Name, Parent, Option).`
- `gs:config(IdOrName, Option).`

These shorthands can be used as follows:

- `gs:window(gs:start(), {map,true}).`
- `gs:button(W).`
- `gs:config(B,{label,{text,"Hi!"}}).`

The `create_tree/2` function is useful for creating a large hierarchy of objects. It has the following syntax:

```
create_tree(ParentId,Tree) -> | {error,Reason}
```

`Tree` is a list of `Object`, and `Object` is any of the following:

- `{ObjectType,Name,Options,Tree}`
- `{ObjectType,Options,Tree}`
- `{ObjectType,Options}`

The following example constructs a window which contains two objects, a button and a frame with a label:

```
R = [{window,[{map,true}],
      [{button,[{label,{text,"Butt1"}}]},
       {frame,[{y,40}],[{label,[{label,{text,"Lbl1"}}]}]}]}],
gs:create_tree(gs:start(),R).
```

1.2.2 A First Example

The first action required is to start up the graphics server. This operation returns an identifier for the server process, which registers itself under the name `gs`. If a graphics server was already started, its identifier is returned. We can now create objects and configure the behavior and appearance of these objects. When all objects are created and configured in a top level window, we map it on the screen to make it visible. The example below shows how to create a window with a button that says "Press Me".

```
-module(ex1).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([init/0]).

init() ->
    S = gs:start(),
    %% the parent of a top-level window is the gs server
    Win = gs:create(window,S,[{width,200},{height,100}]),
    Butt = gs:create(button,Win,[{label, {text,"Press Me"}}]),
    gs:config(Win, {map,true}),
    loop(Butt).

loop(Butt) ->
    receive
        {gs, Butt, click, Data, Args} ->
            io:format("Hello There~n",[]),
```

```
            loop(Butt)
    end.
```

The following steps were completed in this code:

- start a graphics server
- create a window of specified width and height
- create a button with the text "Press Me"
- map the window on the screen
- enter the event loop.

The event loop is where we receive events from `gs`. In this case, we want to receive a click event from the button. This event is delivered when the user presses the button.

Figure 2.1: "Press Me" Button Example

The Erlang `gs` system includes many examples. All examples in this document can be found in the `doc/users_guide/examples/` directory. In addition, there is an example directory which contains examples of fractal trees, bouncing balls, a color editor, and a couple of other `gs` applications.

1.2.3 Creating Objects

You create an object of a specified type with the `create/3` or the `create/4` function. The difference is that the `create/4` function allows you to assign names to the objects. You can then refer to the object instead of using the object identifier. The two forms of the `create` function look as follows:

```
ObjId = gs:create(Objtype, Parent, Options)
ObjId = gs:create(Objtype, Name, Parent, Options)
```

Examples of built-in object types are:

- window
- frame
- menu
- button
- radio button
- list box.

Objects are created in a hierarchical order. The top level object is the window object which is a container object for most other object types.

Figure 2.2: Hierarchy of Objects

A frame object is like a sub-window but also a container object which can have children objects.

The `create/3` or `create/4` functions return an object identifier, or the tuple `{error, Reason}`. The object identifier uniquely identifies the object within the system. The object identifier is used to:

- reconfigure an object
- identify events from a particular object.

1.2.4 Ownership

The process which creates an object is said to own the object. When a process dies, all objects owned by the process are destroyed. The ownership also means that all events generated by a specific object are delivered to the owner process. The graphics server keeps track of all Erlang processes that create objects. It is therefore able to take appropriate actions if a process should die.

1.2.5 Naming Objects

As shown previously, the `create/4` function can be used to name objects. The name should be a unique atom which is used to reference the object. The advantage of naming objects is that we do not have to pass object identifiers as arguments to the event loop. Instead, we can use the object name in our code. To name objects in the following example, the code gives the name `win1` to the window, and `b1` to the button.

```
-module(ex2).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([init/0]).

init() ->
    S = gs:start(),
    gs:create(window,win1,S,[{width,200},{height,100}]),
    gs:create(button,b1,win1,[{label, {text,"Press Me"}}]),
    gs:config(win1, {map,true}),
    loop().

loop() ->
    receive
        {gs, b1, click, Data, Args} ->
            io:format("Hello World!~n",[]),
            loop()
    end.
```

The name is *local* for the process which creates the object. This means that the name have a meaning only for one process. Different processes can give different objects the same name. When passing references to objects between processes, the object identifier has to be used because names only has a meaning in a process context. If necessary, the object identifier can be retrieved by reading the `id` option.

When using distributed Erlang, objects should be named carefully. A named object always refers to an object in the graphics system on the node where it was created. The syntax `{Name,Node}` should be used when referring to a named object on another node.

The following example receives a canvas object from another node and creates a line named `myline1` that will appear in the canvas. Also, this example demonstrates how to configure the line using the special syntax.

```
foo() ->
  receive
    {gs_obj,Canvas,FromNode} -> ok
  end,
  gs:create(line,myline1,Canvas,[{coords,[{10,10},{20,20}]}]),
  gs:config({myline1,FromNode},[{buttonpress,true}]).
```

Unnamed objects are transparent. For example, a line object can be created from a canvas on another node and then configured as any other object.

```
bar() ->
  receive
    {gs_obj,Canvas,_FromNode} -> ok
  end,
  L = gs:create(line,Canvas,[{coords,[{10,10},{20,20}]}]),
  gs:config(L,[{buttonpress,true}]).
```

1.3 Options

1.3.1 The Option Concept

Each object has a set of options. The options are key-value tuples and the key is an atom. Depending on the option, the value can be any Erlang term. Typical options are: x, y, width, height, text, and color. A list of options should be supplied when an object is created . It is also possible to reconfigure an object with the function `gs:config/2`. The following example shows one way to create a red button with the text "Press Me" on it:

```
Butt = gs:create(button,Win, [{x,10},{y,10}]),
gs:config(Butt, [{width,50},{height,50},{bg,red}]),
gs:config(Butt, [{label, {text,"Press Me"}},{y,20}]),
```

The evaluation order of options is not defined. This implies that the grouping of options shown in the following example is not recommended:

```
Rect = gs:create(rectangle,Can, [{coords,[{10,10},{20,20}]},
                                 {move,{5,5}}]),
```

After the operation, the rectangle can be at position `[{10,10},{20,20}]` or `[{15,15},{25,25}]`. The following example produces a deterministic behaviour:

```
Rect = gs:create(rectangle,Can,[{coords,[{10,10},{20,20}]},
gs:config(Rect,[{move,{5,5}}]),
```

The value of each option can be read individually with the `read/2` function as shown in the following example:

```
Value = gs:read(ObjectId,Option)
```

The next example shows how to read the text and the width options from a button:

```
Text = gs:read(Butt, text),
```

```
Width = gs:read(Butt, width),
```

1.3.2 The Option Tables

Each object is described in terms of its options. The options are listed in a table as is shown in the following example:

{Option,Value}	*Default*	*Description*
{fg, Color}	<unspec>	Foreground color of the object
{map, Bool}	false	Visibility on the screen
...	...	...

Table 3.1: Options

The <unspec> default value means that either `gs` or the back-end provides the default value. For example, the `fg` option can be used as follows:

```
Rect = gs:create(rectangle, Window, [{fg, red}]),
Color = gs:read(Rect, fg),
```

1.3.3 Config-Only Options

Most options are read/write key-value tuples such as `{select,true|false}` and `{map,true|false}`, but some options are by nature write-only, or read-only. For example, buttons can flash for a short time and canvas objects can be moved dx, dy. The following table exemplifies some config-only options:

Config-Only	*Description*
flash	Causes the object to flash for 2 seconds.
raise	Raises the object on top of other overlapping objects.
{move, {Dx, Dy}}	Moves the object relative to its current position.

Table 3.2: Config-Only Options

`gs:config(Button,[flash]),` causes the button to flash.

1.3.4 Read-Only Options

The opposite of config-only options are read-only options. The following table exemplifies some read-only options:

Read-Only	*Return*	*Description*
size	Int	The number of items (entries).

{get, Index}	String	The entry at index `Index`.

Table 3.3: Read-Only Options

`EntryString = gs:read(Listbox,{get, Index}),` is an example.

1.3.5 Data Types

As previously stated, each object is described in terms of its options. This section defines the data types for options.

Anchor|Align.
`n|w|s|e|nw|se|ne|sw|center`

Atom.
An Erlang atom such as `myWay`.

Bool.
`true` or `false`

Color.
`{R,G,B}`, or a the predefined name `red`,`green`, `blue`, `white`, `black`, `grey`, or `yellow`. For example `{0,0,0}` is black and `{255,255,255}` is white.

Cursor.
A mouse cursor, or any of the following: `arrow`, `busy`, `cross`, `hand`, `help`, `resize`, `text`, or `parent`. `parent` has a special meaning, namely that this object will have the same cursor as its `parent`.

FileName.
`FileName` is a string. The file name may include a directory path and should point out a file of a suitable type. The path can be either absolute or relative to the directory from where Erlang was started.

Float.
Any float, for example 3.1415.

Font.
A Font is represented as a two or three tuple:`{Family,Size}` or `{Family,Style,Size}`, where `Style` is `bold`, `italic`, or a combination of those in a list. `Size` is an arbitrary integer. `Family` is a typeface of type `times`, `courier`, `helvetica`, `symbol`, `new_century_schoolbook`, or `screen` (which is a suitable screen font).

Int.
Any integer number, for example 42.

Label.
A label can either be a plain text label `{text, String}`, or an image `{image, FileName}` where `FileName` should point out a bitmap.

String.
An Erlang list of ASCII bytes. For example, `"Hi there"=[72,105,32,116,104,101,114,101]`

Term.
Any Erlang term.

In cases where the type is self-explanatory, the name of the parameter is used. For example, `{move, {Dx,Dy}}`.

1.4 Events

1.4.1 Event Messages

Events are messages which are sent to the owner process of the object when the user interacts with the object in some way. A simple case is the user pressing a button. An event is then delivered to the owner process of the button (the process that created the button). In the following example, the program creates a button object and enables the events click and enter. This example shows that events are enabled in the same way as objects are configured with options.

```
B = gs:create(button,Win, [{click,true},{enter,true}]),
event_loop(B).
```

The process is now ready to receive click and enter events from the button. The events delivered are always five tuples and consist of:

```
{gs, IdOrName, EventType, Data, Args}
```

- `gs` is a tag which says it is an event from the `gs` graphics server.
- `IdOrName` contains the object identifier or the name of the object in which the event occurred.
- `EventType` contains the type of event which has occurred. In the example shown, it is either `click` or `enter`.
- `Data` is a field which the user can set to any Erlang term. It is very useful to have the object store arbitrary data which is delivered with the event.
- `Args` is a list which contains event specific information. In a motion event, the Args argument would contain the x and y coordinates.

There are two categories of events:

- *generic events*
- *object specific events*.

1.4.2 Generic Events

Generic events are the same for all types of objects. The following table shows a list of generic event types which the graphics server can send to a process. For generic events, the `Args` argument always contains the same data, independent of which object delivers it.

The following sub-sections explains the event types and what they are used for.

Event	*Args*	*Description*
buttonpress	[ButtonNo,X,Y|_]	A mouse button was pressed over the object.
buttonrelease	[ButtonNo,X,Y|_]	A mouse button was released over the object.
enter	[]	Delivered when the mouse pointer enters the objects area.
focus	[Int|_]	Keyboard focus has changed. 0 means lost focus. 1 means gained focus.
keypress	[KeySym,Keycode, Shift, Control|_]	A key has been pressed.
leave	[]	Mouse pointer leaves the object.

motion	[X,Y|_]	The mouse pointer is moving in the object. Used when tracking the mouse in a window.

Table 4.1: Generic Event Types

The Buttonpress and Buttonrelease Events

These events are generated when a mouse button is pressed or released inside the object frame of a window, or frame object type. The button events are not object specific (compare to click). The format of the buttonpress event is:

```
{gs,ObjectId,buttonpress,Data,[MouseButton,X,Y|_]}
```

The mouse button number which was pressed is the first argument in the `Args` field list. This number is either 1, 2 or 3, if you have a three button mouse. The `X` and `Y` coordinates are sent along to track in what position the user pressed down the button. These events are useful for programming things like "rubberbanding", which is to draw out an area with the mouse. In detail, this event can be described as pressing the mouse button at a specific coordinate and releasing it at another coordinate in order to define a rectangular area. This action is often used in combination with motion events.

The Enter and Leave Events

These events are generated when the mouse pointer (cursor) enters or leaves an object.

The Focus Event

The focus event tracks which object currently holds the keyboard focus. Only one object at a time can hold the keyboard focus. To have the keyboard focus means that all keypresses from the keyboard will be delivered to that object. The format of a focus event is:

```
{gs,ObjectId,focus, Data,[FocusFlag|_]}
```

The FocusFlag argument is either 1, which means that the object has gained keyboard focus, or 0, which means that the object has lost keyboard focus.

The Keypress Event

This event is generated by an object which receives text input from the user, like entry objects. It can also be generated by window objects. The format of a keypress event is:

```
{gs,ObjectId,keypress,Data,[Keysym,Keycode,Shift,Control|_]}
```

The `Keysym` argument is either the character key which was pressed, or a word which describes which key it was. Examples of `Keysyms` are; a,b,c.., 1,2,3..., `'Return'`, `'Delete'`, `'Insert'`, `'Home'`, `'BackSpace'`, `'End'`. The `Keycode` argument is the keycode number for the key that was pressed. Either the `Keysym` or the `Keycode` argument can be used to find out which key was pressed. The `Shift` argument contains either a 0 or a 1

to indicate if the Shift key was held down when the character key was pressed. The Control argument is similar to the Shift key argument, but applies to the Control key instead of the Shift key.

The Motion Event

The motion event is used to track the mouse position in a window. When the user moves the mouse pointer (cursor) to a new position a motion event is generated. The format of a motion event is:

```
{gs,ObjectId,motion,Data,[X,Y|_]}
```

The current x and y coordinates of the cursor are sent along in the `Args` field.

1.4.3 Object Specific Events

The click and doubleclick events are the object specific event types. Only some objects have these events and the `Args` field of the events vary for different type of objects. A click on a check button generates a click event where the data field contains the on/off value of the indicator. On the other hand, the click event for a list box contains information on which item was chosen.

Event	*Args*	*Description*
click	<object specific>	Pressing a button or operating on a object in some predefined way.
double-click	<object specific>	Pressing the mouse button twice quickly. Useful with list boxes.

Table 4.2: Object Specific Events

1.4.4 Matching Events Against Object Identifiers

Events can be matched against the object identifier in the receive statement. The disadvantage of matching against identifiers is that the program must pass the object identifiers as arguments to the event loop.

```
-module(ex3).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([init/0]).

init() ->
    S = gs:start(),
    W = gs:create(window,S,[{width,300},{height,200}]),
    B1 = gs:create(button,W,[{label, {text,"Button1"}},{y,0}]),
    B2 = gs:create(button,W,[{label, {text,"Button2"}},{y,40}]),
    gs:config(W, {map,true}),
    loop(B1,B2).

loop(B1,B2) ->
    receive
        {gs,B1,click,_Data,_Arg} -> % button 1 pressed
            io:format("Button 1 pressed!~n",[]),
```

```
            loop(B1,B2);
        {gs,B2,click,_Data,_Arg} -> % button 2 pressed
            io:format("Button 2 pressed!~n",[]),
            loop(B1,B2)
    end.
```

1.4.5 Matching Events Against Object Names

Another solution is to name the objects using the `create/4` function. In this way, the program does not have to pass any parameters which contain object identifiers for each function call made.

```
-module(ex4).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([init/0]).

init() ->
    S = gs:start(),
    gs:create(window,win1,S,[{width,300},{height,200}]),
    gs:create(button,b1,win1,[{label, {text,"Button1"}},{y,0}]),
    gs:create(button,b2,win1,[{label, {text,"Button2"}},{y,40}]),
    gs:config(win1, {map,true}),
    loop(). %% look, no args!

loop() ->
    receive
        {gs,b1,click,_,_} -> % button 1 pressed
            io:format("Button 1 pressed!~n",[]),
            loop();
        {gs,b2,click,_,_} -> % button 2 pressed
            io:format("Button 2 pressed!~n",[]),
            loop()
    end.
```

1.4.6 Matching Events Against the Data Field

A third solution is to set the `data` option to some value and then match against this value. All built-in objects have an option called `data` which can be set to any Erlang term. For example, we could set the data field to a tuple `{Mod, Fun,Args}` and have the receiving function make an `apply` on the contents of the data field whenever certain events arrive.

```
-module(ex5).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0, init/0, b1/0, b2/0]).

start() ->
    spawn(ex5, init, []).

init() ->
    S = gs:start(),
    W = gs:create(window,S,[{map,true}]),
    gs:create(button,W,[{label,{text,"Button1"}},{data,{ex5,b1,[]}},{y,0}]),
    gs:create(button,W,[{label,{text,"Button2"}},{data,{ex5,b2,[]}},{y,40}]),
```

```
    loop().

loop()->
    receive
        {gs,_,click,{M,F,A},_} -> % any button pressed
            apply(M,F,A),
            loop()
    end.

b1() ->
    io:format("Button 1 pressed!~n",[]).
b2() ->
    io:format("Button 2 pressed!~n",[]).
```

1.4.7 Experimenting with Events

A good way of learning how events work is to write a short demo program like the one shown below and test how different events work.

```
-module(ex6).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0,init/0]).

start() ->
    spawn(ex6,init,[]).

init() ->
    S = gs:start(),
    W = gs:create(window,S,[{map,true},{keypress,true},
                            {buttonpress,true},{motion,true}]),
    gs:create(button,W,[{label,{text,"PressMe"}},{enter,true},
                        {leave,true}]),
    event_loop().

event_loop() ->
    receive
        X ->
            io:format("Got event: ~w~n",[X]),
            event_loop()
    end.
```

1.5 Fonts

1.5.1 The Font Model

Text related objects can be handled with the font option `{font,Font}`. A `Font` is represented as a two or three tuple:

- `{Family,Size}`
- `{Family,Style,Size}`

Examples of fonts are: `{times,12}`, `{symbol,bold,18}`, `{courier,[bold,italic],6}`, `{screen,12}`.

The most important requirement with the font model is to ensure that there is always a "best possible" font present. For example, if an application tries to use the font `{times,17}` on a computer system which does not have this font available, the `gs` font model automatically substitutes `{times,16}`.

Note that GS requires that the following fonts are available if using an X-server display:

- fixed
- -*-courier-*
- -*-times-*
- -*-helvetica-*
- -*-symbol-*
- "-*-new century schoolbook-"
- -*-screen-*

To find out which font is actually chosen by the `gs`, use the option `{choose_font,Font}`. For example, the following situation might occur:

```
1> <input>G=gs:start().</input>
{1,<0.20.0>}
2><input>gs:read(G,{choose_font,{times,38}}).</input>
{times,[],38}
3> <input>gs:read(G,{choose_font,{screen,italic,6}}).</input>
{courier,italic,6}
4>
```

When programming with fonts, it is often necessary to find the size of a string which uses a specific font. `{font_wh,Font}` returns the width and height of any string and any font. The following example illustrates its usage:

Figure 5.1: Font Examples

```
-module(ex15).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/3 $ ').

-export([start/0,init/0]).

start() -> spawn(ex15, init, []).

init() ->
    I=gs:start(),
    Win=gs:create(window, I,
                  [{width, 400},{height, 250},
                   {title,"Font Demo"},{map, true}]),
    E = gs:create(canvas, can1,Win,
    [{x,0},{y, 0},{width,400},{height,250}]),
    Fonts = [{times,19},{screen,16},{helvetica,bold,21},
      {symbol,12},{times,[bold,italic],33},{courier,6}],
    show_fonts_in_boxes(Fonts,0),
    receive
 {gs,_Id,destroy,_Data,_Arg} -> bye
    end.

show_fonts_in_boxes([],_) -> done;
```

```
show_fonts_in_boxes([Font|Fonts],Y) ->
    Txt = io_lib:format("Hi! ~p",[Font]),
    {Width,Height} = gs:read(can1,{font_wh,{Font,Txt}}),
    Y2=Y+Height+2,
    gs:create(rectangle,can1,[{coords,[{0,Y},{Width,Y2}]}]),
    gs:create(text,can1,[{font,Font},{text,Txt},{coords,[{0,Y+1}]}]),
    show_fonts_in_boxes(Fonts,Y2+1).
```

1.6 Default Values

1.6.1 The Default Value Model

When a new object is created, a set of options is provided by the application. Options which are not explicitly given are taken care of by the parent (the container object).

```
B=gs:create(button,Win,[{x,0},{label,{text,"press Me"}}]).
```

In the example shown above, the window provides default values for options like location and background color. If an application cannot use the default values provided by GS, new ones can be configured. For example, the following code creates a red button at location y=30.

```
gs:config(Win,[{default,button,{y,30}},
{default,button,{font,{courier,18}}}]),
B=gs:create(button,Win,[{x,0},{label,{text,"press Me"}}]).
```

The syntax for the default option is `{default,Objecttype,{Option,DefaultValue}}`, where `Objecttype` is the name of any GS object. The special keywords `all` or `buttons` which denote button, radio button, and check button can be used.

The semantics for the default option can be expressed as follows: If an object of kind `Objecttype` is created and no value for `Option` is given, then use `DefaultValue` as the value. Only options of `{Key,Value}` syntax can be given a default values. Default values may be inherited in several steps. In the following example, the button will show the text "Cancel".

```
gs:config(Win,[{default,button,{label,{text,"Cancel"}}}]),
F=gs:create(frame,Win,[]),
B=gs:create(button,F,[]).
```

Default values are inherited so that changed default values only affect new objects, not existing objects.

Default values only have meaning when creating child objects, since objects which cannot have children cannot have default options. An example is buttons.

The following example illustrates how default options can be used:

Figure 6.1: Example of Default Options

```
-module(ex16).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/3 $ ').

-export([start/0,init/0]).

start() -> spawn(ex16, init, []).

init() ->
    I=gs:start(),
    Win=gs:create(window, I,
                  [{width, 200},{height, 200},
                   {title,"Default Demo"},{map, true}]),
    gs:create(canvas, can1,Win,
       [{x,0},{y, 0},{width,200},{height,200},
        {default,text,{font,{courier,bold,19}}},
        {default,text,{fg,blue}},
        {default,rectangle,{fill,red}},{default,text,{text,"Pow!"}},
        {default,oval,{fill,green}}]),
    {A,B,C} = erlang:now(),
    random:seed(A,B,C),
    loop().

loop() ->
    receive
 {gs,_Id,destroy,_Data,_Arg} -> bye
    after 500 ->
     XY = {random:uniform(200),random:uniform(200)},
     draw(random:uniform(3),XY),
     loop()
    end.

draw(1,XY) ->
    gs:create(text,can1,[{coords,[XY]}]);
draw(2,XY) ->
    XY2 = {random:uniform(200),random:uniform(200)},
    gs:create(rectangle,can1,[{coords,[XY,XY2]}]);
draw(3,XY) ->
    XY2 = {random:uniform(200),random:uniform(200)},
    gs:create(oval,can1,[{coords,[XY,XY2]}]).
```

1.7 The Packer

1.7.1 The Packer

This section describes the geometry manager in GS.

When the user resizes a window, the application normally has to resize and move the graphical objects in the window to fit its new size. This can be handled by a so called *packer* or *geometry manager*. In GS, the packer functionality is a property of the `frame` object. A frame with the packer property may control the size and position of its children.

A packer frame organises its children according to a grid pattern of rows and columns. Each row or column has a stretching property associated to it. Some columns may expand more than others and some may have a fixed size. The grid pattern is in itself invisible, but the objects contained by it snap to fit the grid.

The packer controlled by the following options:

Frame options:
`{packer_x,Packlist}` where `Packlist` is list() of `PackOption`, and
`{packer_y,Packlist}` where `Packlist` is list() of `PackOption`.

`PackOption` is:
`{stretch, Weight}` where `Weight` is integer() > 0, or
`{stretch, Weight, MinPixelSize, or}`
`{stretch, Weight, MinPixelSize, MaxPixelSize}, or`
`{fixed, PixelSize}`

A `Weight` is a relative number that specifies how much of the total space of the frame a row or column will get. If the frame has three columns with the weights 2, 1, 3 it tells the geometry manager that the first column should have 2/6, the second 1/6 and the third 3/6 of the space.

Note that giving a minimum or maximum width of one or more columns will change the relation and the way the space is divided.

Then the objects contained by the frame use the following options to position themselves in the grid:
`{pack_x,Column}` where `Column` is integer(), or
`{pack_x,{StartColumn,EndColumn}}`

and
`{pack_y,row}` where `row` is integer(), or
`{pack_y,{Startrow,Endrow}}`

or, the the following option is a convenient shorthand:
`{pack_xy,{Column,row}}`

Consider the following example.

```
-module(ex17).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/1 $ ').

-export([start/0,init/0]).

start() -> spawn(ex17, init, []).

init() ->
    WH = [{width,200},{height,300}],
    Win = gs:window(gs:start(),[{map,true},{configure,true},
    {title,"Packer Demo"}|WH]),
    gs:frame(packer,Win,[{packer_x,[{stretch,1,50},{stretch,2,50},
        {stretch,1,50}]},
    {packer_y,[{fixed,30},{stretch,1}]}]),
    gs:button(packer,[{label,{text,"left"}},{pack_xy,{1,1}}]),
    gs:button(packer,[{label,{text,"middle"}},{pack_xy,{2,1}}]),
    gs:button(packer,[{label,{text,"right"}},{pack_xy,{3,1}}]),
    gs:editor(packer,[{pack_xy,{{1,3},2}},{vscroll,true},{hscroll,true}]),
    gs:config(packer,WH), % refresh to initial size
    loop().

loop() ->
    receive
 {gs,_Id,destroy,_Data,_Arg} -> bye;
 {gs,_Id,configure,_Data,[W,H|_]} ->
     gs:config(packer,[{width,W},{height,H}]), % repack
     loop();
 Other ->
     io:format("loop got: ~p~n",[Other]),
     loop()
    end.
```

It defines a frame with three columns where the second should be twice as wide as the other but no column should be smaller than 50 pixels wide. The frame has two rows where the first has a fixed height of 30 pixels and the last row is totally flexible. Three buttons are placed next to each other on the first row, and below them an editor. The editor covers all three columns.

Figure 7.1: Frame with three columns

The picture below illustrates what happens when the window is resized.

Figure 7.2: Resized Frame

To repack the objects, the size of the packer frame has to be set explicitly. This is done by using the height and width options as usual. Since the packer frame controls the size of its children, using the standard x, y, width, height options, packer frames may be nested recursively.

The packer is very useful since it simplifies the programming. The programmer will not have to spend time fine tuning x, y, width, height of each object, since these options are handled by the frame.

1.8 Built-In Objects

1.8.1 Overview

This section describes the built-in objects of the graphics interface. The following objects exist:

Window
: An ordinary window.

Button
: A simple press button.

Checkbutton
: A button with a check-mark indicator.

Radiobutton
: A button with an indicator that has an only-one-selected-at-a-time property.

Label
: Shows a text or bitmap.

Frame
: A plain container object. It is used for logical and visual grouping of objects.

Entry
: A one-line object for entering text.

Listbox
: A list of text strings.

Canvas
: A drawing area which contains light-weight objects such as rectangle, line, etc.

Menu
: A collection of objects for constructing pull-down and pop-up menus.

Grid
: An object for showing tables. A kind of multi-column listbox.

Editor
: A multi-line text editor.

Scale
: To select a value within a range.

Some objects can act as container objects. The following table describes these relationships:

Objects	*Valid Parents*
window	window, gs
buttons, canvas, editor, entry, frame, grid, label, listbox, menubar, scale	frame, window
arc, image, line, oval, polygon, rectangle, text	canvas
menubutton	menubar, window, frame
gridline	grid
menuitem	menu
menu	menubutton, menuitem (with {itemtype, cascade}), window, frame (the last two are for pop-up menus)

Table 8.1: Relations Between Objects and Container Objects

1.8.2 Generic Options

Most objects have a common subset of options and will be referred to as generic options. They apply to most objects.

{Option,Value}	*Default*	*Description*
beep	<unspec>	A beep will sound. Applies to all objects.
{bg, Color}	<unspec>	Background color. Applies to objects which have a background color.
{data, Term}	[]	Always delivered with the event in the data field. Applies to all objects.
{default,Objecttype,{Key,Value}}	<unspec>	Applies to all container objects. Specifies the default value for an option for children of type Objecttype.
{enable, Bool}	true	Objects can be enabled or disabled. A disabled object cannot be clicked on, and text cannot be entered. Applies to buttons, menuitem, entry, editor, scale.
{font, Font}	<unspec>	Applies to all text related objects and the grid.

{fg, Color}	<unspec>	Foreground color. Applies to objects which have a foreground color.
flush	<unspec>	Ensures that front-end and back-end are synchronized. Applies to all objects.
{setfocus, Bool}	<unspec>	Set or remove keyboard focus to this object. Applies to objects which can receive keyboard events.

Table 8.2: Generic Options

The following options apply to objects which can have a *frame* as parent. Coordinates are relative to the parent.

{Option,Value}	*Default*	*Description*
{cursor, Cursor}	parent	The appearance of the mouse cursor.
{height, Int}	<unspec>	The height in pixels.
{pack_x, Column| {StartColumn,EndColumn}}	<unspec>	Packing position. See The Packer section.
{pack_y, row|{Startrow,Endrow}}	<unspec>	Packing position. See The Packer section.
{pack_xy, {Column,row}}	<unspec>	Packing position. See The Packer section.
{width, Int}	<unspec>	The width in pixels.
{x, Int}	<unspec>	The x coordinate within the parent objects frame in pixels. 0 is to the left.
{y, Int}	<unspec>	The y coordinate in pixels. 0 is at the top.

Table 8.3: Generic Options (Frame as Parent)

Config-Only	*Description*
lower	Lowers this object to the bottom in the visual hierarchy.
raise	Lowers this object in the visual hierarchy.

Table 8.4: Generic Config-Only Options

The following table lists generic Read-Only options:

Read-Only	*Return*	*Description*
children	[ObjectId1, ..., ObjectIdN]	All children
{choose_font,Font}	Font	Return the font that is actually used if a particular font is given.
id	ObjectId	Return the object id for this object. Useful if the object is a named object.
{font_wh,{Font,Text}}	{Width,Height}	Return the size of a text in a specified font. It returns the size of the font that is actually chosen by the back-end.
type	Atom	The type of this object.
parent	ObjectId	The parent of this object.

Table 8.5: Generic Read-Only Options

Generic Event Options

The table below lists all generic event options:

{Option,Value}	*Default*
{buttonpress, Bool}	false
{buttonrelease, Bool}	false
{enter, Bool}	false
{leave, Bool}	false
{keypress, Bool}	false
{motion, Bool}	false

Table 8.6: Generic Event Options

1.8.3 Window

The basic object is the window object. It is the most common container object. All graphical applications use at least one (top-level) window.

Figure 8.1: Empty Window titled "A Window".

The following tables show all window specific options:

{Option,Value}	*Default*	*Description*
{bg, Color}	<unspec>	{R,G,B} or a color name
{configure,Bool}	false	Will generate a `configure`event when the window has been resized or moved. The `Args`field contains [Width,Height,X,Y|_]
{destroy,Bool}	true	Will generate a `destroy`event when the window is destroyed from the window manager. All GS applications should handle this event.
{iconname, String}	<unspec>	
{iconify, Bool}	false	
{map, Bool}	false	Make it visible on the screen
{title, String}	<unspec>	The title of the window. The default is the internal widget name which is platform specific.

Table 8.7: Window Options

Config-Only	*Description*
raise	Raise window on top of all other windows.
lower	Lower window to background.

Table 8.8: Window Config-Only Options

The following example shows how to create a window and configure it to enable various events.

```
-module(ex7).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([mk_window/0]).

mk_window() ->
    S= gs:start(),
    Win= gs:create(window,S,[{motion,true},{map,true}]),
    gs:config(Win,[{configure,true},{keypress,true}]),
    gs:config(Win,[{buttonpress,true}]),
    gs:config(Win,[{buttonrelease,true}]),
```

```
    event_loop(Win).

event_loop(Win) ->
    receive
        {gs,Win,motion,Data,[X,Y | Rest]} ->
            %% mouse moved to position X Y
            io:format("mouse moved to X:~w  Y:~w~n",[X,Y]);
        {gs,Win,configure,Data,[W,H | Rest]} ->
            %% window was resized by user
            io:format("window resized W:~w  H:~w~n",[W,H]);
        {gs,Win,buttonpress,Data,[1,X,Y | Rest]} ->
            %% button 1 was pressed at location X Y
            io:format("button 1 pressed X:~w  Y:~w~n",[X,Y]);
        {gs,Win,buttonrelease,Data,[_,X,Y | Rest]} ->
            %% Any button (1-3) was released over X Y
            io:format("Any button released X:~w  Y:~w~n",[X,Y]);
        {gs,Win,keypress,Data,[a | Rest]} ->
            %% key `a' was pressed in window
            io:format("key a was pressed in window~n");
        {gs,Win,keypress,Data,[_,65,1 | Rest]} ->
            %% Key shift-a
            io:format("shift-a was pressed in window~n");
        {gs,Win,keypress,Data,[c,_,_,1 | Rest]} ->
            %% CTRL_C pressed
            io:format("CTRL_C was pressed in window~n");
        {gs,Win,keypress,Data, ['Return' | Rest]} ->
            %% Return key pressed
            io:format("Return key was pressed in window~n")
        end,
    event_loop(Win).
```

1.8.4 Button

Figure 8.2: Radio Buttons, Check Buttons, and Ordinary Button

Buttons are the simplest and the most commonly used objects. You press them and get a click event. The following tables show the options for all button types.

{Option,Value}	*Default*	*Description*
{align, Align}	center	Text alignment within the frame.
{justify, left | center | right}	center	Justification is only valid when there are several lines of text.
{label, Label}	<unspec>	Text or image to show.
{select, Bool}	false	Check buttons and radio buttons. true means that the button is selected.
{underline, Int}	<unspec>	Underline character N to indicate a keyboard accelerator.
{group, Atom}	<unspec>	Radio button: only one per group is selected at one time. Check button:

		All in the same group are selected automatically.
{value, Atom}	<unspec>	Radio buttons only. Groups radio buttons together within a group.

Table 8.9: Options for all Button Types

Config-Only	*Description*
flash	Flash button
invoke	Explicit button press.
toggle	Check buttons only. Toggles select value.

Table 8.10: Config-Only Options for all Button types

Buttontype	*Event*
normal	{gs, itemId, click, Data, [Text| _]}
check	{gs, itemId, click, Data, [Text, Group, Bool | _]}
radio	{gs, itemId, click, Data, [Text, Group, Value | _]}

Table 8.11: >Events for all Button types

Buttons and check buttons are simple to understand, radio buttons are more difficult. Each radio button has a group and a value option. The group option is used to group together two or more radio buttons. Normally, each radio button within a group has a unique value which means that only one radio button can be selected at a time. If two (or more) radio buttons share the same value and one of them is selected, then both will be selected and all others are de-selected. The following short example shows how to program radio button logic in a situation where two of them share the same value.

```
-module(ex8).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0]).

start() ->
    gs:window(win,gs:start(),{map,true}),
    gs:radiobutton(rb1,win,[{label,{text,"rb1"}},{value,a},{y,0}]),
    gs:radiobutton(rb2,win,[{label,{text,"rb2"}},{value,a},{y,30}]),
    gs:radiobutton(rb3,win,[{label,{text,"rb3"}},{value,b},{y,60}]),
    rb_loop().

rb_loop() ->
    receive
        {gs,Any_Rb,click,Data,[Text, Grp, a | Rest]} ->
```

```
            io:format("either rb1 or rb2 is on.~n",[]),
            rb_loop();
        {gs,rb3,click,Data,[Text, Grp, b | Rest]} ->
            io:format("rb3 is selected.~n",[]),
            rb_loop()
    end.
```

Figure 8.3: Radio Button Group with Last Button Selected

The example shown creates three radio buttons which are members of the same group. The default behavior is that all radio buttons created by the same process are members of the same group. Normally, only one in a group may be selected at the same time, but since we defined the value-option to have the same value for `rb1` and `rb2`, they will both be selected/de-selected simultaneously. The normal radio button group behavior is that all radio buttons within the same group have unique default values.

1.8.5 Label

{Option,Value}	*Default*	*Description*
{align,Align}	center	How the text is aligned within the frame.
{justify,left|right|center}	left	How to justify several lines of text.
{label,Label}	<unspec>	Text or image to show.
{underline,Int}	<unspec>	Underline character N to indicate a keyboard accelerator.

Table 8.12: Label Options

A label is a simple text field which is used to display text to the user. It is possible to have several lines of text by inserting newline '\ ' characters between each line. The label object does not automatically adjust its size so that text will fit inside. This has to be done manually, or the text may be clipped at the edges.

1.8.6 Frame

The frame object acts as a container for other objects. Its main use is to logically and visually group objects together. Grouped objects can then be moved, displayed, or hidden in one single operation.

{Option,Value}	*Default*	*Description*
{bw,Int}	<unspec>	Border width
{packer_x,PackList}	<unspec>	Makes the frame pack its children. See the packer section.
{packer_y,PackList}	<unspec>	Makes the frame pack its children. See the packer section.

Table 8.13: Frame Options

It is possible to have frame objects within frame objects so that large hierarchical structures of objects can be created.

1.8.7 Entry

Figure 8.4: Label and Entry Objects for User Input

{Option,Value}	*Default*	*Description*
{justify, left|right|center}	left	Text justification in entry field.
{text, String}	<unspec>	Use this option to initially set some text, and to read the text.

Table 8.14: Entry Options

Entrys are used to prompt the user for text input.

Config-Only	*Description*
{delete, {From, To}}	Deletes the characters within index {From,To}.
{delete, last}	Deletes the last character.
{delete, Index}	Deletes the character at position Index.
{insert, {Index, String}}	Inserts text at the specific character position. Index starts from 0.
{select, {From, To}}	Selects a range.
{select, clear}	De-selects selected text.

Table 8.15: Entry Config-Only Options

A common usage of the entry object is to listen for the 'Return' key event and then read the text field. The following example shows a simple dialog which prompts the user for a name and returns the tuple `{name,Name}` when a name is entered, or `cancel` if the cancel button is pressed.

```
-module(ex9).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0,init/1]).

start() ->
    spawn(ex9, init, [self()]),
    receive
        {entry_reply, Reply} -> Reply
    end.
```

```
init(Pid) ->
    S = gs:start(),
    Win = gs:create(window,S,[{title,"Entry Demo"},
                              {width,150},{height,100}]),
    gs:create(label,Win,[{label,{text,"What's your name?"}},
                         {width,150}]),
    gs:create(entry,entry,Win,[{x,10},{y,30},{width,130},
                               {keypress,true}]),
    gs:create(button,ok,Win,[{width,45},{y,60},{x,10},
                             {label,{text,"Ok"}}]),
    gs:create(button,cancel,Win,[{width,60},{y,60},{x,80},
                                 {label,{text,"Cancel"}}]),
    gs:config(Win,{map,true}),
    loop(Pid).

loop(Pid) ->
    receive
        {gs,entry,keypress,_,['Return'|_]} ->
            Text=gs:read(entry,text),
            Pid ! {entry_reply,{name,Text}};
        {gs,entry,keypress,_,_} -> % all other keypresses
            loop(Pid);
        {gs,ok,click,_,_} ->
            Text=gs:read(entry,text),
            Pid ! {entry_reply,{name,Text}};
        {gs,cancel,click,_,_} ->
            Pid ! {entry_reply,cancel};
        X ->
            io:format("Got X=~w~n",[X]),
            loop(Pid)
    end.
```

The program draws the dialog and waits for the user to either press the return key or click one of the buttons. It then reads the text option of the entry and returns the string to the client process.

1.8.8 Listbox

A listbox is a list of labels with optional scroll bars attached. The user selects one or more predefined alternative entries. You can add and remove entries in the listbox. The first element in a listbox has index 0.

{Option,Value}	*Default*	*Description*
{hscroll, Bool | top | bottom}	true	Horizontal scroll bar.
{items, [String, String ... String]}	<unspec>	All items (entries) in the listbox.
{scrollbg, Color}	<unspec>	Foreground color of scroll bar.
{scrollfg, Color}	<unspec>	Background color of scroll bar.
{selectmode, single | multiple}	single	Controls if it is possible to have several items selected at the same time.
{vscroll, Bool | left | right}	true	Vertical scroll bar.

Table 8.16: Listbox Options

Config-Only	*Description*
{add, {Index, String}}	Add an item at specified index.
{add, String}	Add an item last.
{change, {Index,String}}	Change one item.
clear	Delete all items.
{del, Index | {From, To}}	Delete an item at specified index, or all from index From to index To.
{see, Index}	Make the item at specified index visible.
{selection, Index | {From,To}| clear}	Select an item (highlight it). Clear erases the selection.

Table 8.17: Listbox Cinfig-only Options

Read-Only	*Return*	*Description*
selection	ListOfStrings	Returns current selection. All selected item indices will be returned in a list.
size	Int	The number of items (entries) in the listbox.
{get, Index}	String	Returns item at specified index.

Table 8.18: Listbox Read-Only Options

Event
{gs, ListBox, click, Data, [Index, Text,Bool | _]}
{gs, ListBox, doubleclick, Data, [Index, Text,Bool | _]}

Table 8.19: Listbox Events

`Bool` is true if object is selected, false if de-selected.

Note that `click` and `doubleclick` are two discrete events: if you have subscribed to both, you will receive both a `click` event and a `doubleclick` event when double-clicking on one item (since two rapid clickings are regarded as both a `click` and a `doubleclick`). The subscription of `doubleclick` events does `not` result in the `click` events being unsubscribed!

The following example shows a simple application which prompts the user for a text item. The user has the following options:

- browse the items and then double-click the required item

- type the name into the entry field and then press the Return key
- select the required item and then click the OK button.

Figure 8.5: Simple Browser Dialog

```
-module(ex10).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0, init/3]).

start() ->
    start("Pick a fruit:",
          [apple, banana, lemon, orange, strawberry,
           mango, kiwi, pear, cherry,pineapple,peach,apricot]).

start(Text,Items) ->
    spawn(ex10,init,[self(),Text,Items]),
    receive
        {browser,Result} -> Result
    end.

init(Pid,Text,Items) ->
    S=gs:start(),
    Win=gs:window(S,[{width,250},{height,270},
                     {title,"Browser"}]),
    Lbl=gs:label(Win,[{label,{text,Text}},{width,250}]),
    Entry=gs:entry(Win,[{y,35},{width,240},{x,5},
                        {keypress,true},
                        {setfocus,true}]),
    Lb=gs:listbox(Win,[{x,5},{y,65},{width,160},
                       {height,195},{vscroll,right},
                       {click,true},{doubleclick,true}]),
    Ok=gs:button(Win,[{label,{text,"OK"}},
                      {width,40},{x,185},{y,175}]),
    Cancel=gs:button(Win,[{label,{text,"Cancel"}},
                          {x,175},{y,225},{width,65}]),
    gs:config(Lb,[{items,Items}]),
    gs:config(Win,{map,true}),
    browser_loop(Pid,Ok,Cancel,Entry,Lb).

browser_loop(Pid,Ok,Cancel,Entry,Lb) ->
    receive
        {gs,Ok,click,_,_} ->
            Txt=gs:read(Entry,text),
            Pid ! {browser,{ok,Txt}};
        {gs,Cancel,click,_,_} ->
            Pid ! {browser,cancel};
        {gs,Entry,keypress,_,['Return'|_]} ->
            Txt=gs:read(Entry,text),
            Pid ! {browser,{ok,Txt}};
        {gs,Entry,keypress,_,_} ->
            browser_loop(Pid,Ok,Cancel,Entry,Lb);
        {gs,Lb,click,_,[Idx, Txt|_]} ->
            gs:config(Entry,{text,Txt}),
            browser_loop(Pid,Ok,Cancel,Entry,Lb);
        {gs,Lb,doubleclick,_,[Idx, Txt|_]} ->
            Pid ! {browser,{ok,Txt}};
        {gs,_,destroy,_,_} ->
            Pid ! {browser,cancel};
```

```
      X ->
          io:format("Got X=~w~n",[X]),
          browser_loop(Pid,Ok,Cancel,Entry,Lb)
  end.
```

1.8.9 Canvas

The canvas object is a simple drawing area. The user can draw graphical objects and move them around the drawing area. The canvas also has optional scroll bars which can be used to scroll the drawing area. The graphical objects that can be created on a canvas object are:

- arc
- image
- line
- oval
- polygon
- rectangle
- text.

These objects must have a canvas object as a parent, but they are otherwise similar to all other basic objects. The following tables show the options which apply to canvas objects.

{Option,Value}	*Default*	*Description*
{bg, Color}	<unspec>	Color of the drawing area.
{hscroll, Bool | top | bottom}	false	Horizontal scroll bar.
{scrollbg, Color}	<unspec>	Foreground color of scroll bar.
{scrollfg, Color}	<unspec>	Background color of scroll bar.
{scrollregion, {X1,Y1,X2,Y2}}	<unspec>	The size of the drawing area to be scrolled.
{vscroll, Bool | left | right}	false	Vertical scroll bar.

Table 8.20: Canvas Options

Read-Only	*Return*	*Description*
{hit, {X,Y}}	list of ObjectId	Returns the canvas objects at X,Y.
{hit, [{X1,Y1},{X2,Y2}]}	list of ObjectId	Returns the canvas objects which are hit by the rectangle.

Table 8.21: Canvas Read-Only Options

Canvas objects have the same types of events as other objects. The following Config-Only options also apply to canvas objects:

Config-Only	*Description*
lower	Lowers the object.
{move, {Dx, Dy}}	Moves object relative to its current position.
raise	Raises the object above all other objects.

Table 8.22: Canvas Config-Only Options

The following sections describe the graphical objects which can be drawn on a canvas object.

The Canvas Arc Object

The canvas arc object is defined within a rectangle and is drawn from a start angle to the extent angle. Origo is in the center of the rectangle.

Figure 8.6: Canvas Arc Object

```
gs:create(arc,Canvas,[{coords,[{10,10},{80,80}]},{fill,yellow}]).
```

{Option,Value}	*Default*	*Description*
{bw, Int}	1	Defines the width.
{coords, [{X1,Y1},{X2,Y2}]}		Defines a rectangle to draw the arc within.
{extent, Degrees}		
{fg, Color}		
{fill, Color|none}	none	Defines fill color of arc object.
{start, Degrees}		
{style, arc}		No line segments.
{style, chord}		A single line segment connects the two end points of the perimeter section.
{style, pieslice}	This Style	Two lines are drawn between the center of the oval and each end of the perimeter section.

Table 8.23: Canvas Arc Options

The Canvas Image Object

The canvas image object displays images and moves them around in a simple way. The currently supported image formats are bitmap and gif.

Figure 8.7: Canvas Image Object

```
gs:create(image,Canvas,[{load_gif,"brick.gif"}]).
```

{Option,Value}	*Default*	*Description*
{anchor, Anchor}	nw	Anchor reference specified by {X,Y} .
{bg, Color}	<unspec>	Background color. Pixel value 0.
{bitmap, FileName}	<unspec>	A bitmap file which contains a bmp bitmap.
{coords, [{X,Y}]}	<unspec>	Position on the canvas.
{fg, Color}	<unspec>	Foreground color. Pixel value 1.
{load_gif, FileName}	<unspec>	Loads a gif image.

Table 8.24: Canvas Image Object Options

The Canvas Line Object

Figure 8.8: Line Object Drawn on a Canvas

```
gs:create(line,Canvas,
          [{coords,[{25,25},{50,50},{50,40},{85,75}]},
           {arrow,last},{width,2}]).
```

{Option,Value}	*Default*	*Description*
{arrow, both | none | first | last}	none	Draws arrows at the end points of the line.
{coords, [{X1,Y1},{X2,Y2}, ... {Xn,Yn}]}	<unspec>	A list of coordinates. The line will be drawn between all pairs in the list.
{fg, Color}	<unspec>	The color of the line.

{smooth, Bool}	false	Smoothing with Bezier splines.
{splinesteps, Int}	<unspec>	
{width, Int}	1	The width of the line.

Table 8.25: Canvas Line Object Options

The Canvas Oval Object

Figure 8.9: Oval Object Drawn on a Canvas

```
gs:create(oval,Canvas,
          [{coords,[{25,25},{125,75}]},{fill,red},{bw,2}]).
```

{Option,Value}	*Default*	*Description*
{bw, Int}	1	Width.
{coords, [{X1,Y1},{X2,Y2}]}	<unspec>	Bounding rectangle which defines shape of object.
{fg, Color}		
{fill, Color|none}	none	Object fill color.

Table 8.26: Canvas Oval Object Options

The Canvas Polygon Object

Figure 8.10: Canvas Polygon Object

```
gs:create(polygon,Canvas,
          [{coords,[{10,10},{50,50},{75,30}]}]).
```

{Option,Value}	*Default*	*Description*
{bw, Int}	1	Width.
{coords, [{X1,Y1},{X2,Y2} | {Xn,Yn}]}	<unspec>	Defines all points in the polygon. There may be any number of points in the polygon.
{fg, Color}	black	The color of the polygon outline.

{fill, Color|none}	none	
{smooth, Bool}	false	Smoothing with Bezier splines.
{splinesteps, Int}	<unspec>	

Table 8.27: Canvas Polygon Object Options

The Canvas Rectangle Object

Figure 8.11: Rectangle Object Created on a Canvas

```
gs:create(rectangle,Canvas,
          [{coords,[{30,30},{70,70}]},{fill,cyan},{bw,2}]).
```

{Option,Value}	*Default*	*Description*
{bw, Int}	1	The width of the border line.
{coords, [{X1,Y1},{X2,Y2}]}	<unspec>	Defines rectangle coordinates.
{fg, Color}	<unspec>	The color of the border line.
{fill, Color|none}	none	Fill color of rectangle.

Table 8.28: Canvas Rectangle Object Options

The Canvas Text Object

Figure 8.12: Canvas Text Object

```
gs:create(text,C,[{coords,[{50,50}]},
                  {font,{times,18}},
                  {fg,red},
                  {text,"Hello World!"}]).
```

{Option,Value}	*Default*	*Description*
{anchor, Anchor}	nw	Anchor reference specified by {X,Y}.
{coords, [{X, Y}]}	<unspec>	Position in the canvas.

{fg, Color}	<unspec>	Text color (background color is the canvas color).
{justify, left | center | right}	<unspec>	Tex justification. Only valid with several lines of text.
{text, String}	<unspec>	The text string to display.
{width, Int}		The width in pixels. The text will be wrapped into several lines to fit inside the width.

Table 8.29: Canvas Text Object Options

1.8.10 Menu

Menus consist of four object types:

- the menu bar
- the menu button
- the menu
- the menu item.

Menu Bar

The menu bar is a simple object. It is placed at the top of the window and contains menu items. {x,y} or width cannot be controlled since, by definition, the menu bar is placed at the top of the window.

{Option,Value}	*Default*	*Description*
<only generic options>		

Table 8.30: Menu Bar Options

Menu Button

The menu button displays a menu when pressed. The width of the menu button is automatically determined by the size of the text.

{Option,Value}	*Default*	*Description*
{align, Align}	center	Text alignment within the frame.
{justify, left | center | right}	center	Justification is only valid when there are several lines of text.
{label,{text,Text}}	<unspec>	
{side, left | right}	<unspec>	Placement on the menu bar. The menu button created first will have the left/right position.

{underline, Int}	<unspec>	Underline character N to indicate an keyboard accelerator.

Table 8.31: Menu Button Options

Menu

The menu contains menu items, which are displayed vertically. Its width is automatically determined by the width of the menu items it contains.

{Option,Value}	*Default*	*Description*
{selectcolor, Color}	<unspec>	The indicator color of radio buttons and check buttons.

Table 8.32: Menu Options

Config-Only	*Description*
{post_at,{X,Y}}	Displays the menu as a pop-up menu at {X,Y} (coordinate system of the parent).

Table 8.33: Menu Config-Only Options

Menu Item

The menu item is an object of its own. It can send events when the user selects it.

{Option,Value}	*Default*	*Description*
{group, Atom}	<unspec>	For {type, radio|check}.
{itemtype, type}	normal	The type of this item. Cannot be reconfigured.
{label, {text,Text}}	<unspec>	The text of the item.
{underline, Int}	<unspec>	Underline character N to indicate an keyboard accelerator.
{value, Atom}	<unspec>	

Table 8.34: Menu Item Options

```
type: normal | separator | check | radio | cascade
```

itemtype	*Event*

normal	{gs, itemId, click, Data, [Text, Index | _]}
check	{gs, itemId, click, Data, [Text, Index, Group, Bool| _]}
radio	{gs, itemId, click, Data, [Text, Index, Group, Value| _]}

Table 8.35: Menu Item Events

Read-Only	*Return*	*Description*
index	Int	Index in the menu. Starts counting from 0.

Table 8.36: Menu Item Read-Only Options

Menu Demo

Figure 8.13: Simple Menu

The following example shows a short demo of the gs menus:

```
-module(ex13).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0,init/0]).

start() -> spawn(ex13, init, []).

init() ->
    I=gs:start(),
    Win=gs:window(I, [{width,200},{height,100},
                      {title,"menu"},{map, true}]),
    Bar = gs:create(menubar, Win, []),
    Fmb = gs:create(menubutton, Bar,
                    [{label,{text,"File"}}]),
    Emb = gs:create(menubutton, Bar,
                    [{label,{text,"Edit"}}]),
    Hmb = gs:create(menubutton, Bar,
                    [{label,{text,"Help"}},{side,right}]),
    Fmnu = gs:create(menu, Fmb, []),
    Emnu = gs:create(menu, Emb, []),
    Hmnu = gs:create(menu, Hmb, []),
    gs:create(menuitem, load, Fmnu,
              [{label,{text, "Load"}}]),
    gs:create(menuitem, save, Fmnu,
              [{label,{text, "Save"}}]),
    Exit = gs:create(menuitem, Fmnu,
                     [{label,{text, "Exit"}}]),
    Color = gs:create(menuitem, Emnu,
                      [{label,{text, "Color"}},
                       {itemtype, cascade}]),
    Cmnu = gs:create(menu, Color, [{disabledfg,gray}]),
```

```
    gs:create(menuitem, Cmnu, [{label, {text,"Red"}},
                               {data, {new_color, red}},
                               {itemtype,radio},{group,gr1}]),
    gs:create(menuitem, Cmnu, [{label, {text,"Blue"}},
                               {data, {new_color, blue}},
                               {itemtype,radio},{group,gr1}]),
    gs:create(menuitem,Cmnu, [{label, {text,"Black"}},
                              {data, {new_color, black}},
                              {itemtype,radio},{group,gr1}]),
    Y = gs:create(menuitem, Hmnu, [{label, {text,"You"}},
                                   {itemtype, check}]),
    M = gs:create(menuitem, me, Hmnu, [{label, {text, "Me"}},
                                       {itemtype, check}]),
    gs:create(menuitem, Hmnu, [{itemtype, separator}]),
    gs:create(menuitem, Hmnu, [{label, {text, "Other"}},
                               {itemtype, check},
                               {enable,false}]),
    gs:create(menuitem, doit, Hmnu, [{label, {text, "Doit!"}},
                                     {data, {doit, Y, M}}]),
    loop(Exit, Win).

loop(Exit, Win) ->
    receive
        {gs, save, click, _Data, [Txt, Index | Rest]} ->
            io:format("Save~n");
        {gs, load, click, _Data, [Txt, Index | Rest]} ->
            io:format("Load~n");
        {gs, Exit, click, _Data, [Txt, Index | Rest]} ->
            io:format("Exit~n"),
            exit(normal);
        {gs, _MnuItem, click, {new_color, Color}, Args} ->
            io:format("Change color to ~w. Args:~p~n",
                      [Color, Args]),
            gs:config(Win, [{bg, Color}]);
        {gs, doit, click, {doit, YouId, MeId}, Args} ->
            HelpMe = gs:read(MeId, select),
            HelpYou = gs:read(YouId, select),
            io:format("Doit. HelpMe:~w, HelpYou:~w, Args:~p~n",
                      [HelpMe, HelpYou, Args]);
        Other -> io:format("Other:~p~n",[Other])
    end,
    loop(Exit, Win).
```

1.8.11 Grid

The grid object is similar to the listbox object. The main difference is that the grid is a multi-column object which is used to display tables. If needed, the grid can send click events when a user presses the mouse button in a table cell. Although the grid has a behavior which is similar to the listbox, the programming is somewhat different. The data in a table cell is represented as a pure `gs` object and can be treated as such. This object is called a grid line. It is located at a row in the parent grid. If a grid line is clicked, it sends an event to its owner.

Grid Line

{Option,Value}	*Default*	*Description*
{{bg, Column},Color}	<unspec>	The background color of a cell.
{bg, {Column,Color}}	<unspec>	Equivalent to {{bg, Column},Color}.

{bg, Color}	<unspec>	The background color of all cells.
{click, Bool}	true	Turns click events on/off.
{doubleclick, Bool}	false	Turns double-click events on/off.
{{fg, Column},Color}	<unspec>	The foreground color of a cell.
{fg, {Column,Color}}	<unspec>	Equivalent to {{fg, Column},Color}
{fg,Color}	<unspec>	The foreground color of all cells.
{text, {Column,Text}}	<unspec>	The text in the cell.
{{text, Column},Text}	<unspec>	Equivalent to {text,{Column,Text}}.
{text,Text}	<unspec>	The text for all cells.
{row, {row}}	<unspec>	The grid row. Must not be occupied by another grid line.

Table 8.37: Grid Line Options

Event
{gs, GridLineId, click, Data, [Col, row, Text | _]}
{gs, GridLineId, doubleclick, Data, [Col, row, Text | _]}

Table 8.38: Gride Line Events

Grid

{Option,Value}	*Default*	*Description*
{font,Font}	<unspec>	A "global" grid font.
{hscroll, Bool|top|bottom}	true	Horizontal scroll bar.
{vscoll, Bool|left|right}	true	Vertical scroll bar.
{rows, {Minrow,Maxrow}}	<unspec>	The rows which are currently displayed.
{columnwidths, [WidthCol1,WidthCol2, ..., WidthColN}}	<unspec>	Defines the number of columns and their widths in coordinates. The size of the columns can be reconfigured, but not the number of columns.

{fg, Color}	<unspec>	The color of the grid pattern and the text.
{bg, Color}	<unspec>	The background color.

Table 8.39: Grid Options

Read-Only	*Return*	*Description*
{obj_at_row, row}	Object |undefined	The grid line at row.

Table 8.40: Grid Read-Only Options

The rows and columns start counting at 1.

Grid Demo

Figure 8.14: Simple Grid

The following simple example shows how to use the grid.

```
-module(ex12).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0,init/0]).

start() -> spawn(ex12, init, []).

init() ->
    R=[{window,[{width,200},{height,200},{title,"grid"},{map, true}],
 {grid, [{x,10},{y,10},{height,180},{width,180},{columnwidths,[80,60]},
  {rows,{1, 20}}],
  [{gridline,[{text,{1,"NAME"}},{text,{2,"PHONE"}},
      {font,{screen,bold,12}},{row,1},{click,false}]},
   {gridline,[{text,{1,"Adam"}},{text,{2,"1234"}},{row,2}]},
   {gridline,[{text,{1,"Beata"}},{text,{2,"4321"}},{row,3}]},
   {gridline,[{text,{1,"Thomas"}},{text,{2,"1432"}},{row,4}]},
   {gridline,[{text,{1,"Bond"}},{text,{2,"007"}},{row,5}]},
   {gridline,[{text,{1,"King"}},{text,{2,"112"}},{row,6}]},
   {gridline,[{text,{1,"Eva"}},{text,{2,"4123"}},{row,7}]}]}}],
    gs:create_tree(gs:start(),R),
    loop().

loop() ->
    receive
 {gs,_Win,destroy,_Data,_Args} -> bye;
 {gs,_Gridline,click,_Data,[Col,Row,Text|_]} ->
     io:format("Click at col:~p row:~p text:~p~n",[Col,Row,Text]),
     loop();
        Msg ->
            io:format("Got ~p~n",[Msg]),
     loop()
```

```
end.
```

1.8.12 Editor

The editor object is a simple text editor.

{Option,Value}	*Default*	*Description*
{hscroll, Bool | top | bottom}	false	Horizontal scroll bar.
{insertpos,{row,Col}}	<unspec>	The position of the cursor.
{insertpos,'end'}	<unspec>	The position of the cursor.
{justify, left| right| center}	left	Text justification.
{scrollbg, Color}	<unspec>	Background color of scroll bar.
{scrollfg, Color}	<unspec>	Foreground color of scroll bar.
{selection, {FromIndex,ToIndex}}	<unspec>	The text range that is currently selected.
{vscroll, Bool | left | right}	false	Vertical scroll bar.
{vscrollpos, row}	<unspec>	The top most visible row in the editor.
{wrap, none|char | word}	none	How to wrap text when the line is full.

Table 8.41: Editor Options

Config-Only	*Description*
clear	Clears the editor.
{del, {FromIndex, ToIndex}}}	Deletes text.
{fg, {{FromIndex,ToIndex},Color}}	Sets the foreground color of a range of text.
{load, FileName}	Read FileName into the editor.
{insert, {Index, Text}}	Inserts new text.
{overwrite, {Index, Text}}	Writes new text at index.
{save, FileName}	Writes editor contents to file.

Table 8.42: Editor Config-Only Options

Read-Only	*Return*	*Description*
char_height	Int	The height of the editor window measured in characters.
char_width	Int	The width of the editor window measured in characters.
{fg,Index}	Int	The foreground color of the text at Index.
{get,{FromIndex, ToIndex}}	Text	The text between the indices.
size	Int	The number of rows in the editor.

Table 8.43: Editor Read-Only Options

```
Index: 'end'|insert|{row,Col}|{row,lineend}
```

Editor Demo

Figure 8.15: Simple Editor

```
-module(ex14).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $ ').

-export([start/0,init/0]).

start() -> spawn(ex14, init, []).

init() ->
    Y = [{y,0},{height, 30},{width, 90}],
    R=[{window, [{width, 400},{height, 300}, {title,"editor"},{map, true}],
       [{editor,editor,[{x,0},{y, 35},{width,300},{height,250},
   {insert,{'end',"Edit this text!"}},{vscroll,right}]},
 {button, clear, [{label, {text, "Clear"}},{x,0} | Y]},
 {checkbutton,enable,[{label,{text,"Enable"}},{select,false},{x,100}|Y]},
 {button, time, [{label, {text, "Insert Time"}},{x,200} | Y]},
 {button, quit,  [{label, {text, "Quit"}},{x,300} | Y]}]}],
    gs:create_tree(gs:start(),R),
    gs:config(editor,{enable,false}),
    loop().

loop() ->
    receive
        {gs, clear, _, _, _} ->
            io:format("clear editor~n"),
            Enable = gs:read(editor, enable),
            gs:config(editor,{enable, true}),
            gs:config(editor,clear),
            gs:config(editor,{enable, Enable});
        {gs, enable, _, _, [_Txt, _Grp, Enable|_]} ->
```

```
            io:format("Enable: ~w~n", [Enable]),
            gs:config(editor,{enable, Enable});
        {gs, time, _, _, _} ->
            TimeStr = io_lib:format("Hr:Min:Sec is now ~w:~w:~w~n",
                                    tuple_to_list(time())),
            io:format("Insert Time: ~s~n", [TimeStr]),
            Enable = gs:read(editor, enable),
            gs:config(editor,{enable, true}),
            gs:config(editor,{insert, {insert, TimeStr}}),
            gs:config(editor,{enable, Enable});
        {gs, quit, _, _, _} ->
            exit(normal);
        Other ->
            io:format("Other:~w~n",[Other])
    end,
    loop().
```

1.8.13 Scale

A scale object is used to select a value within a specified range.

{Option,Value}	*Default*	*Description*
{orient, vertical | horizontal}	horizontal	The orientation of the scale.
{pos, Int}	<unspec>	The current value of the scale objects within the range.
{range, {Min, Max}}	<unspec>	The value range.
{showvalue, Bool}	true	Turns showing of scale value on/off.
{text, String}	<unspec>	If specified, a label will be attached to the scale.

Table 8.44: Scale Object Options

Event
{gs, Scale, click, Data, [Value | _]}

Table 8.45: Scale Object Options

The following example prompts a user to specify an RGB-value for the background color of a window.

Figure 8.16: Scale Objects for Selecting RGB Values for a Window

```
-module(ex11).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/3 $ ').

-export([start/0,init/0]).
```

```
start() ->
    spawn(ex11,init,[]).

init() ->
    I= gs:start(),
    W= gs:window(I,[{title,"Color Demo"},
                    {width,300},{height,195}]),
    B=gs:button(W,[{label,{image,"die_icon"}},{x,271},{y,166},
                   {width,30}]),
    gs:config(B,[{bg,yellow},{fg,hotpink1},{data,quit}]),
    gs:scale(W,[{text,"Red"},{y,0},{range,{0,255}},
                {orient,horizontal},
                {height,65},{data,red},{pos,42}]),
    gs:scale(W,[{text,"Blue"},{y,65},{range,{0,255}},
                {orient,horizontal},
                {height,65},{data,blue},{pos,42}]),
    gs:scale(W,[{text,"Green"},{y,130},{range,{0,255}},
                {orient,horizontal},
                {height,65},{data,green},{pos,42}]),
    gs:config(W,{map,true}),
    loop(W,0,0,0).

loop(W,R,G,B) ->
    gs:config(W,{bg,{R,G,B}}),
    receive
        {gs,_,click,red,[New_R|_]} ->
            loop(W,New_R,G,B);
        {gs,_,click,green,[New_G|_]} ->
            loop(W,R,New_G,B);
        {gs,_,click,blue,[New_B|_]} ->
            loop(W,R,G,New_B);
        {gs,_,click,quit,_} ->
            true;
        {gs,W,destroy,_,_} ->
            true
    end.
```

2 Reference Manual

The Graphics System application, *GS*, is a library of routines for writing graphical user interfaces. Programs written using GS work on all Erlang platforms and do not depend upon the underlying windowing system.

gs

Erlang module

> ### Warning:
>
> GS is not recommended for use in new applications. Instead we recommend WX for applications that need a graphical user interface.
>
> GS is deprecated and will be removed in the 18.0 release.

The Graphics System, GS, is easy to learn and designed to be portable to many different platforms.

In the description below, the type `gsobj()` denotes a reference to a graphical object created with GS. Such a reference is either a GS object identifier or the name of the object (an atom), if such a name exists. The functions all return the specified values or `{error,Reason}` if an error occurs.

Please refer to the GS User's Guide for a description of the different object types and possible options.

Exports

`config(GSObj, Options) -> ok`

Types:

```
GSOBj = gsobj()
Options = [Option] | Option
 Option = {Key,Value}
```

Configures a graphical object according to the specified options.

`create(ObjType, Parent) -> ObjId`
`create(ObjType, Parent, Options) -> ObjId`
`create(ObjType, Name, Parent, Options) -> ObjId`

Types:

```
ObjType = atom()
Parent = gsobj()
Name = atom()
Options = [Option] | Option
 Option = {Key,Value}
```

Creates a new graphical object of the specified type as a child to the specified parent object. The object is configured according to the options and its identifier is returned. If no options are provided, default option values are used.

If a name is provided, this name can be used to reference the object instead of the object identifier. The name is local to the process which creates the object.

The following object types exist: `window | button | radiobutton | checkbutton | label | frame | entry | listbox | canvas | arc | image | line | oval | polygon | rectangle | text | menubar | menubutton | menu | menuitem | grid | gridline | editor | scale`

create_tree(Parent, Tree) -> ok

Types:

Parent = gsobj()

Tree = [Object]

Object = {ObjType,Options} | {ObjType,Options,Tree} | {ObjType,Name,Options,Tree}

Creates a hierarchy of graphical objects.

destroy(GSObj) -> void()

Types:

GSObj = gsobj()

Destroys a graphical object and all its children.

ObjType(Parent)

ObjType(Parent, Options)

ObjType(Name, Parent, Options)

These functions are shorthand equivalents of `create/2`, `create/3`, and `create/4`, respectively.

read(GSObj, Key) -> Value

Types:

GSObj = gsobj()

Key = atom()

Value = term()

Returns the value of an option key for the specified graphical object.

start() -> ObjId

Starts GS, unless it is already started, and returns its object identifier.

stop() -> void()

Stops GS and closes all windows. This function is not the opposite of `start/0` as it will cause *all* applications to lose their GS objects.