CS 23 Software Design and Implementation

Lecture 20

GTK+ Programming

In the this lecture we study GTK programming.

Goals

We plan to learn the following from today's lecture:

- What is a GTK?
- How to make a GTK application
- Events, Signals, and Callbacks
- Widgets

Introduction

The GIMP Toolkit (GTK+) was originally designed for a raster graphics editor called the GNU Image Manipulation (GIMP). GTK+ was adopted as the default graphical toolkit of GNOME and XFCE, two of the most popular Linux desktop environments. While it was originally used on the Linux operating system, GTK+ has been expanded to support other UNIX-like operating systems: Microsoft Windows, BeOS, Solaris, Mac OS X, and others. GTK+ is written entirely in C, and the majority of GTK+ software is also written in C. Fortunately there are a number of language bindings that allow you to use GTK+ in your preferred language like C++, Python, PHP, Ruby, Perl, C#, or Java. We can develop programs with graphical interfaces by using GTK+ on the X window system.

Architecture

As seen in Figure 1, GTK+ is built on top of a number of other libraries.

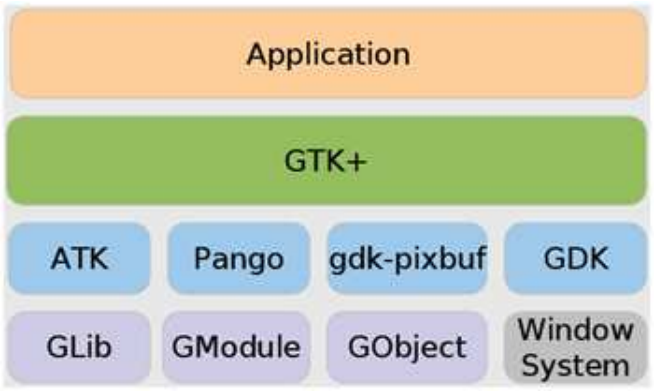

Figure 1: Architecture of GTK+

Glib - provides low-level data structures, types, thread support, the event loop, and dynamic loading.

GObject - implements an object-oriented system in C without requiring C++.

Pango - supports text rendering and layout.

ATK (Accessability ToolKit) - helps you create accessible applications and allows users to run your applications with screen readers and other accessibility tools.

GDK (GIMP Drawing Kit) - handles low-level graphics rendering on top of the Xlib.

GdkPixbuf - helps manipulate images within GTK+ programs.

Xlib - provides the low-level graphics on Linux and UNIX systems.

GModule - portable method for dynamically loading 'plug-ins'

Simple example

Now, we will write a simple GTK+ program. It doesn't do much, but it shows the basic structure which is needed to make a program with a GUI on Linux. Figure 2 is the screenshot of the program. The program will not be killed even though you press the 'X' button. You should kill this program by using *Ctrl+c* on the terminal because it does not have code to destroy a program.

The following code is for making a simple window.

Figure 2: screenshot of hello.c

```
#include <gtk/gtk.h>

int main (int argc, char *argv[])
{
    GtkWidget *window;

    /* Initialize the GTK+ and all of its supporting libraries. */
    gtk_init (&argc, &argv);

    /* Create a new window, give it a title and display it to the user. */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Hello World");
    gtk_widget_show (window);

    /* Hand control over to the main loop. */
    gtk_main ();
    return 0;
}
```

The `<gtk/gtk.h>` file includes all of the widgets, variables, functions, and structures available in GTK+ as well as header files from other libraries that GTK+ depends on. When you implement applications using GTK+, it is enough to use only `<gtk/gtk.h>`, except for some advanced applications.

The following code declares a pointer variable for a `GtkWidget` object.

```
GtkWidget *window;
```

A GTK+ program consists of widgets. Components such as windows, buttons, and scrollbars are called widgets. We will see widgets in detail later.

The following function initializes GTK+

```
/* Initialize GTK+ and all of its supporting libraries. */
gtk_init (&argc, &argv);
```

By calling `gtk_init()`, all initialization work is automatically performed for you. It begins by setting up the GTK+ environment, including obtaining the GDK display and preparing the GLib main event loop and basic signal handling. It is important to call `gtk_init()` before any other function calls to the GTK+ libraries. Otherwise, your application will not work properly and will likely crash.

The following code creates a new `GtkWindow` object and sets some properties.

```
/* Create a new window, give it a title and display it to the user. */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title (GTK_WINDOW (window), "Hello World");
gtk_widget_show (window);
```

`gtk_window_new()` creates a `GtkWindow` object that is set to the default width and height of 200 pixels. We passed `GTK_WINDOW_TOPLEVEL` to `gtk_window_new()`. This makes GTK+ create a new top-level window. Top-level windows use window manager decorations, have a border frame, and allow themselves to be placed by the window manager. Conversely, you can use `GTK_WINDOW_POPUP` to create a pop-up window. Pop-up windows are used for things that are not normally thought of as windows, such as tooltips and menus. `GTK_WINDOW_TOPLEVEL` and `GTK_WINDOW_POPUP` are the only two elements available in the `GtkWindowType` enumeration.

The following function, `gtk_window_set_title()`, requests the title bar and taskbar to display "Hello World" as the title of the window. The first parameter is for `GtkWindow` object and the second one is the string that you want to display.

Lastly, the `gtk_widget_show()` function tells GTK+ to set the specified widget as visible. The widget may not be immediately shown when you call `gtk_widget_show()`, because GTK+ queues the widget until all preprocessing is completed before it is drawn onto the screen.

This function is the main loop that takes control and starts processing events.

```
/* Hand control over to the main loop. */
gtk_main ();
```

The `gtk_main()` function will continue to run until you call `gtk_main_quit()` or the application terminates. This should be the last GTK+ function called in main(). In GTK+, signals and callback functions are triggered by user actions such as button clicks, asynchronous input-output events, programmable timeouts, and others. We will explore them in the Signal, Callbacks, and Events section later.

So far, we have written a simple GTK+ program, learned about the structure of GTK+, and understood what each part means and how it works. Now, we should compile the code we wrote in order to make it executable.

The following command compiles the code.

```
gcc hello.c -o hello ‘pgk-config -cflags -libs gtk+-2.0‘
```

This command can be used for most of the examples except for some which use specific library like `pthread`.

```
Take care to type backticks, not apostrophes - backticks are instructions to
 the shell to execute and append the output of the enclosed command.
( ‘ is correct, not ’. ‘ is located on the upper side of ’tab’ key )
```

In addition to the GCC compiler, you need to use the `pkg-config` application, which returns a list of specified libraries or paths. The first instance, `pkg-config --cflags gtk`-2.0+, returns directory names to the compiler's 'include path'. This will make sure that the GTK+ header files are available to the compiler. The second call, `pkg-config --libs gtk`-2.0+, appends options to the command line used by the linker including library directory path extensions and a list of libraries needed for linking to the executable. The libraries that are returned in a standard Linux environment are:

- GTK+ (-lgtk): Graphical widgets
- GDK (-lgdk): The standard graphics rendering library
- GdkPixbuf (-lgdk_pixbuf): Client-side image manipulation
- Pango (-lpango): Font rendering and output
- GObject (-lgobject): Object-oriented type system
- GModule (-lgmodule): Dynamically loading libraries
- GLib (-lglib): Data types and utility functions
- Xlib (-lX11): X Window System protocol library
- Xext (-lXext): X extensions library routines
- GNU math library (-lm): The GNU library from which GTK+ uses many routines

If you have PATH trouble in compilation, you need to set up your path.

```
Bash : export PKG_CONFIG_PATH=/usr/lib/pkgconfig

Csh  : setenv PKG_CONFIG_PATH /usr/lib/pkgconfig
```

Events, Signals, and Callbacks

GTK+ is a system that relies on events, signals, and callbacks. An event is a message emitted by the X Window System. When a user performs some action like clicking a mouse or typing a keyboard, it is emitted and sent to your application to be interpreted by the signal system provided by `GLib`. A signal is reaction to an event, which is emitted by a GtkObject. When the event reaches a widget, the signal occurs. You can tell GTK+ to run a function when the signal is emitted. This is called a callback function.

Note that a GTK+ signal is quite separate from a UNIX signal.

After we initialize our user interface, the control is given to the `gtk_main()` function, which sleeps until a signal is emitted. The callback function will be called when the action has occurred and the signal is emitted or when you have explicitly emitted the signal. You can also make signals stop being emitted.

The `g_signal_connect()` function connects the signal.

```
gulong g_signal_connect ( gpointer object,
            const gchar *signal_name,
            GCallback handler,
            gpointer data);
```

There are four parameters. The object is the widget that is to be monitored for the signal. Next, you specify the name of the signal you want to keep track of with the `signal_name`. The handler is the callback function that will be called when the signal is emitted, cast with `G_CALLBACK()`. The last one, data allows you to send a pointer to the callback function. The return value of `g_signal_connect()` is the handler identifier of the signal.

Prior callback connection function was `gtk_signal_connect()`. This function has been replaced with `g_signal_connect()` and should not be used in new code.

Callback funtions

Callback functions specified in `g_signal_connect()` will be called when the signal is emitted on the widget to which it was connected. The callback functions are in the following form and are named by a programmer.

```
static void callback_function ( GtkWidget *widget,
                                ... /* other possible arguments */ ... ,
                                gpointer data);
```

The first parameter widget is the object from `g_signal_connect()`. It must always be cast as the widget type for which the signal was created. There are other possible arguments that may appear in the middle as well, although this is not always the case. The data parameter correspond to the last argument of `g_signal_connect()`, which is `gpointer data`. Since the data is passed as a void pointer, you can replace the data type with what you want to cast.

Example

We will extend the simple 'hello world' application which has been implemented. The extension is to connect callback functions to the window signal, so the application can terminate itself without using *Ctrl+C*. We will review Signals, Callbacks, and Events through this example.

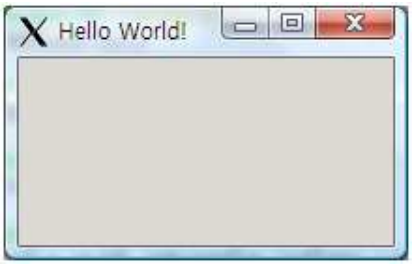

Figure 3: screenshot of hello2.c

This example shows you how to use signals, callbacks, and events.

```
#include <gtk/gtk.h>

void destroy(GtkWidget *widget, gpointer data)
{
    gtk_main_quit();
}

int main(int argc, char *argv[])
{
    GtkWidget *window;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Hello World!");
    gtk_widget_show (window);

    /* Connect the main window to the destroy */
    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(destroy), NULL);

    gtk_widget_show(window);

    gtk_main();

    return 0;
}
```

Widgets

Widgets are the basic building blocks of a GUI application. The widgets in GTK+ use the GObject hierarchy system, which allows you to derive new widgets from those that already exist. Child widgets inherit properties, functions, and signals from their parent, their grandparent, and so on

There are a huge number of widgets in GTK+. So, it's hard to describe all of them here. I will try to introduce some useful widgets for the project. You can look up other widgets on the web pages which describe the GTK+ APIs described in 'Useful links' section.

Window widget

`GtkWindow` is the basic element of all GTK+ applications. There are dozens of `GtkWindow` API calls, but here are the functions worthy of special attention.

```
/* creates a new, empty window in memory */
GtkWidget* gtk_window_new (GtkWindowType type);

/* changes the text of the tile bar by informing the window manager of the request */
void gtk_window_set_title (GtkWindow *window, const gchar *title);

/* controls the position of the initial placement onscreen */
void gtk_window_set_position (GtkWindow *window, GtkWindowPosition position);

/* sets the size of the window onscreen in GTK+ drawing units */
void gtk_window_set_default_size (GtkWindow *window, gint width, gint height);

/* forces a resize of the window once it's onscreen */
void gtk_window_resize (GtkWindow *window, gint width, gint height);

/* sets whether the user can resize a window */
void gtk_window_set_resizable (GtkWindow *window, gboolean resizable);

/* asks to maximize window, so that it becomes full-screen */
void gtk_window_maximize (GtkWindow *window);
```

Container widgets (layout)

The main purpose of a container class is to allow a parent widget to contain one or more children. We can organize our widgets with non-visible widgets called layout containers.

`GtkBox` widget

`GtkBox` is an abstract container widget that allows multiple children to be packed in a one dimensional, rectangular area. There are two types of boxes: `GtkVBox` and `GtkHBox`.

`GtkHBox` widget

This widget is a single row horizontal packing box widget.

`GtkVBox` widget

This widget is a single column vertical packing box widget.

Example

In this example, we will implement the application which contains `hbox` and `vbox` widgets.

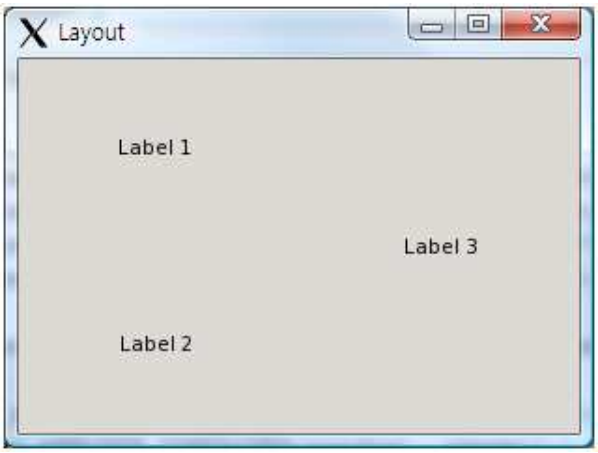

Figure 4: screenshot of layout.c

```
#include <gtk/gtk.h>

void closeApp(GtkWidget *widget, gpointer data)
{
    gtk_main_quit();
}

int main( int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *label1, *label2, *label3;
    GtkWidget *hbox;
    GtkWidget *vbox;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title(GTK_WINDOW(window), "Layout");
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);

    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(closeApp), NULL);

    label1 = gtk_label_new("Label 1");
    label2 = gtk_label_new("Label 2");
    label3 = gtk_label_new("Label 3");

    hbox = gtk_hbox_new(TRUE, 5);
    vbox = gtk_vbox_new(FALSE, 10);

    gtk_box_pack_start(GTK_BOX(vbox), label1, TRUE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), label2, TRUE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(hbox), vbox, FALSE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(hbox), label3, FALSE, FALSE, 5);

    gtk_container_add(GTK_CONTAINER(window), hbox);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

We have three labels and one `vbox` and one `hbox` in the program. The `vbox` contains the `label1` and the `lable2`. The `hbox` contains the `vbox` which consists of `label1` and `label2` and the `label3`. Finally, the `hbox` is located in the window.

These functions create a new `hbox` and `vbox`, respectively.

```
hbox = gtk_hbox_new(TRUE, 5);
vbox = gtk_vbox_new(FALSE, 10);
```

The first parameter represents homogeneity of all children. If it is set to TRUE, all children have equal space allotments. The second one is space between children.

This function adds child to box, packed with reference to the start of box.

```
gtk_box_pack_start(GTK_BOX(vbox), label1, TRUE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(vbox), label2, TRUE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), vbox, FALSE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), label3, FALSE, FALSE, 5);
```

```
void gtk_box_pack_start (GtkBox *box, GtkWidget *child,
                         gboolean expand, gboolean fill, guint padding);
```

This is a prototype of the function. The first parameter should be `box` object where children will be packed. The second one means a `child` widget to be added to `box`. These three parameters, `expand`, `fill`, and `padding`, are related to spacing of children.

This function adds widget to container.

```
gtk_container_add(GTK_CONTAINER(window), hbox);
```

This function is typically used for simple containers like `GtkWindow`, `GtkFrame`, or `GtkButton`. The first parameter is for a `container` and the second one is for a `widget`.

Basic widgets

`GtkLabel` widget

This widget is normally used to label other widgets. However, they can also be used for such things as creating large blocks of noneditable, formatted, or wrapped text.

`GtkButton` widget

This widget is a special type of container that turns its child into a clickable entity. It is only capable of holding one child. There are many derived widgets from `GtkButton` such as `GtkToggleButton`, `GtkCheckButton`, and `GtkRadioButton`.

`GtkEntry` widget

This widget is a single-line text entry widget that is commonly used to enter simple textual information. It is implemented in a general manner, so that it can be molded to fit many types of solutions. It can be used for text entry, password entry, and even number selections.

Example

Figure 5: screenshot of entry.c

This application deals with several widgets and signals, as well as callbacks we have seen above.

```
#include <gtk/gtk.h>
#include <stdio.h>
#include <string.h>

const char *password = "secret";

void closeApp(GtkWidget *window, gpointer data)
{
    printf("Destroy\n");
    gtk_main_quit();
}

void button_clicked(GtkWidget *button, gpointer data)
{
    const char *password_text = gtk_entry_get_text(GTK_ENTRY((GtkWidget *)data));

    if(strcmp(password_text, password) == 0)
        printf("Access granted!\n");
    else
        printf("Access denied!\n");
}
```

```
int main(int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *username_label, *password_label;
    GtkWidget *username_entry, *password_entry;
    GtkWidget *ok_button;
    GtkWidget *hbox1, *hbox2;
    GtkWidget *vbox;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Basic Widgets");
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 200);

    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(closeApp), NULL);

    username_label = gtk_label_new("Login: ");
    password_label = gtk_label_new("Password: ");
    username_entry = gtk_entry_new();
    password_entry = gtk_entry_new();
    gtk_entry_set_visibility(GTK_ENTRY(password_entry), FALSE);
    ok_button = gtk_button_new_with_label("OK");

    g_signal_connect(G_OBJECT(ok_button), "clicked", G_CALLBACK(button_clicked),
                     password_entry);

    hbox1 = gtk_hbox_new(TRUE, 5);
    hbox2 = gtk_hbox_new(TRUE, 5);
    vbox = gtk_vbox_new(FALSE, 10);

    gtk_box_pack_start(GTK_BOX(hbox1), username_label, TRUE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(hbox1), username_entry, TRUE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(hbox2), password_label, TRUE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(hbox2), password_entry, TRUE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), hbox1, FALSE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), hbox2, FALSE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), ok_button, FALSE, FALSE, 5);
    gtk_container_add(GTK_CONTAINER(window), vbox);

    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```

In this example, we try to use widgets and callback functions which we have learned so far.

This sub-function plays a role to check whether or not input password is correct.

```
void button_clicked(GtkWidget *button, gpointer data)
{
    const char *password_text = gtk_entry_get_text(GTK_ENTRY((GtkWidget *)data));

    if(strcmp(password_text, password) == 0)
        printf("Access granted!\n");
    else
        printf("Access denied!\n");
}
```

We developed the callback function as usual. However, note that we get `GtkEntry` widget as an argument to retrieve password information in the function.

`gtk_label_new()` function creates a new label.

```
username_label = gtk_label_new("Login: ");
```

This function gets a text as a parameter.

These functions are related to entry widget.

```
password_entry = gtk_entry_new();
gtk_entry_set_visibility(GTK_ENTRY(password_entry), FALSE);
gtk_entry_get_text(GTK_ENTRY((GtkWidget *)data));
```

`gtk_entry_new()` function creates a new entry widget. After creating the entry, we can set visibility of entry widget using `gtk_entry_set_visibility()` that has two parameters, `entry` and `visible`. Finally, we can retrieve text information from the entry widget using `gtk_entry_get_text()`.

This function creates a new button with a label.

```
ok_button = gtk_button_new_with_label("OK");
```

`GtkImage` widget

This widget is used to display an image.

Example

Figure 6: screenshot of image.c

```
#include <gtk/gtk.h>

int main( int argc, char *argv[])
{
    GtkWidget *window, *image;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);
    gtk_window_set_title(GTK_WINDOW(window), "Image");
    gtk_window_set_resizable(GTK_WINDOW(window), FALSE);

    gtk_container_set_border_width(GTK_CONTAINER(window), 2);

    image = gtk_image_new_from_file("pic/60cm.gif");
    gtk_container_add(GTK_CONTAINER(window), image);

    g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(gtk_main_quit), NULL);
    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```

This function creates a new image object from a file.

```
image = gtk_image_new_from_file("pic/60cm.gif");
```

We can load a variety of image formats such as JPG, BMP, GIF, TIF, PNG, and so on.

Example

This example uses different method to load an image. Unlike the previous example, an array of data will be used for the image source.

Figure 7: screenshot of image2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gtk/gtk.h>

#define CAMERA_WIDTH 128
#define CAMERA_HEIGHT 128

int loadImage(unsigned char *data)
{
        printf("Got image!\n");
        GdkPixbuf *pixbuf = gdk_pixbuf_new_from_data(data, GDK_COLORSPACE_RGB,
                                        FALSE, 8, CAMERA_WIDTH, CAMERA_HEIGHT,
                                        CAMERA_WIDTH  * 3, NULL, NULL);
        gtk_image_set_from_pixbuf((GtkImage*) image, pixbuf);
        gtk_widget_queue_draw(image);
        printf("Loaded\n");

        return 0;
}
```

```
unsigned char *rgbImage;
GtkWidget *image;

int main( int argc, char *argv[])
{
    GtkWidget *window;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Image2");
    gtk_window_set_resizable(GTK_WINDOW(window), FALSE);

    gtk_container_set_border_width(GTK_CONTAINER(window), 2);

    image = gtk_image_new();

    gtk_container_add(GTK_CONTAINER(window), image);

    g_signal_connect(G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);

    gtk_widget_show_all(window);

        loadImage(rgbImage);

    gtk_main();

    return 0;
}
```

This function creates a new empty image object.

```
image = gtk_image_new();
```

We used three functions to make an image file from data.

```
GdkPixbuf *pixbuf = gdk_pixbuf_new_from_data(data, GDK_COLORSPACE_RGB, FALSE, 8,
                                             CAMERA_WIDTH, CAMERA_HEIGHT,
                                             CAMERA_WIDTH  * 3, NULL, NULL);

gtk_image_set_from_pixbuf((GtkImage*) image, pixbuf);

gtk_widget_queue_draw(image);
```

`gdk_pixbuf_new_from_data()` function creates a new GdkPixbuf from in-memory data. Currently only RGB images with 8 bits per sample are supported.

`gtk_image_set_from_pixbuf()` function sets the Pixbuf data to the image object.

Lastly, `gtk_widget_queue_draw()` function notifies X Window that the entire area of a widget needs to be updated. Eventually, the image widget will be re-drawn an image.

Useful links

- The GTK+ Project

  `http://www.gtk.org`

- GTK+ Reference Manual

  `http://library.gnome.org/devel/gtk/stable`

- GDK Reference Manual

  `http://library.gnome.org/devel/gdk/unstable`

- GObject Reference Manual

  `http://library.gnome.org/devel/gobject/stable`

- GTK+ tutorial

  `http://zetcode.com/tutorials/gtktutorial`

Reference

- Useful links above
- Neil Matthew, Richard Stones, "Beginning Linux Programming 4th Edition", WROX
- Andrew Krause, Foundation of GTK+ Development, APRESS