Programming with Clutter

Murray Cumming

Programming with Clutter

by Murray Cumming

Copyright © 2007, 2008 Openismus GmbH

We very much appreciate any reports of inaccuracies or other errors in this document. Contributions are also most welcome. Post your suggestions, critiques or addenda to the team (mailto:murrayc@openismus.com).

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. You may obtain a copy of the GNU Free Documentation License from the Free Software Foundation by visiting their Web site or by writing to: Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Table of Contents

1. Introduction........1
1.1. This book........1
1.2. Clutter........1
2. Installation........3
2.1. Prebuilt Packages........3
2.2. Installing From Source........3
2.2.1. Dependencies........3
3. Header Files And Linking........4
4. The Stage........5
4.1. Stage Basics........5
4.1.1. Example........5
4.2. Stage Widget........7
4.2.1. GTK+ integration........7
4.2.2. Example........8
4.3. Stage Widget Scrolling........10
4.3.1. Example........11
5. Actors........14
5.1. Actor Basics........14
5.1.1. Example........14
5.2. Transformations........16
5.2.1. Scaling........16
5.2.2. Rotation........16
5.2.3. Clipping........17
5.2.4. Movement........17
5.2.5. Example........17
5.3. Containers........19
5.3.1. Example........20
5.4. Events........21
5.4.1. Example........22
6. Timelines........27
6.1. Using Timelines........27
6.2. Markers........27
6.3. Example........28
6.4. Grouping TimeLines in a Score........30
6.5. Example........30
7. Animations........34
7.1. Using Animations........34
7.2. Using Alpha Functions........34
7.3. Example........35
8. Behaviours........38
8.1. Using Behaviours........38
8.2. Example........40

9. Text editing **43**
9.1. ClutterText 43
9.1.1. Size Management 43
9.2. Example 43
10. Full Example **47**
A. Implementing Actors **58**
A.1. Implementing Simple Actors 58
A.2. Example 59
A.3. Implementing Container Actors 66
A.3.1. ClutterActor virtual functions to implement 67
A.3.2. ClutterContainer virtual functions to implement 67
A.4. Example 68
B. Implementing Scrolling in a Window-like Actor **79**
B.1. The Technique 79
B.2. Example 79
11. Contributing **90**

List of Figures

4-1. Stage ... 5
4-2. Stage Widget ... 8
4-3. Stage Widget Scrolling ... 11
5-1. Actor ... 15
5-2. Actor ... 17
5-3. Group ... 20
5-4. Actor Events ... 23
6-1. Timeline ... 28
6-2. Score ... 31
7-1. Animation ... 35
8-1. Effects of alpha functions on a path. ... 38
8-2. Behaviour ... 40
9-1. ClutterText ... 44
10-1. Full Example ... 47
A-1. Behaviour ... 59
A-2. Behaviour ... 68
B-1. Scrolling Container ... 79

Chapter 1. Introduction

1.1. This book

This book assumes a good understanding of C, and how to create C programs.

This book attempts to explain key Clutter concepts and introduce some of the more commonly used user interface elements ("actors"). For full API information you should follow the links into the reference documentation. This document covers the API in Clutter version 1.0.

Each chapter contains very simple examples. These are meant to show the use of the API rather than show an impressive visual result. However, the full example should give some idea of what can be achieved with Clutter

The Clutter platform uses techniques found in the GTK+ (http://www.gtk.org) platform, so you will sometimes wish to refer to the GTK+ documentation.

We would very much like to hear of any problems you have learning Clutter with this document, and would appreciate input regarding improvements. Please see the Contributing section for further information.

1.2. Clutter

Clutter is a C programming API that allows you to create simple but visually appealing and involving user interfaces. It offers a variety of objects (actors) which can be placed on a canvas (stage) and manipulated by the application or the user. It is therefore a "retained mode" graphics API. Unlike traditional 2D canvas APIs, Clutter allows these actors to move partly in the Z dimension.

This concept simplifies the creation of 3D interfaces compared to direct use of OpenGL or other 3D drawing APIs. For instance, it restricts the user interaction to the 2D plane facing the user, which is appropriate for today's devices allowing interaction only with a 2D plane such as a touchscreen. In addition, your application does not need to provide visual context to show the user which objects are, for instance, small rather than far away.

In addition, Clutter provides timeline and behavior abstractions which simplify animation by allowing you to associate actor properties (such as position, rotation, or opacity) with callback functions, including pre-defined functions of time such as sine waves.

Clutter uses the popular OpenGL 3D graphics API on regular desktop PCs, allowing it access to hardware acceleration. On handheld devices it can use OpenGL ES, a subset of the OpenGL API aimed at embedded devices. So, where necessary, you may also use OpenGL or OpenGL ES directly.

In the next few chapters you will learn how to place actors on the stage, how to set their properties, how to change their properties (including their position) over time by using timelines and behaviours, and how to do all this in response to user interaction.

Chapter 2. Installation

2.1. Prebuilt Packages

Clutter packages are probably available from your Linux distribution. For instance, on Ubuntu Linux or Debian you can install the `libclutter-1.0-dev` package.

2.2. Installing From Source

After you've installed all of the dependencies, download the Clutter source code, unpack it, and change to the newly created directory. Clutter can be built and installed with the following sequence of commands:

```
# ./configure
# make
# make install
```

The `configure` script will check to make sure all of the required dependencies are already installed. If you are missing any dependencies it will exit and display an error.

By default, Clutter will be installed under the `/usr/local` directory.

If you want to help develop Clutter or experiment with new features, you can also install Clutter from SVN. Details are available at the Clutter web site (http://www.clutter-project.org/).

2.2.1. Dependencies

Before attempting to install Clutter, you should first install these other packages:

- GTK+
- libgl (Mesa)

These dependencies have their own dependencies, including the following applications and libraries:

- pkg-config
- glib
- ATK
- Pango

Chapter 3. Header Files And Linking

To use the Clutter APIs, you must include the headers for the libraries, and link to their shared libraries. The necessary compiler and linker commands can be obtained from the pkg-config utiltity like so:

```
pkg-config clutter-1.0 --cflags
pkg-config clutter-1.0 --libs
```

However, if you are using the "autotools" (**automake**, **autoconf**, etc) build system, you will find it more convenient to use the `PKG_CHECK_MODULES` macro in your configure.ac file. For instance:

```
PKG_CHECK_MODULES(EXAMPLE, clutter-1.0)
AC_SUBST(EXAMPLE_CFLAGS)
AC_SUBST(EXAMPLE_LIBS)
```

You should then use the generated _CFLAGS and _LIBS definitions in your Makefile.am files. Note that you may mention other libraries in the same `PKG_CHECK_MODULES` call, separated by spaces. For instance, some examples in this tutorial require extra Clutter libraries, such as `clutter-gtk-1.0`, `clutter-cairo-1.0` or `clutter-gst-1.0`.

Chapter 4. The Stage

4.1. Stage Basics

Each Clutter application contains at least one `ClutterStage`. This stage contains Actors such as rectangles, images, or text. We will talk more about the actors in the next chapter, but for now let's see how a stage can be created and how we can respond to user interaction with the stage itself.

First make sure that you have called `clutter_init()` to initialize Clutter. You may then get the application's stage with `clutter_stage_get_default()`. This function always returns the same instance, with its own window. You could instead use a `GtkClutterEmbed` widget inside a more complicated GTK+ window - see the Stage Widget section.

`ClutterStage` is derived from the `ClutterActor` object so many of that object's functions are useful for the stage. For instance, call `clutter_actor_show()` to make the stage visible.

`ClutterStage` also implements the `ClutterContainer` interface, allowing it to contain child actors via calls to `clutter_container_add()`.

Call `clutter_main()` to start a main loop so that the stage can animate its contents and respond to user interaction.

Reference (http://clutter-project.org/docs/clutter/1.0/ClutterStage.html)

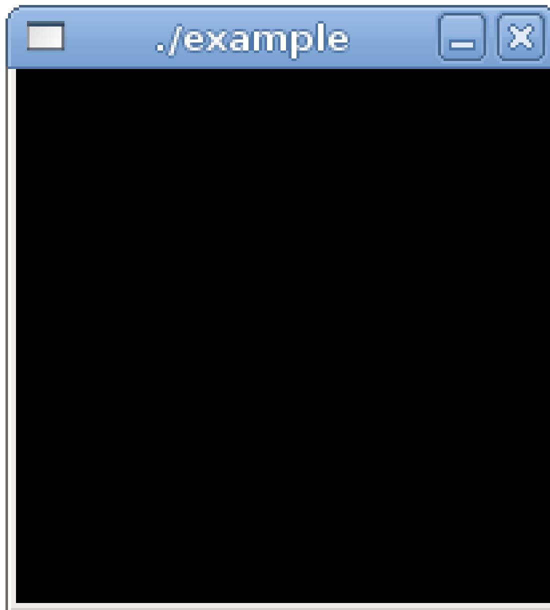
4.1.1. Example

The following example shows a `ClutterStage` and handles clicks on the stage. There are no actors yet so all you will see is a black rectangle.

You can create an executable from this code like so, being careful to use `backticks` around the call to **pkg-config**. See also the Header Files And Linking section.

```
gcc -Wall -g example.c -o example `pkg-config clutter-1.0 --cflags  --libs`
```

Figure 4-1. Stage



Source Code (../../../examples/stage)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

static gboolean
on_stage_button_press (ClutterStage *stage, ClutterEvent *event, gpointer data)
{
  float x = 0;
  float y = 0;
  clutter_event_get_coords (event, &x, &y);

  g_print ("Stage clicked at (%f, %f)\n", x, y);

  return TRUE; /* Stop further handling of this event. */
}

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff }; /* Black */

  clutter_init (&argc, &argv);
```

```
  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Connect a signal handler to handle mouse clicks and key presses on the stage: */
  g_signal_connect (stage, "button-press-event",
    G_CALLBACK (on_stage_button_press), NULL);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

4.2. Stage Widget

The `GtkClutterEmbed` widget allows you to place a `ClutterStage` inside an existing GTK+ window. For instance, the window might contain other GTK+ widgets allowing the user to affect the actors in stage. Use `gtk_clutter_embed_new()` to instantiate the widget and then add it to a container just like any other GTK+ widget. Call `gtk_clutter_embed_get_stage` to get the `ClutterStage` from the `GtkClutterEmbed` widget so you can then use the main Clutter API.

When using the `GtkClutterEmbed` widget you should use `gtk_clutter_init` instead of `clutter_init()` and `gtk_init()` to initialize Clutter and GTK+. And you should use the regular `gtk_main()` function to start the mainloop rather than `clutter_main()`.

For simplicity, all other examples in this document will instead use `clutter_stage_get_default()`, but all the techniques can also be used with a stage inside the `GtkClutterEmbed` widget.

Reference (http://www.clutter-project.org/docs/clutter-gtk/1.0/GtkClutterEmbed.html)

4.2.1. GTK+ integration

Clutter contains some useful utility functions to help you integrate your use of GTK+ and the parts you draw with Clutter.

To embed stock or other icons into clutter you can use the `gtk_clutter_texture_new_from_*()` functions. If you need to draw in the correct theme colors then the `gtk_clutter_get_*_color()` functions can receive a theme color for a `GtkWidget` in the current state and convert it to a `ClutterColor` for use with Clutter.

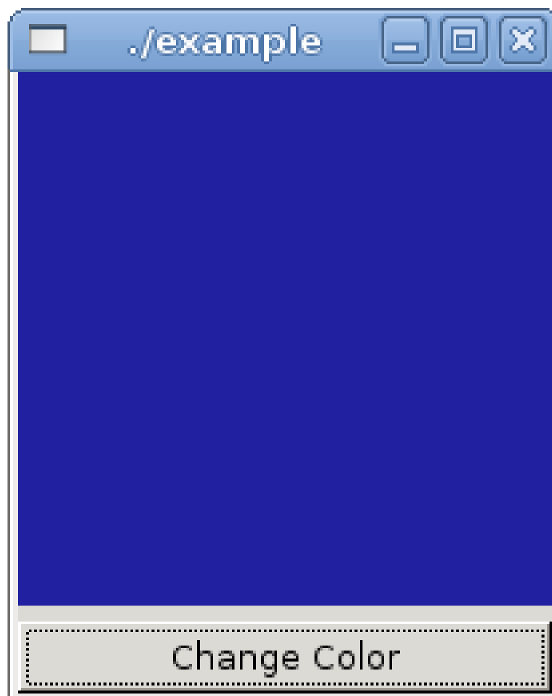
4.2.2. Example

The following example shows a `GtkClutterEmbed` GTK+ widget and changes the stage color when a button is clicked.

Note that this example requires the `clutter-gtk-1.0` library as well as the standard `clutter-1.0` library. You can create an executable from this code like so, being careful to use `backticks` around the call to **pkg-config**. See also the Header Files And Linking section.

```
gcc -Wall -g example.c -o example `pkg-config clutter-1.0 clutter-gtk-1.0 --cflags  --libs`
```

Figure 4-2. Stage Widget



Source Code (../../../examples/gtk_embed)

File: `main.c`

```
#include <gtk/gtk.h>
#include <clutter/clutter.h>
#include <clutter-gtk/clutter-gtk.h>
#include <stdlib.h>

ClutterActor *stage = NULL;

static gboolean
on_button_clicked (GtkButton *button, gpointer user_data)
{
  static gboolean already_changed = FALSE;
  if(already_changed)
  {
    ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff }; /* Black */
    clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);
  }
  else
  {
    ClutterColor stage_color = { 0x20, 0x20, 0xA0, 0xff };
    clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);
  }

  already_changed = !already_changed;

  return TRUE; /* Stop further handling of this event. */
}

static gboolean
on_stage_button_press (ClutterStage *stage, ClutterEvent *event, gpointer user_data)
{
  gint x = 0;
  gint y = 0;
  clutter_event_get_coords (event, &x, &y);

  g_print ("Stage clicked at (%d, %d)\n", x, y);

  return TRUE; /* Stop further handling of this event. */
}

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff }; /* Black */

  gtk_clutter_init (&argc, &argv);

  /* Create the window and some child widgets: */
  GtkWidget *window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  GtkWidget *vbox = gtk_vbox_new (FALSE, 6);
```

```
  gtk_container_add (GTK_CONTAINER (window), vbox);
  gtk_widget_show (vbox);
  GtkWidget *button = gtk_button_new_with_label ("Change Color");
  gtk_box_pack_end (GTK_BOX (vbox), button, FALSE, FALSE, 0);
  gtk_widget_show (button);
  g_signal_connect (button, "clicked",
    G_CALLBACK (on_button_clicked), NULL);

  /* Stop the application when the window is closed: */
  g_signal_connect (window, "hide",
                    G_CALLBACK (gtk_main_quit), NULL);

  /* Create the clutter widget: */
  GtkWidget *clutter_widget = gtk_clutter_embed_new ();
  gtk_box_pack_start (GTK_BOX (vbox), clutter_widget, TRUE, TRUE, 0);
  gtk_widget_show (clutter_widget);

  /* Set the size of the widget,
   * because we should not set the size of its stage when using GtkClutterEmbed.
   */
  gtk_widget_set_size_request (clutter_widget, 200, 200);

  /* Get the stage and set its size and color: */
  stage = gtk_clutter_embed_get_stage (GTK_CLUTTER_EMBED (clutter_widget));
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Connect a signal handler to handle mouse clicks and key presses on the stage: */
  g_signal_connect (stage, "button-press-event",
    G_CALLBACK (on_stage_button_press), NULL);

  /* Show the window: */
  gtk_widget_show (GTK_WIDGET (window));

  /* Start the main loop, so we can respond to events: */
  gtk_main ();

  return EXIT_SUCCESS;

}
```

4.3. Stage Widget Scrolling

When integrating a `ClutterStage`s into GTK+ with `GtkClutterEmbed`, you may need to scroll due to limited screen space. Normally you would add the widget inside a `GtkScrolledWindow` but this is

not possible with `GtkClutterEmbed`, as Clutter accesses the graphics hardware directly, bypassing the normal GTK+ drawing system.

Instead you should use a `GtkClutterViewport`. Unlike `GtkScrolledWindow`, this does not draw any scrollbars itself. Instead it uses a `GtkAdjustment` which you should also use with your own `GtkScrollbar` widgets, or any other `GtkRange` widget, su such as `GtkScale`.

Reference (http://www.clutter-project.org/docs/clutter-gtk/1.0/GtkClutterViewport.html)

4.3.1. Example

This example is a simple image viewer that allows scrolling of the image. Note the layout of the `GtkTable`, with the two scrollbars.

Figure 4-3. Stage Widget Scrolling

Source Code (../../../examples/gtk_scrolling)

File: `main.c`

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

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x61, 0x64, 0x8c, 0xff };

  /* Call gtk_clutter_init() to init both clutter and gtk+ */
  if (gtk_clutter_init (&argc, &argv) != CLUTTER_INIT_SUCCESS)
    g_error ("Unable to initialize GtkClutter");

  if (argc != 2)
    g_error ("Usage: example <image file>");

  /* Create a toplevel window: */
  GtkWidget *window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_default_size (GTK_WINDOW (window), 640, 480);
  g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);

  /* Create a table to hold the scrollbars and the ClutterEmbed widget: */
  GtkWidget *table = gtk_table_new (2, 2, FALSE);
  gtk_container_add (GTK_CONTAINER (window), table);
  gtk_widget_show (table);

  /* Create ClutterEmbed widget for the stage: */
  GtkWidget *embed = gtk_clutter_embed_new ();
  gtk_table_attach (GTK_TABLE (table), embed,
    0, 1,
    0, 1,
    GTK_EXPAND | GTK_FILL,
    GTK_EXPAND | GTK_FILL,
    0, 0);
  gtk_widget_show (embed);

  /* Init the stage: */
  ClutterActor *stage = gtk_clutter_embed_get_stage (GTK_CLUTTER_EMBED (embed));
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);
  clutter_actor_set_size (stage, 640, 480);

  /* Create a viewport actor to be able to scroll actor. By passing NULL it
   * will create new GtkAdjustments. */
  ClutterActor *viewport = gtk_clutter_viewport_new (NULL, NULL, NULL);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), viewport);

  /* Load image from first command line argument and add it to viewport: */
  ClutterActor *texture = clutter_texture_new_from_file (argv[1], NULL);
  clutter_container_add_actor (CLUTTER_CONTAINER (viewport), texture);
  clutter_actor_set_position (texture, 0, 0);
  clutter_actor_set_position (texture, 0, 0);
  clutter_actor_set_position (viewport, 0, 0);
  clutter_actor_set_size (viewport, 640, 480);

  /* Create scrollbars and connect them to viewport: */
  GtkAdjustment *h_adjustment = NULL;
```

```
  GtkAdjustment *v_adjustment = NULL;
  gtk_clutter_scrollable_get_adjustments (GTK_CLUTTER_SCROLLABLE (viewport),
    &h_adjustment, &v_adjustment);
  GtkWidget *scrollbar = gtk_vscrollbar_new (v_adjustment);
  gtk_table_attach (GTK_TABLE (table), scrollbar,
    1, 2,
    0, 1,
    0, GTK_EXPAND | GTK_FILL,
    0, 0);
  gtk_widget_show (scrollbar);

  scrollbar = gtk_hscrollbar_new (h_adjustment);
  gtk_table_attach (GTK_TABLE (table), scrollbar,
    0, 1,
    1, 2,
    GTK_EXPAND | GTK_FILL, 0,
    0, 0);

  gtk_widget_show (scrollbar);
  gtk_widget_show (window);

  gtk_main();

return EXIT_SUCCESS;
}
```

Chapter 5. Actors

5.1. Actor Basics

As mentioned in the introduction, Clutter is a canvas API for 2D surfaces in 3D space. Standard Clutter actors have 2D shapes and can be positioned and rotated in all three dimensions, but they have no depth. Theoretically, therefore, most actors would be invisible if they were exactly rotated so that only their edge faced the screen. When complex 3D objects are needed, you may use the full OpenGL ES API, as mentioned in the Implementing Actors appendix, but let's look at the standard actors for now:

- ClutterStage (http://clutter-project.org/docs/clutter/1.0/ClutterStage.html): The stage itself, mentioned already
- ClutterRectangle (http://clutter-project.org/docs/clutter/1.0/ClutterRectangle.html): A rectangle.
- ClutterText (http://clutter-project.org/docs/clutter/1.0/ClutterText.html): Displays and edits text.
- ClutterTexture (http://clutter-project.org/docs/clutter/1.0/ClutterTexture.html): An image.

Each actor should be added to the stage with `clutter_container_add()` and its positions should then be specified. All actors derive from `ClutterActor` so you can call `clutter_actor_set_position()` to set the x and y coordinates, and the z coordinate can be set with `clutter_actor_set_depth()`, with larger values placing the actor further away from the observer. `clutter_actor_set_size()` sets the width and height.

The actor's position is relative to the top-left (0, 0) of the parent container (such as the stage), but this origin can be changed by calling `clutter_actor_set_anchor_point()`.

By default, actors are hidden, so remember to call `clutter_actor_show`(). You may later call `clutter_actor_hide()` to temporarily hide the object again.

Like GTK+ widgets, Clutter actors have a "floating reference" when they are first instantiated with a function such as `clutter_rectangle_new()`. This reference is then taken when the actor is added to a container, such as the stage. This means that you do not need to unreference the actor after creating it.

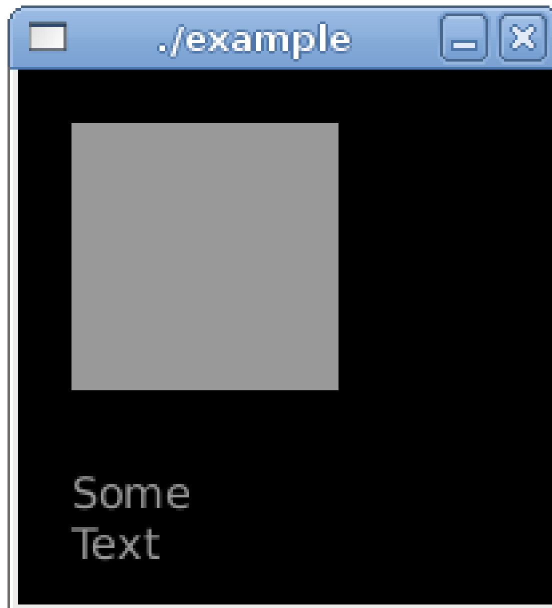
Actors may also be transformed by scaling or rotation, and may be made partly transparent.

Reference (http://clutter-project.org/docs/clutter/1.0/ClutterActor.html)

5.1.1. Example

The following example demonstrates two unmoving actors in a stage:

Figure 5-1. Actor



Source Code (../../../examples/actor)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>


int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);
```

```
  /* Add a rectangle to the stage: */
  ClutterActor *rect = clutter_rectangle_new_with_color (&actor_color);
  clutter_actor_set_size (rect, 100, 100);
  clutter_actor_set_position (rect, 20, 20);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
  clutter_actor_show (rect);

  /* Add a label to the stage: */
  ClutterActor *label = clutter_text_new_full ("Sans 12", "Some Text", &actor_color);
  clutter_actor_set_size (label, 500, 500);
  clutter_actor_set_position (label, 20, 150);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), label);
  clutter_actor_show (label);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

5.2. Transformations

Actors can be scaled, rotated, and moved.

5.2.1. Scaling

Call `clutter_actor_set_scale()` to increase or decrease the apparent size of the actor. Note that this will not change the result of `clutter_actor_get_width()` and `clutter_actor_get_height()` because it only changes the size of the actor as seen by the user. Calling `clutter_actor_set_scale()` again will replace the first scale rather than multiplying it.

5.2.2. Rotation

Call `clutter_actor_set_rotation()` to rotate the actor around an axis, specifying either `CLUTTER_X_AXIS`, `CLUTTER_Y_AXIS` or `CLUTTER_Z_AXIS` and the desired angle. Only two of the x, y, and z coordinates are used, depending on the specified axis. For instance, when using `CLUTTER_X_AXIS`, the y and z parameters specify the center of rotation on the plane of the x axis.

Like the `clutter_actor_set_scale()`, this does not affect the position, width, or height of the actor as returned by functions such as `clutter_actor_get_x()`.

5.2.3. Clipping

`Actors` may be "clipped" so that only one rectangular part of the actor is visible, by calling `clutter_actor_set_clip()`, providing a position relative to the actor, along with the size. For instance, you might implement scrolling by creating a large container actor and setting a clip rectangle so that only a small part of the whole is visible at any one time. Scrolling up could then be implemented by moving the actor down while moving the clip up. Clipping can be reverted by calling `clutter_actor_remove_clip()`.

The area outside of the clip does not consume video memory and generally does not require much processing.

5.2.4. Movement

Clutter does not have a translation function that behaves similarly to `clutter_actor_set_scale()` and `clutter_actor_set_rotation()`, but you can move the actor by calling `clutter_actor_move_by()` or `clutter_actor_set_depth`.

Unlike the scaling and rotation functions, `clutter_actor_move_by()` does change the result of functions such as `clutter_actor_get_x()`.

5.2.5. Example

The following example demonstrates two unmoving actors in a stage, using rotation, scaling and movement:

Figure 5-2. Actor

Source Code (../../../examples/actor_transformations)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>


int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add a rectangle to the stage: */
  ClutterActor *rect = clutter_rectangle_new_with_color (&actor_color);
  clutter_actor_set_size (rect, 100, 100);
  clutter_actor_set_position (rect, 20, 20);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
```

```
  clutter_actor_show (rect);

  /* Rotate it 20 degrees away from us around the x axis
   * (around its top edge)
   */
  clutter_actor_set_rotation (rect, CLUTTER_X_AXIS, -20, 0, 0, 0);


  /* Add a label to the stage: */
  ClutterActor *label = clutter_text_new_full ("Sans 12", "Some Text", &actor_color);
  clutter_actor_set_size (label, 500, 500);
  clutter_actor_set_position (label, 20, 150);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), label);
  clutter_actor_show (label);

  /* Scale it 300% along the x axis:
   */
  clutter_actor_set_scale (label, 3.00, 1.0);

  /* Move it up and to the right: */
  clutter_actor_move_by (label, 10, -10);

  /* Move it along the z axis, further from the viewer: */
  clutter_actor_set_depth (label, -20);


  /* Show the stage: */
  clutter_actor_show (stage);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

5.3. Containers

Some clutter actors implement the `ClutterContainer` interface. These actors can contain child actors and may position them in relation to each other, for instance in a list or a table formation. In addition, transformations or property changes may be applied to all children. Child actors can be added to a container with the `clutter_container_add()` function.

The main `ClutterStage` is itself a container, allowing it to contain all the child actors. The only other container in Core Clutter is `ClutterGroup`, which can contain child actors, with positions relative to the parent `ClutterGroup`. Scaling, rotation and clipping of the group applies to the child actors, which can simplify your code.

Additional Clutter containers can be found in the Tidy toolkit library. See also the Implementing Containers section.

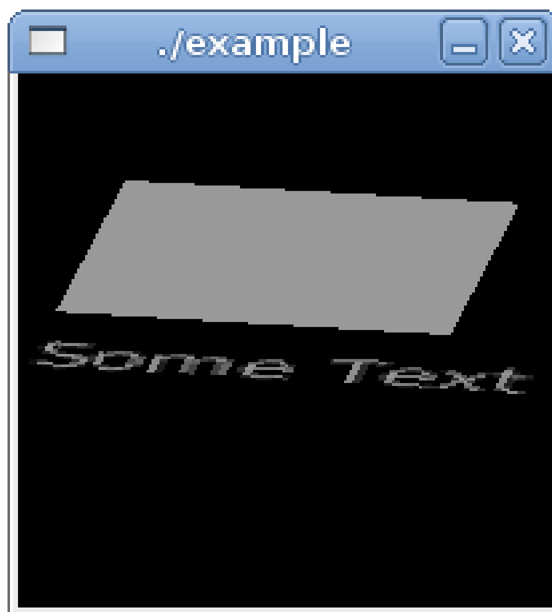
ClutterContainer Reference (http://clutter-project.org/docs/clutter/1.0/ClutterContainer.html)

ClutterGroup Reference (http://clutter-project.org/docs/clutter/1.0/ClutterGroup.html)

5.3.1. Example

The following example shows the use of the `ClutterGroup` container, with two child actors being rotated together.

Figure 5-3. Group



Source Code (../../../examples/actor_group)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>


int main(int argc, char *argv[])
```

```
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add a group to the stage: */
  ClutterActor *group = clutter_group_new ();
  clutter_actor_set_position (group, 40, 40);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), group);
  clutter_actor_show (group);

  /* Add a rectangle to the group: */
  ClutterActor *rect = clutter_rectangle_new_with_color (&actor_color);
  clutter_actor_set_size (rect, 50, 50);
  clutter_actor_set_position (rect, 0, 0);
  clutter_container_add_actor (CLUTTER_CONTAINER (group), rect);
  clutter_actor_show (rect);

  /* Add a label to the group: */
  ClutterActor *label = clutter_text_new_full ("Sans 9", "Some Text", &actor_color);
  clutter_actor_set_position (label, 0, 60);
  clutter_container_add_actor (CLUTTER_CONTAINER (group), label);
  clutter_actor_show (label);

  /* Scale the group 120% along the x axis:
   */
  clutter_actor_set_scale (group, 3.00, 1.0);

  /* Rotate it around the z axis: */
  clutter_actor_set_rotation (group, CLUTTER_Z_AXIS, 10, 0, 0, 0);


  /* Show the stage: */
  clutter_actor_show (stage);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

5.4. Events

The base ClutterActor has several signals that are emitted when the user interacts with the actor:

- button-press-event: Emitted when the user presses the mouse over the actor.
- button-release-event: Emitted when the user releases the mouse over the actor.
- motion-event: Emitted when the user moves the mouse over the actor.
- enter-event: Emitted when the user moves the mouse in to the actor's area.
- leave-event: Emitted when the user moves the mouse out of the actor's area.

For instance, you can detect button clicks on an actor like so:

```
g_signal_connect (rect, "button-press-event", G_CALLBACK (on_rect_button_press), NULL);
```

Alternatively, you might just handle signals from the parent ClutterStage and use clutter_stage_get_actor_at_pos to discover which actor should be affected.

However, as a performance optimization, Clutter does not emit all event signals by default. For instance, to receive event signals for an actor instead of just the stage, you must call clutter_actor_set_reactive(). If you don't need the motion event signals (motion-event, enter-event and leave-event), you may call the global clutter_set_motion_events_enabled() function with FALSE to further optimize performance.

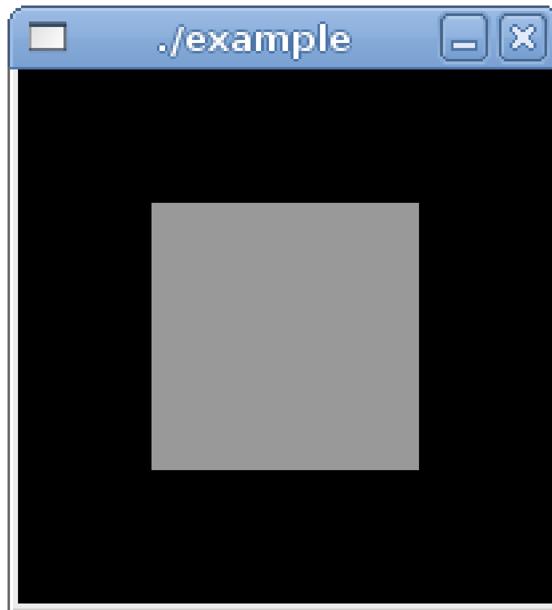
Your event signal handler should return true when it has fully handled the event, or false if you want the event to be sent also to the next actor in the event chain. Clutter first allows the stage to handle each event via the captured-event signal. But if the stage does not handle the event then it will be passed down to the child actor, first passing through the actor's parent containers, giving each actor in the hierarchy a chance to handle the event via a captured-event signal handler. If the event has still not been handled fully by any actor then the event will then be emitted via a specific signal (such as button-press-event or key-press-event. These specific signals are emitted first from the child actor, then by its parent, passing all they way back up to the stage if no signal handler returns true to indicate that it has handled the event fully.

Actors usually only receive keyboard events when the actor has key focus, but you can give an actor exclusive access to any events by grabbing either the pointer or the keyboard, using clutter_grab_pointer or clutter_grab_keyboard.

5.4.1. Example

The following example demonstrates handing of clicks on an actor:

Figure 5-4. Actor Events



Source Code (../../../examples/actor_events)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

static gboolean
on_stage_button_press (ClutterStage *stage, ClutterEvent *event, gpointer data)
{
  gfloat x = 0;
  gfloat y = 0;
  clutter_event_get_coords (event, &x, &y);

  g_print ("Clicked stage at (%f, %f)\n", x, y);

  /* Discover whether there is an actor at that position.
   * Note that you can also connect directly to the actor's signals instead.
   */
  ClutterActor *rect = clutter_stage_get_actor_at_pos (stage, CLUTTER_PICK_ALL, x, y);
```

```
  if (!rect)
    return FALSE;

  if (CLUTTER_IS_RECTANGLE (rect))
    g_print ("  A rectangle is at that position.\n");

  return TRUE; /* Stop further handling of this event. */
}

static gboolean
on_rect_button_press (ClutterRectangle *rect, ClutterEvent *event, gpointer data)
{
  gfloat x = 0;
  gfloat y = 0;
  clutter_event_get_coords (event, &x, &y);

  g_print ("Clicked rectangle at (%f, %f)\n", x, y);

  /* clutter_main_quit(); */

  return TRUE; /* Stop further handling of this event. */
}

static gboolean
on_rect_button_release (ClutterRectangle *rect, ClutterEvent *event, gpointer data)
{
  gfloat x = 0;
  gfloat y = 0;
  clutter_event_get_coords (event, &x, &y);

  g_print ("Click-release on rectangle at (%f, %f)\n", x, y);

  return TRUE; /* Stop further handling of this event. */
}

static gboolean
on_rect_motion (ClutterRectangle *rect, ClutterEvent *event, gpointer data)
{
  g_print ("Motion in the rectangle.\n");

  return TRUE; /* Stop further handling of this event. */
}

static gboolean
on_rect_enter (ClutterRectangle *rect, ClutterEvent *event, gpointer data)
{
  g_print ("Entered rectangle.\n");

  return TRUE; /* Stop further handling of this event. */
}

static gboolean
on_rect_leave (ClutterRectangle *rect, ClutterEvent *event, gpointer data)
```

```
{
  g_print ("Left rectangle.\n");

  return TRUE; /* Stop further handling of this event. */
}



int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor label_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Connect signal handlers to handle mouse clicks on the stage: */
  g_signal_connect (stage, "button-press-event",
    G_CALLBACK (on_stage_button_press), NULL);

  /* Add a Rectangle to the stage: */
  ClutterActor *rect = clutter_rectangle_new_with_color (&label_color);
  clutter_actor_set_size (rect, 100, 100);
  clutter_actor_set_position (rect, 50, 50);
  clutter_actor_show (rect);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);

  /* Allow the actor to emit events.
   * By default only the stage does this.
   */
  clutter_actor_set_reactive (rect, TRUE);

  /* Connect signal handlers for events: */
  g_signal_connect (rect, "button-press-event",
    G_CALLBACK (on_rect_button_press), NULL);
  g_signal_connect (rect, "button-release-event",
    G_CALLBACK (on_rect_button_release), NULL);
  g_signal_connect (rect, "motion-event",
    G_CALLBACK (on_rect_motion), NULL);
  g_signal_connect (rect, "enter-event",
    G_CALLBACK (on_rect_enter), NULL);
  g_signal_connect (rect, "leave-event",
    G_CALLBACK (on_rect_leave), NULL);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();
```

```
  return EXIT_SUCCESS;
}
```

Chapter 6. Timelines

6.1. Using Timelines

A `ClutterTimeline` can be used to change the position or appearance of an actor over time. These can be used directly as described in this chapter, or together with an animation or behaviour, as you will see in the following chapters.

The timeline object emits its `new-frame` signal for each frame that should be drawn, for as many frames per second as appropriate. In your signal handler you can set the actor's properties. For instance, the actor might be moved and rotated over time, and its color might change while this is happening. You could even change the properties of several actors to animate the entire stage.

The `clutter_timeline_new()` constructor function takes a duration in milliseconds. The actual number of frames per second requested by the timeline will depend on the behaviour of your entire program, the performance of your hardware, and your monitor's refresh rate. It may even vary over time as conditions change. At best, the `new-frame` signal will be emitted at your monitor's refresh rate. At worst it will be called once at the start and once at the end of your timeline's duration.

Clutter will not attempt to redraw the scene if the new frame has no change compared to the previous frame, so you don't need to do your own optimization to prevent unnecessary redraws.

You may also use `clutter_timeline_set_loop()` to cause the timeline to repeat for ever, or until you call `clutter_timeline_stop()`. The timeline does not start until you call `clutter_timeline_start()`.

Remember to unref the timeline when you are finished with it. Unlike actors, this does not have a "floating reference". You may either do this after your mainloop has finished, or when the timeline has finished, by handling the timeline's `completed` signal.

Reference (http://clutter-project.org/docs/clutter/1.0/ClutterTimeline.html)

6.2. Markers

You may want an action to happen at a specific moment in the timeline. Instead of polling with `clutter_timeline_get_progress()` you should instead use timeline markers. Timeline markers can be added with the `clutter_timeline_add_marker_at_time()` function. Handle the `marker-reached` signal to start the appropriate action at that moment, after checking the provided

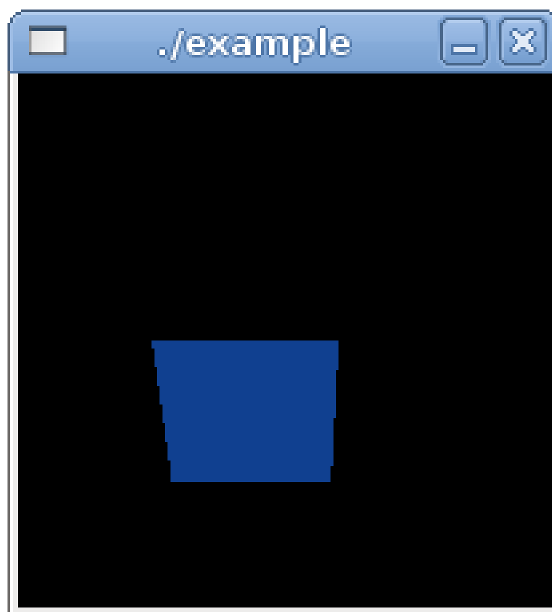
marker name. To handle only the signal only for a particular marker, you may connect to the `marker-reached::my_marker` signal, where `my-marker` is the name of your marker.

You can also use markers to navigate through the timeline using the `clutter_timeline_advance_to_marker()` function.

6.3. Example

The following example demonstrates the use of a timeline to rotate a rectangle around the x axis while changing its color:

Figure 6-1. Timeline



Source Code (../../../examples/timeline)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

ClutterActor *rect = NULL;

gint rotation_angle = 0;
```

```
gint color_change_count = 0;

void
on_timeline_new_frame (ClutterTimeline *timeline,
  gint frame_num, gpointer data)
{
  rotation_angle += 1;
  if(rotation_angle >= 360)
    rotation_angle = 0;

  /* Rotate the rectangle clockwise around the z axis, around it's top-left corner: */
  clutter_actor_set_rotation (rect, CLUTTER_X_AXIS,
    rotation_angle, 0, 0, 0);

  /* Change the color
   * (This is a silly example, making the rectangle flash):
   */
  ++color_change_count;
  if(color_change_count > 100)
    color_change_count = 0;

  if(color_change_count == 0)
  {
    ClutterColor rect_color = { 0xff, 0xff, 0xff, 0x99 };
    clutter_rectangle_set_color (CLUTTER_RECTANGLE (rect), &rect_color);
  }
  else if (color_change_count == 50)
  {
    ClutterColor rect_color = { 0x10, 0x40, 0x90, 0xff };
    clutter_rectangle_set_color (CLUTTER_RECTANGLE (rect), &rect_color);
  }
}

void
on_timeline_marker_reached (ClutterTimeline* timeline,
                            gchar*           marker_name,
                            gint             frame_num,
                            gpointer         user_data)
{
  printf ("Reached marker %s at frame %d.\n",
    marker_name, frame_num);
}

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor rect_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
```

```
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add a rectangle to the stage: */
  rect = clutter_rectangle_new_with_color (&rect_color);
  clutter_actor_set_size (rect, 70, 70);
  clutter_actor_set_position (rect, 50, 100);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
  clutter_actor_show (rect);

  /* Show the stage: */
  clutter_actor_show (stage);

  ClutterTimeline *timeline = clutter_timeline_new(5000 /* milliseconds */);
  clutter_timeline_add_marker_at_time (timeline, "clutter-tutorial", 2000 /* milliseconds *
  g_signal_connect (timeline, "new-frame", G_CALLBACK (on_timeline_new_frame), NULL);
  g_signal_connect (timeline, "marker-reached", G_CALLBACK (on_timeline_marker_reached), NU
  clutter_timeline_set_loop(timeline, TRUE);
  clutter_timeline_start(timeline);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  g_object_unref (timeline);

  return EXIT_SUCCESS;

}
```

6.4. Grouping TimeLines in a Score

A `ClutterScore` allows you to start and stop several timelines at once, or run them in sequence one after the other.

To add a timeline that should start first, call `clutter_score_append()` with NULL for the parent argument. All such timelines will be started when you call `clutter_score_start()`.

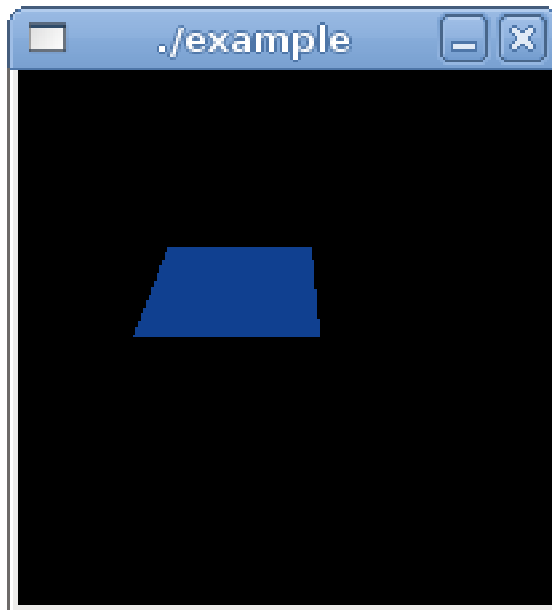
To add a timeline that should be started when a previously added timeline stops, call `clutter_score_append()` with the first timeline for the parent argument.

Reference (http://clutter-project.org/docs/clutter/1.0/ClutterScore.html)

6.5. Example

The following example demonstrates the use of a score containing two timelines, with one starting when the other ends, and the whole score running as a loop. The first timeline rotates the rectangle as in the previous example, and the second timeline moves the rectangle horizontally.

Figure 6-2. Score



Source Code (../../../examples/score)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

ClutterActor *rect = NULL;

gint rotation_angle = 0;
gint color_change_count = 0;

/* Rotate the rectangle and alternate its color. */
void
on_timeline_rotation_new_frame (ClutterTimeline *timeline,
  gint frame_num, gpointer data)
{
  rotation_angle += 1;
```

```
  if(rotation_angle >= 360)
    rotation_angle = 0;

  /* Rotate the rectangle clockwise around the z axis, around it's top-left corner: */
  clutter_actor_set_rotation (rect, CLUTTER_X_AXIS,
    rotation_angle, 0, 0, 0);

  /* Change the color
   * (This is a silly example, making the rectangle flash):
   */
  ++color_change_count;
  if(color_change_count > 100)
    color_change_count = 0;

  if(color_change_count == 0)
  {
    ClutterColor rect_color = { 0xff, 0xff, 0xff, 0x99 };
    clutter_rectangle_set_color (CLUTTER_RECTANGLE (rect), &rect_color);
  }
  else if (color_change_count == 50)
  {
    ClutterColor rect_color = { 0x10, 0x40, 0x90, 0xff };
    clutter_rectangle_set_color (CLUTTER_RECTANGLE (rect), &rect_color);
  }
}

/* Move the rectangle. */
void
on_timeline_move_new_frame (ClutterTimeline *timeline,
  gint frame_num, gpointer data)
{
  gint x_position = clutter_actor_get_x (rect);

  x_position += 1;
  if(x_position >= 150)
    x_position = 0;

  clutter_actor_set_x (rect, x_position);
}


int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor rect_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);
```

```
  /* Add a rectangle to the stage: */
  rect = clutter_rectangle_new_with_color (&rect_color);
  clutter_actor_set_size (rect, 70, 70);
  clutter_actor_set_position (rect, 50, 100);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
  clutter_actor_show (rect);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Create a score and add two timelines to it,
   * so the second timeline starts when the first one stops: */
  ClutterScore *score = clutter_score_new ();
  clutter_score_set_loop (score, TRUE);

  ClutterTimeline *timeline_rotation = clutter_timeline_new (5000 /* milliseconds */);
  g_signal_connect (timeline_rotation, "new-frame", G_CALLBACK (on_timeline_rotation_new_fr
  clutter_score_append (score, NULL, timeline_rotation);

  ClutterTimeline *timeline_move = clutter_timeline_new (5000 /* milliseconds */);
  g_signal_connect (timeline_move, "new-frame", G_CALLBACK (on_timeline_move_new_frame), NU
  clutter_score_append (score,  timeline_rotation, timeline_move);

  clutter_score_start (score);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  g_object_unref (timeline_rotation);
  g_object_unref (timeline_move);
  g_object_unref (score);

  return EXIT_SUCCESS;

}
```

Chapter 7. Animations

7.1. Using Animations

The `clutter_actor_animate()`, `clutter_actor_animate_with_timeline()`, and `clutter_actor_animate_with_alpha()` functions change the properties of a single actor over time, using either a standard `ClutterAnimationMode` or a simple numeric calculation. In many cases this is an easier way to implement animation.

These are convenience functions that use the `ClutterAnimation` object. You should not need to use `ClutterAnimation` directly.

For instance, you could use `clutter_actor_animate()` to fade an actor by changing its "opacity" property, while also moving it to a specified position by changing its "x" and "y" properties, changing all three property values linearly over 1 second (1000 milliseconds). You should specify the values that the properties should reach, regardless of their initial values.

```
clutter_actor_animate (rectangle, CLUTTER_LINEAR, 1000,
  "opacity", 150,
  "x", 100.0,
  "y", 100.0,
  NULL);
```

ClutterActor Reference (http://clutter-project.org/docs/clutter/1.0/ClutterActor.html)

ClutterAnimation Reference (http://clutter-project.org/docs/clutter/1.0/ClutterAnimations.html)

ClutterAnimationMode Reference
(http://clutter-project.org/docs/clutter/1.0/ClutterAnimations.html#ClutterAnimationMode)

7.2. Using Alpha Functions.

You may use a custom calculation callback, instead of a standard `ClutterAnimationMode` enum value such as `CLUTTER_LINEAR`, by using the `clutter_actor_animate_with_alpha()` function, providing a `ClutterAlpha`.

The `ClutterAlpha` object is constructed with a calculation callback and a `ClutterTimeline` which tells it when a new frame needs a new value to be calculated. Your `alpha` callback will need to call `clutter_alpha_get_timeline()` so it can return a value based on the timeline's current progress, using the `clutter_timeline_get_progress()` function.

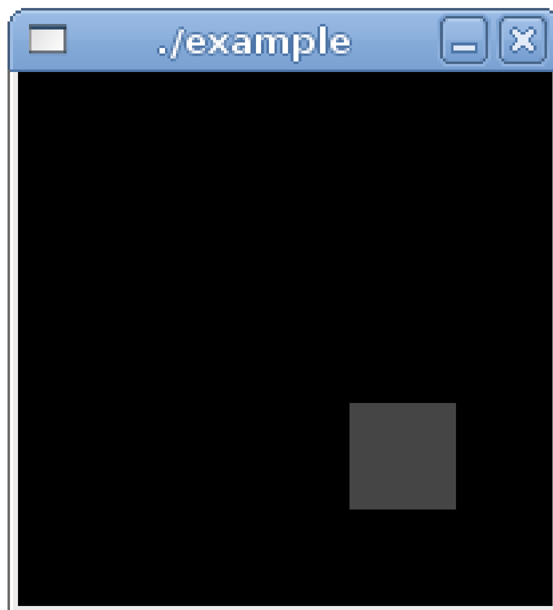
Like actors, `ClutterAlpha` has a "floating references" so you don't need to unref it if you have added it to a behaviour. However, the behaviours do not have a floating reference, so you should unref them when you are finished with them. You might do this at the same time as you unref the timeline, for instance in a signal handler for the timeline's `completed` signal.

See the Behaviours chapter for an example that uses a `ClutterAlpha`.

7.3. Example

The following example demonstrates the use of an animation to achieve a fade and a movement on the same actor, changing a rectangle's opacity while it is moved along a straight line:

Figure 7-1. Animation



Source Code (../../../examples/animation)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

ClutterActor *rect = NULL;

/* This must return a value between 0 and 1.0
 *
 * This will be called as many times per seconds as specified in our call to clutter_timeli
 *
 */
gdouble
on_alpha (ClutterAlpha *alpha, gpointer data)
{
  /* Get the position in the timeline,
   *  so we can base our value upon it:
   */
  ClutterTimeline *timeline = clutter_alpha_get_timeline (alpha);
  return clutter_timeline_get_progress (timeline);
}


int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor rect_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add a rectangle to the stage: */
  rect = clutter_rectangle_new_with_color (&rect_color);
  clutter_actor_set_size (rect, 40, 40);
  clutter_actor_set_position (rect, 10, 10);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
  clutter_actor_show (rect);

  /* Show the stage: */
  clutter_actor_show (stage);

  ClutterTimeline *timeline =  clutter_timeline_new(5000 /* milliseconds */);
  clutter_timeline_set_loop(timeline, TRUE);
  clutter_timeline_start(timeline);

  /* Create a clutter alpha for the animation */
  ClutterAlpha* alpha = clutter_alpha_new_with_func (timeline, &on_alpha, NULL, NULL);
  g_object_unref (timeline);

  /* Create an animation to change the properties */
  ClutterAnimation* animation =
```

```
    clutter_actor_animate_with_alpha (rect, alpha,
      "x", 150.0,
      "y", 150.0,
      "opacity", 0,
      NULL);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  g_object_unref (animation);

  return EXIT_SUCCESS;
}
```

Chapter 8. Behaviours

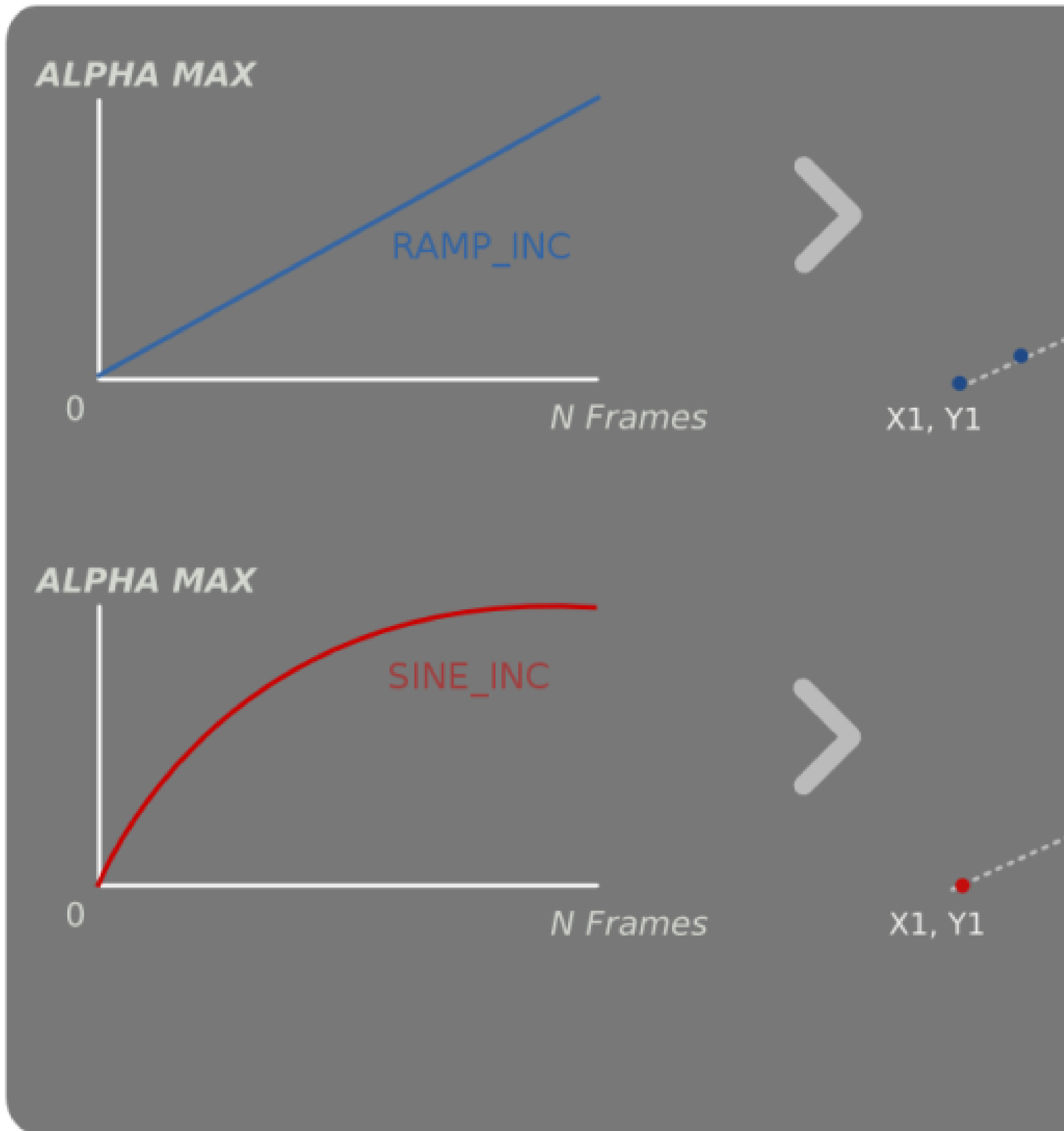
8.1. Using Behaviours

The Animation API is simple but you will often need more control, while still avoiding the complication of using `ClutterTimeline` directly.

Although the `ClutterTimeline`'s `new-frame` signal allows you to set actor properties for each frame, Clutter also provides `Behaviour`s which can change specific properties of one specific actor over time, using a simple numeric calculation. However, unlike the simplified Animation API, using behaviours directly allows you to combine them to control multiple actors simultaneously and allows you to change the parameters of the behaviours while the timeline is running.

For instance, `ClutterBehaviourPath` moves the actor along a specified path, calculating the position on the path once per frame by calling a supplied `alpha` callback. See the Using Alpha Functions section in the Animation chapter.

Figure 8-1. Effects of alpha functions on a path.



ClutterBehaviour Reference (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviour.html)

ClutterAlpha Reference (http://clutter-project.org/docs/clutter/1.0/ClutterAlpha.html)

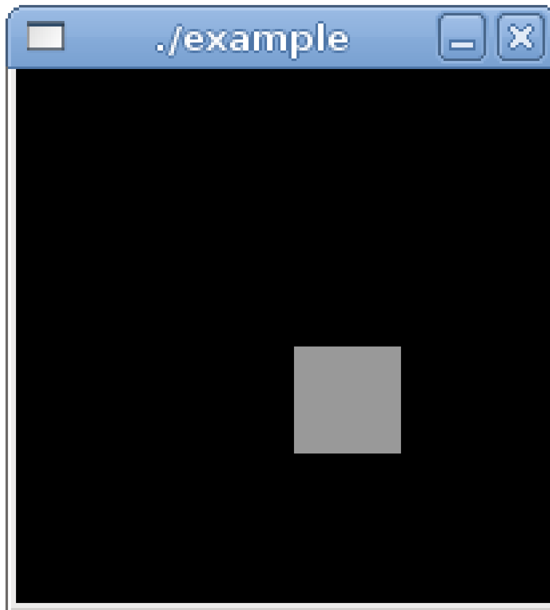
The following standard behaviours are available in Clutter:

- ClutterBehaviourDepth (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviourDepth.html): Moves the actor along the z axis.
- ClutterBehaviourEllipse (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviourEllipse.html): Moves the actor around an ellipse.
- ClutterBehaviourOpacity (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviourOpacity.html): Changes the opacity of the actor.
- ClutterBehaviourPath (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviourPath.html): Moves the actor along straight lines and bezier curves.
- ClutterBehaviourRotate (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviourRotate.html): Rotates the actor.
- ClutterBehaviourScale (http://clutter-project.org/docs/clutter/1.0/ClutterBehaviourScale.html): Scales the actor.

8.2. Example

The following example demonstrates the use of a `ClutterBehaviourPath` with a simple custom alpha callback. This simply moves the rectangle from the top-left to the bottom-right of the canvas at constant speed:

Figure 8-2. Behaviour



Source Code (../../../examples/behaviour)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

ClutterActor *rect = NULL;

/* This must return a value between 0 and 1.0.
 * This will be called as many times per seconds as specified in our call to
 * clutter_timeline_new().
 */
gdouble
on_alpha (ClutterAlpha *alpha, gpointer data)
{
  /* Get the position in the timeline,
   *  so we can base our value upon it:
   */
  ClutterTimeline *timeline = clutter_alpha_get_timeline (alpha);
  return clutter_timeline_get_progress (timeline);
}


int main(int argc, char *argv[])
```

```
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor rect_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add a rectangle to the stage: */
  rect = clutter_rectangle_new_with_color (&rect_color);
  clutter_actor_set_size (rect, 40, 40);
  clutter_actor_set_position (rect, 10, 10);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
  clutter_actor_show (rect);

  /* Show the stage: */
  clutter_actor_show (stage);

  ClutterTimeline *timeline = clutter_timeline_new(5000 /* milliseconds */);
  clutter_timeline_set_loop(timeline, TRUE);
  clutter_timeline_start(timeline);

  /* Instead of our custom callback,
   * we could use a standard callback. For instance, CLUTTER_ALPHA_SINE_INC.
   */
  ClutterAlpha *alpha = clutter_alpha_new_with_func (timeline, &on_alpha, NULL, NULL);

  ClutterKnot knot[2];
  knot[0].x = 10;
  knot[0].y = 10;
  knot[1].x= 150;
  knot[1].y= 150;

  ClutterBehaviour *behaviour = clutter_behaviour_path_new_with_knots (alpha, knot, sizeof(
  clutter_behaviour_apply (behaviour, rect);
  g_object_unref (timeline);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

Chapter 9. Text editing

9.1. ClutterText

Clutter's `ClutterText` actor can display text and allow it to be edited. It doesn't have as much functionality as, for instance, GTK+'s `GtkTextView` widget, but it is enough for displaying information and for simple text entry. Therefore it serves the same purposes as GTK+'s `GtkLabel` or `GtkEntry` widgets, with the addition of multi line editing.

There are three ways you might use a `ClutterText` actor:

- For a simple label, use `clutter_text_new_with_text()` or `clutter_text_set_markup()`. To make it non-editable use `clutter_text_set_editable()`.
- For single line text entry, similar to a `GtkEntry` in a normal GTK+ application, create a `ClutterText` using `clutter_text_new()` and `clutter_text_set_single_line_mode()`. You may call `clutter_text_set_activatable()` and connect to the `activate` signal to react when the user presses `Enter`.
- For full-featured multi-line text editing, `ClutterText` gives you access to the cursor position using the `clutter_text_get/set_cursor_*()` functions and to the selection using the `clutter_text_get/set_selection*()` functions. You can also add and remove text at any position.

ClutterText Reference (http://clutter-project.org/docs/clutter/1.0/ClutterText.html)

> **Note:** When you want the `ClutterText` to be editable you must give it key focus using `clutter_stage_set_key_focus()`.

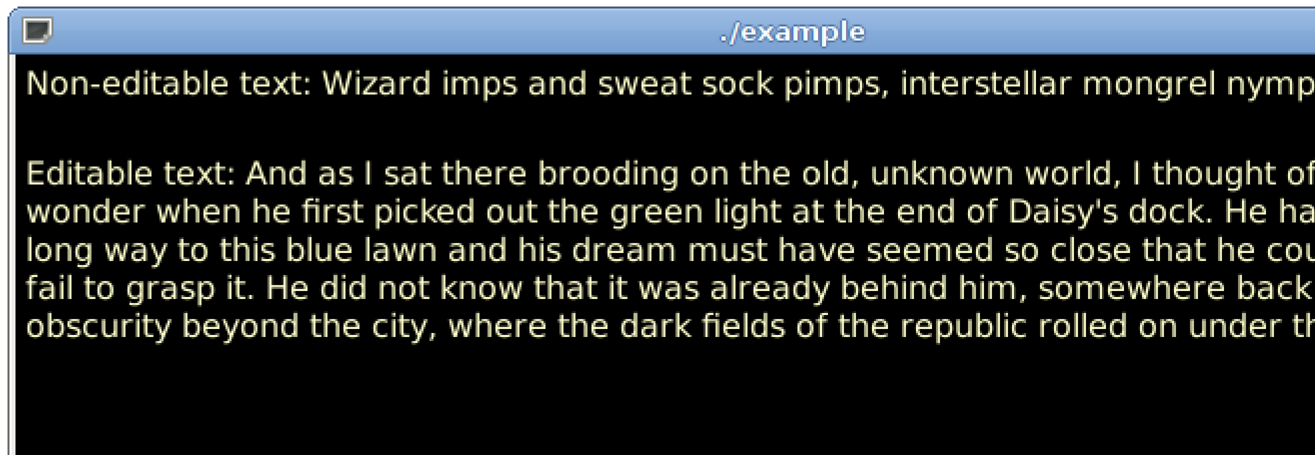
9.1.1. Size Management

To discover the size needed by the text inside a `ClutterText` actor, you may call will the `clutter_actor_get_preferred_height()` function. This will provide the vertical space necessary to display all of the text at the currently specified width. Alternatively, you could call `clutter_actor_get_preferred_width()` to discover the horizontal space necessary for the text at the currently specified height.

9.2. Example

This is a very basic example showing how to use a `ClutterText` for information display or multi-line text editing.

Figure 9-1. ClutterText



Source Code (../../../examples/text)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0xff, 0xff, 0xcc, 0xff };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 800, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);


  /* Add a non-editable text actor to the stage: */
  ClutterActor *text = clutter_text_new ();

  /* Setup text properties */
```

```
clutter_text_set_color (CLUTTER_TEXT (text), &actor_color);
clutter_text_set_text (CLUTTER_TEXT (text),
  "Non-editable text: Wizard imps and sweat sock pimps, interstellar mongrel nymphs.");
clutter_text_set_font_name (CLUTTER_TEXT (text), "Sans 12");
clutter_text_set_editable (CLUTTER_TEXT (text), FALSE);
clutter_text_set_line_wrap (CLUTTER_TEXT (text), FALSE);

/* Discover the preferred height and use that height: */
float min_height = 0;
float natural_height = 0;
clutter_actor_get_preferred_height (text, 750, &min_height,
  &natural_height);
clutter_actor_set_size (text, 750, natural_height);

clutter_actor_set_position (text, 5, 5);
clutter_container_add_actor (CLUTTER_CONTAINER (stage), text);
clutter_actor_show (text);


/* Add a multi-line editable text actor to the stage: */
text = clutter_text_new ();

/* Setup text properties */
clutter_text_set_color (CLUTTER_TEXT (text), &actor_color);
clutter_text_set_text (CLUTTER_TEXT (text),
  "Editable text: And as I sat there brooding on the old, unknown world, I thought of "
   "Gatsby's wonder when he first picked out the green light at the end of "
   "Daisy's dock. He had come a long way to this blue lawn and his dream "
   "must have seemed so close that he could hardly fail to grasp it. He did "
   "not know that it was already behind him, somewhere back in that vast "
   "obscurity beyond the city, where the dark fields of the republic rolled "
   "on under the night.");
clutter_text_set_font_name (CLUTTER_TEXT (text), "Sans 12");
clutter_text_set_editable (CLUTTER_TEXT (text), TRUE);
clutter_text_set_line_wrap (CLUTTER_TEXT (text), TRUE);

/* Discover the preferred height and use that height: */
min_height = 0;
natural_height = 0;
clutter_actor_get_preferred_height (text, 750, &min_height,
  &natural_height);
clutter_actor_set_size (text, 750, natural_height);

clutter_actor_set_position (text, 5, 50);
clutter_container_add_actor (CLUTTER_CONTAINER (stage), text);
clutter_actor_show (text);

/* Set focus to handle key presses on the stage: */
clutter_stage_set_key_focus (CLUTTER_STAGE (stage), text);


/* Show the stage: */
clutter_actor_show (stage);
```

```
  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

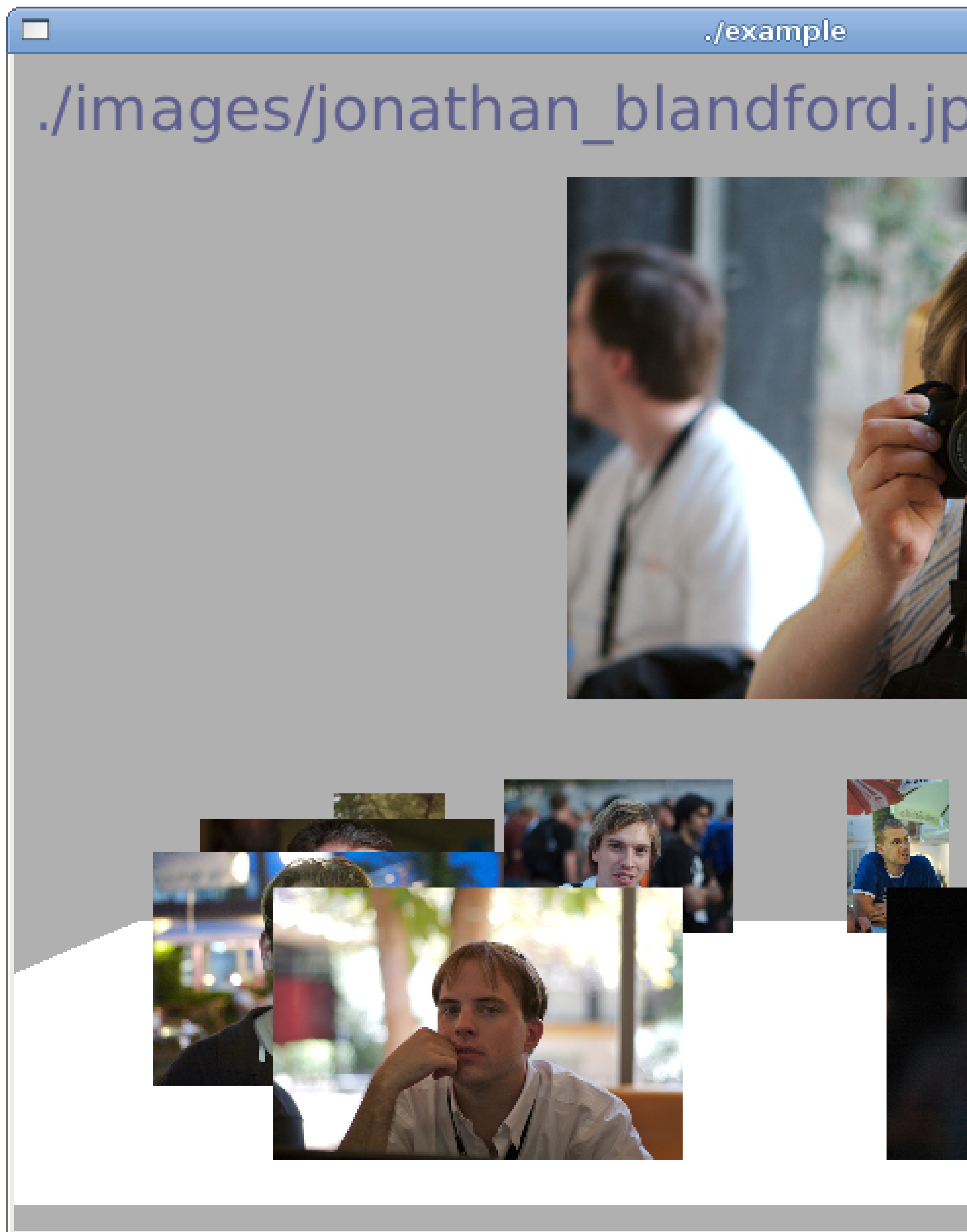
Chapter 10. Full Example

This example loads images from a directory and displays them in a rotating ellipse. You may click an image to bring it to the front. When the image has rotated to the front it will move up while increasing in size, and the file path will appear at the top of the window.

This is larger than the examples used so far, with multiple timelines and multiple behaviours affecting multiple actors. However, it’s still a relatively simple example. A real application would need to be more flexible and have more functionality.

TODO: Make this prettier. Use containers to do that.

Figure 10-1. Full Example



Source Code (../../../examples/full_example)

File: `main.c`

```
#include <clutter/clutter.h>
#include <stdlib.h>

ClutterActor *stage = NULL;

/* For showing the filename: */
ClutterActor *label_filename = NULL;

/* For rotating all images around an ellipse: */
ClutterTimeline *timeline_rotation = NULL;

/* For moving one image up and scaling it: */
ClutterTimeline *timeline_moveup = NULL;
ClutterBehaviour *behaviour_scale = NULL;
ClutterBehaviour *behaviour_path = NULL;
ClutterBehaviour *behaviour_opacity = NULL;

/* The y position of the ellipse of images: */
const gint ELLIPSE_Y = 390;
const gint ELLIPSE_HEIGHT = 450; /* The distance from front to back when it’s rotated 90 de
const gint IMAGE_HEIGHT = 100;

static gboolean
on_texture_button_press (ClutterActor *actor, ClutterEvent *event, gpointer data);

const double angle_step = 30;

typedef struct Item
{
  ClutterActor *actor;
  ClutterBehaviour *ellipse_behaviour;
  gchar* filepath;
}
Item;

Item* item_at_front = NULL;

GSList *list_items = 0;

void on_foreach_clear_list_items(gpointer data, gpointer user_data)
{
  Item* item = (Item*)data;

  /* We don’t need to unref the actor because the floating reference was taken by the stage
  g_object_unref (item->ellipse_behaviour);
```

```
  g_free (item->filepath);
  g_free (item);
}

void scale_texture_default(ClutterActor *texture)
{
  int pixbuf_height = 0;
  clutter_texture_get_base_size (CLUTTER_TEXTURE (texture), NULL, &pixbuf_height);

  const gdouble scale = pixbuf_height ? IMAGE_HEIGHT /  (gdouble)pixbuf_height : 0;
  clutter_actor_set_scale (texture, scale, scale);
}

void load_images(const gchar* directory_path)
{
  g_return_if_fail(directory_path);

  /* Clear any existing images: */
  g_slist_foreach (list_items, on_foreach_clear_list_items, NULL);
  g_slist_free (list_items);

  /* Create a new list: */
  list_items = NULL;

  /* Discover the images in the directory: */
  GError *error = NULL;
  GDir* dir = g_dir_open (directory_path, 0, &error);
  if(error)
  {
    g_warning("g_dir_open() failed: %s\n", error->message);
    g_clear_error(&error);
    return;
  }

  const gchar* filename = NULL;
  while ( (filename = g_dir_read_name(dir)) )
  {
    gchar* path = g_build_filename (directory_path, filename, NULL);

    /* Try to load the file as an image: */
    ClutterActor *actor = clutter_texture_new_from_file (path, NULL);
    if(actor)
    {
      Item* item = g_new0(Item, 1);

      item->actor = actor;
      item->filepath = g_strdup(path);

      /* Make sure that all images are shown with the same height: */
      scale_texture_default (item->actor);

      list_items = g_slist_append (list_items, item);
    }
```

```
    g_free (path);
  }
}


void add_to_ellipse_behaviour(ClutterTimeline *timeline_rotation, gdouble start_angle, Item
{
  g_return_if_fail (timeline_rotation);

  ClutterAlpha *alpha = clutter_alpha_new_full (timeline_rotation, CLUTTER_EASE_OUT_SINE);

  item->ellipse_behaviour = clutter_behaviour_ellipse_new (alpha,
    320, ELLIPSE_Y, /* x, y */
    ELLIPSE_HEIGHT, ELLIPSE_HEIGHT, /* width, height */
    CLUTTER_ROTATE_CW,
    start_angle, start_angle + 360);
  clutter_behaviour_ellipse_set_angle_tilt (CLUTTER_BEHAVIOUR_ELLIPSE (item->ellipse_behavi
    CLUTTER_X_AXIS, -90);
  /* Note that ClutterAlpha has a floating reference, so we don't need to unref it. */

  clutter_behaviour_apply (item->ellipse_behaviour, item->actor);
}

void add_image_actors()
{
  int x = 20;
  int y = 0;
  gdouble angle = 0;
  GSList *list = list_items;
  while (list)
  {
    /* Add the actor to the stage: */
    Item *item = (Item*)list->data;
    ClutterActor *actor = item->actor;
    clutter_container_add_actor (CLUTTER_CONTAINER (stage), actor);

    /* Set an initial position: */
    clutter_actor_set_position (actor, x, y);
    y += 100;

    /* Allow the actor to emit events.
     * By default only the stage does this.
     */
    clutter_actor_set_reactive (actor, TRUE);

    /* Connect signal handlers for events: */
    g_signal_connect (actor, "button-press-event",
      G_CALLBACK (on_texture_button_press), item);

    add_to_ellipse_behaviour (timeline_rotation, angle, item);
    angle += angle_step;
```

```
    clutter_actor_show (actor);

    list = g_slist_next (list);
  }
}

gdouble angle_in_360(gdouble angle)
{
  gdouble result = angle;
  while(result >= 360)
    result -= 360;

  return result;
}

/* This signal handler is called when the item has finished
 * moving up and increasing in size.
 */
void on_timeline_moveup_completed(ClutterTimeline* timeline, gpointer user_data)
{
  /* Unref this timeline because we have now finished with it: */
  g_object_unref (timeline_moveup);
  timeline_moveup = NULL;

  g_object_unref (behaviour_scale);
  behaviour_scale = NULL;

  g_object_unref (behaviour_path);
  behaviour_path = NULL;

  g_object_unref (behaviour_opacity);
  behaviour_opacity = NULL;
}

/* This signal handler is called when the items have completely
 * rotated around the ellipse.
 */
void on_timeline_rotation_completed(ClutterTimeline* timeline, gpointer user_data)
{
  /* All the items have now been rotated so that the clicked item is at the
   * front.  Now we transform just this one item gradually some more, and
   * show the filename.
   */
  /* Transform the image: */
  ClutterActor *actor = item_at_front->actor;
  timeline_moveup = clutter_timeline_new(1000 /* milliseconds */);
  ClutterAlpha *alpha =
    clutter_alpha_new_full (timeline_moveup, CLUTTER_EASE_OUT_SINE);

  /* Scale the item from its normal scale to approximately twice the normal scale: */
  gdouble scale_start = 0;
  clutter_actor_get_scale (actor, &scale_start, NULL);
  const gdouble scale_end = scale_start * 1.8;
```

```
  behaviour_scale = clutter_behaviour_scale_new (alpha,
                                                 scale_start, scale_start,
                                                 scale_end, scale_end);
  clutter_behaviour_apply (behaviour_scale, actor);

  /* Move the item up the y axis: */
  ClutterKnot knots[2];
  knots[0].x = clutter_actor_get_x (actor);
  knots[0].y = clutter_actor_get_y (actor);
  knots[1].x = knots[0].x;
  knots[1].y = knots[0].y - 250;
  behaviour_path =
    clutter_behaviour_path_new_with_knots (alpha, knots, G_N_ELEMENTS(knots));
  clutter_behaviour_apply (behaviour_path, actor);

  /* Show the filename gradually: */
  clutter_text_set_text (CLUTTER_TEXT (label_filename), item_at_front->filepath);
  behaviour_opacity = clutter_behaviour_opacity_new (alpha, 0, 255);
  clutter_behaviour_apply (behaviour_opacity, label_filename);

  /* Start the timeline and handle its "completed" signal so we can unref it. */
  g_signal_connect (timeline_moveup, "completed", G_CALLBACK (on_timeline_moveup_completed)
  clutter_timeline_start (timeline_moveup);

  /* Note that ClutterAlpha has a floating reference so we don't need to unref it. */
}

void rotate_all_until_item_is_at_front(Item *item)
{
  g_return_if_fail (item);

  clutter_timeline_stop(timeline_rotation);

  /* Stop the other timeline in case that is active at the same time: */
  if(timeline_moveup)
    clutter_timeline_stop (timeline_moveup);

  clutter_actor_set_opacity (label_filename, 0);

  /* Get the item's position in the list: */
  const gint pos = g_slist_index (list_items, item);
  g_assert (pos != -1);

  if(!item_at_front && list_items)
    item_at_front = (Item*)list_items->data;

  gint pos_front = 0;
  if(item_at_front)
     pos_front = g_slist_index (list_items, item_at_front);
  g_assert (pos_front != -1);

  /* const gint pos_offset_before_start = pos_front - pos; */
```

```
/* Calculate the end angle of the first item: */
const gdouble angle_front = 180;
gdouble angle_start = angle_front - (angle_step * pos_front);
angle_start = angle_in_360 (angle_start);
gdouble angle_end = angle_front - (angle_step * pos);

gdouble angle_diff = 0;

/* Set the end angles: */
GSList *list = list_items;
while (list)
{
  Item *this_item = (Item*)list->data;

  /* Reset its size: */
  scale_texture_default (this_item->actor);

  angle_start = angle_in_360 (angle_start);
  angle_end = angle_in_360 (angle_end);

  /* Move 360 instead of 0
   * when moving for the first time,
   * and when clicking on something that is already at the front.
   */
  if(item_at_front == item)
    angle_end += 360;

  clutter_behaviour_ellipse_set_angle_start (
    CLUTTER_BEHAVIOUR_ELLIPSE (this_item->ellipse_behaviour), angle_start);
  clutter_behaviour_ellipse_set_angle_end (
    CLUTTER_BEHAVIOUR_ELLIPSE (this_item->ellipse_behaviour), angle_end);

  if(this_item == item)
  {
    if(angle_start < angle_end)
      angle_diff =  angle_end - angle_start;
    else
      angle_diff = 360 - (angle_start - angle_end);

    /* printf("  debug: angle diff=%f\n", angle_diff); */

  }

  /* TODO: Set the number of frames, depending on the angle.
   * otherwise the actor will take the same amount of time to reach
   * the end angle regardless of how far it must move, causing it to
   * move very slowly if it does not have far to move.
   */
  angle_end += angle_step;
  angle_start += angle_step;
  list = g_slist_next (list);
}
```

```
  /* Set the number of frames to be proportional to the distance to travel,
     so the speed is always the same: */
  gint pos_to_move = 0;
  if(pos_front < pos)
  {
     const gint count = g_slist_length (list_items);
     pos_to_move = count + (pos - pos_front);
  }
  else
  {
     pos_to_move = pos_front - pos;
  }

  clutter_timeline_set_duration (timeline_rotation, angle_diff * 0.2);

  /* Remember what item will be at the front when this timeline finishes: */
  item_at_front = item;

  clutter_timeline_start (timeline_rotation);
}

static gboolean
on_texture_button_press (ClutterActor *actor, ClutterEvent *event, gpointer user_data)
{
  /* Ignore the events if the timeline_rotation is running (meaning, if the objects are mov
   * to simplify things:
   */
  if(timeline_rotation && clutter_timeline_is_playing (timeline_rotation))
  {
    printf("on_texture_button_press(): ignoring\n");
    return FALSE;
  }
  else
    printf("on_texture_button_press(): handling\n");

  Item *item = (Item*)user_data;
  rotate_all_until_item_is_at_front (item);

  return TRUE;
}

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0xB0, 0xB0, 0xB0, 0xff }; /* light gray */

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 800, 600);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);
```

```
/* Create and add a label actor, hidden at first: */
label_filename = clutter_text_new ();
ClutterColor label_color = { 0x60, 0x60, 0x90, 0xff }; /* blueish */
clutter_text_set_color (CLUTTER_TEXT (label_filename), &label_color);
clutter_text_set_font_name (CLUTTER_TEXT (label_filename), "Sans 24");
clutter_actor_set_position (label_filename, 10, 10);
clutter_actor_set_opacity (label_filename, 0);
clutter_container_add_actor (CLUTTER_CONTAINER (stage), label_filename);
clutter_actor_show (label_filename);

/* Add a plane under the ellipse of images: */
ClutterColor rect_color = { 0xff, 0xff, 0xff, 0xff }; /* white */
ClutterActor *rect = clutter_rectangle_new_with_color (&rect_color);
clutter_actor_set_height (rect, ELLIPSE_HEIGHT + 20);
clutter_actor_set_width (rect, clutter_actor_get_width (stage) + 100);
/* Position it so that its center is under the images: */
clutter_actor_set_position (rect,
  -(clutter_actor_get_width (rect) - clutter_actor_get_width (stage)) / 2,
  ELLIPSE_Y + IMAGE_HEIGHT - (clutter_actor_get_height (rect) / 2));
/* Rotate it around its center: */
clutter_actor_set_rotation (rect, CLUTTER_X_AXIS, -90, 0, (clutter_actor_get_height (rect
clutter_container_add_actor (CLUTTER_CONTAINER (stage), rect);
clutter_actor_show (rect);

/* Show the stage: */
clutter_actor_show (stage);

timeline_rotation = clutter_timeline_new(2000 /* milliseconds */);
g_signal_connect (timeline_rotation, "completed", G_CALLBACK (on_timeline_rotation_comple

/* Add an actor for each image: */
load_images ("./images/");
add_image_actors ();

/* clutter_timeline_set_loop(timeline_rotation, TRUE); */

/* Move them a bit to start with: */
if(list_items)
  rotate_all_until_item_is_at_front ((Item*)list_items->data);


/* Start the main loop, so we can respond to events: */
clutter_main ();


/* Free the list items and the list: */
g_slist_foreach(list_items, on_foreach_clear_list_items, NULL);
g_slist_free (list_items);

g_object_unref (timeline_rotation);

return EXIT_SUCCESS;
```

```
}
```

Appendix A. Implementing Actors

A.1. Implementing Simple Actors

If the standard Clutter actors don't meet all your needs then you may create your own custom actor objects. Implementing a custom actor is much like implementing any new GObject type. You may use the `G_DEFINE_TYPE` macro to specify that the type is derived from `ClutterActor`. For instance:

```
G_DEFINE_TYPE (ClutterTriangle, clutter_triangle, CLUTTER_TYPE_ACTOR);
```

You should then specify your object's implementation of the `ClutterActor::paint()` virtual function in your `class_init` function:

```
static void
clutter_triangle_class_init (ClutterTriangleClass *klass)
{
  ClutterActorClass *actor_class = CLUTTER_ACTOR_CLASS (klass);

  actor_class->paint = clutter_triangle_paint;

  ...
}
```

Your `ClutterActor::paint()` implementation should use the OpenGL API to actually paint something. You will probably need some information from your object's generic `ClutterActor` base class, for instance by calling `clutter_actor_get_geometry()` and `clutter_actor_get_opacity()`, and by using your object's specific property values.

To make your code work with both OpenGL ES and regular OpenGL (and maybe even future Clutter backends), you may wish to use Clutter's `cogl` abstraction API which provides functions such as `cogl_rectangle()` and `cogl_push_matrix()`. You can also detect whether the platform has support for either the OpenGL or OpenGL ES API by ifdefing for `CLUTTER_COGL_HAS_GL` or `CLUTTER_COGL_HAS_GLES`.

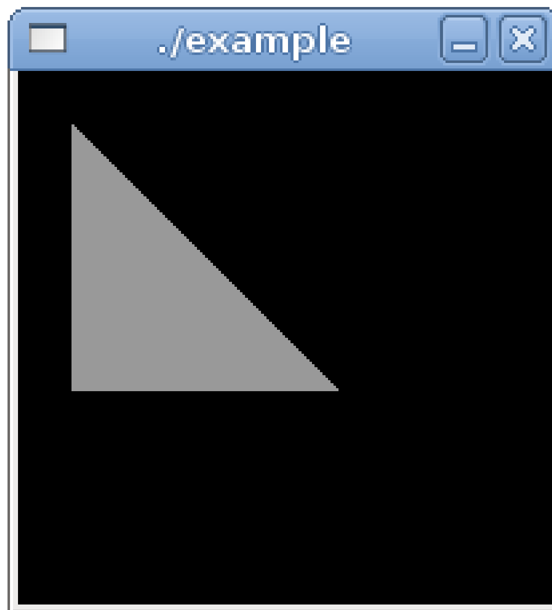
You should also implement the `ClutterActor::pick()` virtual function, painting a silhouette of your actor in the provided color. Clutter uses this to draw each actor's silhouette offscreen in a unique color, using the color to quickly identify the actor under the cursor. If your actor is simple then you can probably reuse the code from your paint() implementation.

Most of the rest of `ClutterActor`'s virtual functions don't need to be reimplemented, because a suitable default implemention exists in `ClutterActor`. For instance, `show()`, `show_all()`, `hide()`, `hide_all()`, `request_coord()`.

A.2. Example

The following example demonstrates the implementation of a new triangle Actor type.

Figure A-1. Behaviour



Source Code (../../../examples/custom_actor)

File: `triangle_actor.h`

```
#ifndef _CLUTTER_TUTORIAL_TRIANGLE_ACTOR_H
#define _CLUTTER_TUTORIAL_TRIANGLE_ACTOR_H

#include <glib-object.h>
#include <clutter/clutter.h>

G_BEGIN_DECLS

#define CLUTTER_TYPE_TRIANGLE clutter_triangle_get_type()
```

```
#define CLUTTER_TRIANGLE(obj) \
  (G_TYPE_CHECK_INSTANCE_CAST ((obj), \
  CLUTTER_TYPE_TRIANGLE, ClutterTriangle))

#define CLUTTER_TRIANGLE_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_CAST ((klass), \
  CLUTTER_TYPE_TRIANGLE, ClutterTriangleClass))

#define CLUTTER_IS_TRIANGLE(obj) \
  (G_TYPE_CHECK_INSTANCE_TYPE ((obj), \
  CLUTTER_TYPE_TRIANGLE))

#define CLUTTER_IS_TRIANGLE_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_TYPE ((klass), \
  CLUTTER_TYPE_TRIANGLE))

#define CLUTTER_TRIANGLE_GET_CLASS(obj) \
  (G_TYPE_INSTANCE_GET_CLASS ((obj), \
  CLUTTER_TYPE_TRIANGLE, ClutterTriangleClass))

typedef struct _ClutterTriangle        ClutterTriangle;
typedef struct _ClutterTriangleClass   ClutterTriangleClass;
typedef struct _ClutterTrianglePrivate ClutterTrianglePrivate;

struct _ClutterTriangle
{
  ClutterActor           parent;

  /*< private >*/
  ClutterTrianglePrivate *priv;
};

struct _ClutterTriangleClass
{
  ClutterActorClass parent_class;
};

GType clutter_triangle_get_type (void) G_GNUC_CONST;

ClutterActor *clutter_triangle_new              (void);
ClutterActor *clutter_triangle_new_with_color   (const ClutterColor *color);

void          clutter_triangle_get_color        (ClutterTriangle   *triangle,
                                                  ClutterColor       *color);
void          clutter_triangle_set_color        (ClutterTriangle   *triangle,
  const ClutterColor *color);

G_END_DECLS

#endif
```

File: `main.c`

```
#include <clutter/clutter.h>
#include "triangle_actor.h"
#include <stdlib.h>


int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0xff, 0xff, 0xff, 0x99 };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add our custom actor to the stage: */
  ClutterActor *actor = clutter_triangle_new_with_color (&actor_color);
  clutter_actor_set_size (actor, 100, 100);
  clutter_actor_set_position (actor, 20, 20);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), actor);
  clutter_actor_show (actor);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

File: `triangle_actor.c`

```
#include "triangle_actor.h"

#include <cogl/cogl.h>

G_DEFINE_TYPE (ClutterTriangle, clutter_triangle, CLUTTER_TYPE_ACTOR);

enum
{
  PROP_0,

  PROP_COLOR
};

#define CLUTTER_TRIANGLE_GET_PRIVATE(obj) \
(G_TYPE_INSTANCE_GET_PRIVATE ((obj), CLUTTER_TYPE_TRIANGLE, ClutterTrianglePrivate))
```

```
struct _ClutterTrianglePrivate
{
  ClutterColor color;
};

static void
do_triangle_paint (ClutterActor *self, const CoglColor *color)
{
  ClutterTriangle        *triangle;
  ClutterTrianglePrivate *priv;
  ClutterGeometry         geom;
  float                  coords[6];

  triangle = CLUTTER_TRIANGLE(self);
  priv = triangle->priv;

  clutter_actor_get_geometry (self, &geom);

  cogl_set_source_color (color);

  /* Paint a triangle:
   *
   * The parent paint call will have translated us into position so
   * paint from 0, 0 */
  coords[0] = 0;
  coords[1] = 0;

  coords[2] = 0;
  coords[3] = geom.height;

  coords[4] = geom.width;
  coords[5] = geom.height;

  cogl_path_polygon (coords, 3);
  cogl_path_fill ();
}

static void
clutter_triangle_paint (ClutterActor *self)
{
  ClutterTrianglePrivate *priv;
  CoglColor color;

  priv = CLUTTER_TRIANGLE(self)->priv;

  /* Paint the triangle with the actor's color: */
  cogl_color_set_from_4ub(&color,
                          priv->color.red,
                          priv->color.green,
                          priv->color.blue,
                          clutter_actor_get_opacity (self));

  do_triangle_paint (self, &color);
```

```
}

static void
clutter_triangle_pick (ClutterActor *self, const ClutterColor *color)
{
  CoglColor coglcolor;
  /* Paint the triangle with the pick color, offscreen.
     This is used by Clutter to detect the actor under the cursor
     by identifying the unique color under the cursor. */
  cogl_color_set_from_4ub(&coglcolor,
                          color->red, color->green, color->blue, color->alpha);
  do_triangle_paint (self, &coglcolor);
}

static void
clutter_triangle_set_property (GObject      *object,
guint         prop_id,
const GValue *value,
GParamSpec   *pspec)
{
  ClutterTriangle *triangle = CLUTTER_TRIANGLE(object);

  switch (prop_id)
    {
    case PROP_COLOR:
      clutter_triangle_set_color (triangle, g_value_get_boxed (value));
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
      break;
  }
}

static void
clutter_triangle_get_property (GObject    *object,
guint       prop_id,
GValue     *value,
GParamSpec *pspec)
{
  ClutterTriangle *triangle = CLUTTER_TRIANGLE(object);
  ClutterColor      color;

  switch (prop_id)
    {
    case PROP_COLOR:
      clutter_triangle_get_color (triangle, &color);
      g_value_set_boxed (value, &color);
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
      break;
    }
}
```

```
static void
clutter_triangle_finalize (GObject *object)
{
  G_OBJECT_CLASS (clutter_triangle_parent_class)->finalize (object);
}

static void
clutter_triangle_dispose (GObject *object)
{
  G_OBJECT_CLASS (clutter_triangle_parent_class)->dispose (object);
}


static void
clutter_triangle_class_init (ClutterTriangleClass *klass)
{
  GObjectClass        *gobject_class = G_OBJECT_CLASS (klass);
  ClutterActorClass *actor_class = CLUTTER_ACTOR_CLASS (klass);

  /* Provide implementations for ClutterActor vfuncs: */
  actor_class->paint = clutter_triangle_paint;
  actor_class->pick = clutter_triangle_pick;

  gobject_class->finalize     = clutter_triangle_finalize;
  gobject_class->dispose      = clutter_triangle_dispose;
  gobject_class->set_property = clutter_triangle_set_property;
  gobject_class->get_property = clutter_triangle_get_property;

  /**
   * ClutterTriangle:color:
   *
   * The color of the triangle.
   */
  g_object_class_install_property (gobject_class,
                                   PROP_COLOR,
                                   g_param_spec_boxed ("color",
                                                       "Color",
                                                       "The color of the triangle",
                                                       CLUTTER_TYPE_COLOR,
                                                       G_PARAM_READABLE | G_PARAM_WRITABLE)

  g_type_class_add_private (gobject_class, sizeof (ClutterTrianglePrivate));
}

static void
clutter_triangle_init (ClutterTriangle *self)
{
  ClutterTrianglePrivate *priv;

  self->priv = priv = CLUTTER_TRIANGLE_GET_PRIVATE (self);
```

```
  priv->color.red = 0xff;
  priv->color.green = 0xff;
  priv->color.blue = 0xff;
  priv->color.alpha = 0xff;
}

/**
 * clutter_triangle_new:
 *
 * Creates a new #ClutterActor with a rectangular shape.
 *
 * Return value: a new #ClutterActor
 */
ClutterActor*
clutter_triangle_new (void)
{
  return g_object_new (CLUTTER_TYPE_TRIANGLE, NULL);
}

/**
 * clutter_triangle_new_with_color:
 * @color: a #ClutterColor
 *
 * Creates a new #ClutterActor with a rectangular shape
 * and with @color.
 *
 * Return value: a new #ClutterActor
 */
ClutterActor *
clutter_triangle_new_with_color (const ClutterColor *color)
{
  return g_object_new (CLUTTER_TYPE_TRIANGLE,
       "color", color,
       NULL);
}

/**
 * clutter_triangle_get_color:
 * @triangle: a #ClutterTriangle
 * @color: return location for a #ClutterColor
 *
 * Retrieves the color of @triangle.
 */
void
clutter_triangle_get_color (ClutterTriangle *triangle,
     ClutterColor     *color)
{
  ClutterTrianglePrivate *priv;

  g_return_if_fail (CLUTTER_IS_TRIANGLE (triangle));
  g_return_if_fail (color != NULL);

  priv = triangle->priv;
```

```
  color->red = priv->color.red;
  color->green = priv->color.green;
  color->blue = priv->color.blue;
  color->alpha = priv->color.alpha;
}

/**
 * clutter_triangle_set_color:
 * @triangle: a #ClutterTriangle
 * @color: a #ClutterColor
 *
 * Sets the color of @triangle.
 */
void
clutter_triangle_set_color (ClutterTriangle   *triangle,
     const ClutterColor *color)
{
  ClutterTrianglePrivate *priv;

  g_return_if_fail (CLUTTER_IS_TRIANGLE (triangle));
  g_return_if_fail (color != NULL);

  g_object_ref (triangle);

  priv = triangle->priv;

  priv->color.red = color->red;
  priv->color.green = color->green;
  priv->color.blue = color->blue;
  priv->color.alpha = color->alpha;

  clutter_actor_set_opacity (CLUTTER_ACTOR (triangle),
          priv->color.alpha);

  if (CLUTTER_ACTOR_IS_VISIBLE (CLUTTER_ACTOR (triangle)))
    clutter_actor_queue_redraw (CLUTTER_ACTOR (triangle));

  g_object_notify (G_OBJECT (triangle), "color");
  g_object_unref (triangle);
}
```

A.3. Implementing Container Actors

You can create container actors that arrange child actors by implementing the ClutterContainer interface in your actor. You may use the `G_DEFINE_TYPE_WITH_CODE` macro to specify that the type is derived from `ClutterActor` and also implements the `ClutterContainer` interface. For instance:

```
static void clutter_container_iface_init (ClutterContainerIface *iface);

G_DEFINE_TYPE_WITH_CODE (ExampleContainer, example_container, CLUTTER_TYPE_ACTOR,
                         G_IMPLEMENT_INTERFACE (CLUTTER_TYPE_CONTAINER,
                                                clutter_container_iface_init));
```

A.3.1. ClutterActor virtual functions to implement

If your container should control the position or size of its children then it should implement the `ClutterActor`'s `allocate()`, `get_preferred_width()` and `get_preferred_height()` virtual functions.

For instance, your `allocate()` implementation should use the provided allocation to layout its child actors, by calling `clutter_actor_allocate()` on each child actor with appropriate allocations.

Your `get_preferred_width()` and `get_preferred_height()` implementations should return the size desired by your container, by providing both the minimum and natural size as output parameters. Depending on your container, this might be dependent on the child actors. By calling `clutter_actor_get_preferred_size()` you can discover the preferred height and width of a child actor.

Your actor should implement one of two geometry management modes - either height for width (`CLUTTER_REQUEST_HEIGHT_FOR_WIDTH`) or width for height `CLUTTER_REQUEST_WIDTH_FOR_HEIGHT`) and should set its `request-mode` property to indicate which one it uses. For instance, in height-for-width mode, the parent actor first asks for the preferred height and then asks for a preferred width appropriate for that height. `clutter_actor_get_preferred_size()` checks this property and then calls either `clutter_actor_get_preferred_width()` or `clutter_actor_get_preferred_height()` in the correct sequence.

You should implement the `paint()` function, usually calling `clutter_actor_paint()` on the child actors. All containers should also implement the `pick()` function, calling `clutter_actor_pick()` on each child actor.

See the Custom Actor section for more details these virtual functions.

A.3.2. ClutterContainer virtual functions to implement

Your container implementation should also implement some of the `ClutterContainer` virtual functions so that the container's children will be affected appropriately when functions are called on the container.

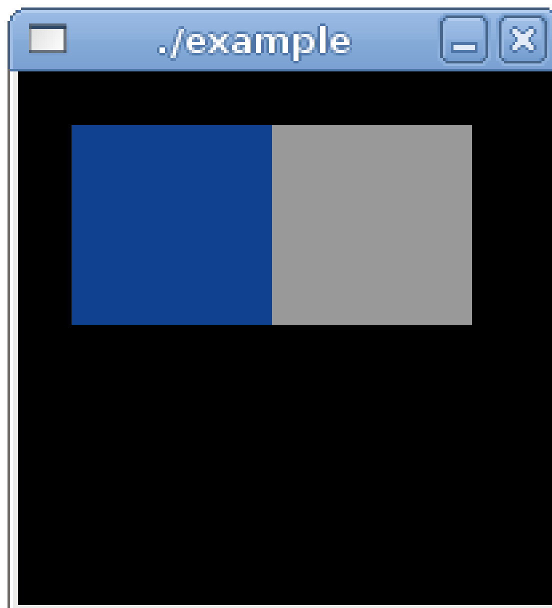
For instance, your `add()` and `remove()` implementions should manage your container's internal list of child actors and might need to trigger repositioning or resizing of the child actors by calling `clutter_actor_queue_relayout()`.

Your `foreach` implementation would simply call the provided callback on your container's list of child actors. Note that your container could actually contain additional actors that are not considered to be child actors for the purposes of add(), remove(), and foreach().

A.4. Example

The following example demonstrates the implementation of a box container that lays its child actors out horizontally. A real container should probably allow optional padding around the container and spacing between the child actors. You might also want to allow some child actors to expand to fill the available space, or align differently inside the container.

Figure A-2. Behaviour



Source Code (../../../examples/custom_container)

File: `examplebox.h`

```
#ifndef __EXAMPLE_BOX_H__
#define __EXAMPLE_BOX_H__

#include <clutter/clutter.h>

G_BEGIN_DECLS

#define EXAMPLE_TYPE_BOX                (example_box_get_type ())
#define EXAMPLE_BOX(obj)                (G_TYPE_CHECK_INSTANCE_CAST ((obj), EXAMPLE_TYPE_BO
#define EXAMPLE_IS_BOX(obj)             (G_TYPE_CHECK_INSTANCE_TYPE ((obj), EXAMPLE_TYPE_BO
#define EXAMPLE_BOX_CLASS(klass)        (G_TYPE_CHECK_CLASS_CAST ((klass), EXAMPLE_TYPE_BOX
#define EXAMPLE_IS_BOX_CLASS(klass)     (G_TYPE_CHECK_CLASS_TYPE ((klass), EXAMPLE_TYPE_BOX
#define EXAMPLE_BOX_GET_CLASS(obj)      (G_TYPE_INSTANCE_GET_CLASS ((obj), EXAMPLE_TYPE_BOX


typedef struct _ExampleBoxChild         ExampleBoxChild;
typedef struct _ExampleBox              ExampleBox;
typedef struct _ExampleBoxClass         ExampleBoxClass;

struct _ExampleBox
{
  /*< private >*/
  ClutterActor parent_instance;

  /* List of ExampleBoxChild structures */
  GList *children;
};

struct _ExampleBoxClass
{
  /*< private >*/
  ClutterActorClass parent_class;
};


GType example_box_get_type (void) G_GNUC_CONST;

ClutterActor *example_box_new (void);
void example_box_pack (ExampleBox*box, ClutterActor *actor);
void example_box_remove_all (ExampleBox *box);

G_END_DECLS

#endif /* __EXAMPLE_BOX_H__ */
```

File: `main.c`

```
#include <clutter/clutter.h>
#include "examplebox.h"
#include <stdlib.h>


int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0xff, 0xff, 0xff, 0x99 };
  ClutterColor actor_color2 = { 0x10, 0x40, 0x90, 0xff };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add our custom container to the stage: */
  ClutterActor *box = example_box_new ();

  /* Set the size to the preferred size of the container: */
  clutter_actor_set_size (box, -1, -1);
  clutter_actor_set_position (box, 20, 20);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), box);
  clutter_actor_show (box);

  /* Add some actors to our container: */
  ClutterActor *actor = clutter_rectangle_new_with_color (&actor_color);
  clutter_actor_set_size (actor, 75, 75);
  clutter_container_add_actor (CLUTTER_CONTAINER (box), actor);
  clutter_actor_show (actor);

  ClutterActor *actor2 = clutter_rectangle_new_with_color (&actor_color2);
  clutter_actor_set_size (actor2, 75, 75);
  clutter_container_add_actor (CLUTTER_CONTAINER (box), actor2);
  clutter_actor_show (actor2);

  /* Show the stage: */
  clutter_actor_show (stage);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

File: `examplebox.c`

```
#include "examplebox.h"

#include <clutter/clutter.h>
#include <cogl/cogl.h>

#include <string.h>

/**
 * SECTION:example-box
 * @short_description: Simple example of a container actor.
 *
 * #ExampleBox imposes a specific layout on its children,
 * unlike #ClutterGroup which is a free-form container.
 *
 * Specifically, ExampleBox lays out its children along an imaginary
 * horizontal line.
 */

static void clutter_container_iface_init (ClutterContainerIface *iface);

G_DEFINE_TYPE_WITH_CODE (ExampleBox,
                         example_box,
                         CLUTTER_TYPE_ACTOR,
                         G_IMPLEMENT_INTERFACE (CLUTTER_TYPE_CONTAINER,
                                                clutter_container_iface_init));

/* An implementation for the ClutterContainer::add() vfunc: */
static void
example_box_add (ClutterContainer *container,
                 ClutterActor     *actor)
{
  example_box_pack (EXAMPLE_BOX (container), actor);
}

/* An implementation for the ClutterContainer::remove() vfunc: */
static void
example_box_remove (ClutterContainer *container,
                    ClutterActor     *actor)
{
  ExampleBox *box = EXAMPLE_BOX (container);
  GList *l;

  g_object_ref (actor);

  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      if (child == actor)
        {
          clutter_actor_unparent (child);

          box->children = g_list_remove_link (box->children, l);
```

```
          g_list_free (l);

          g_signal_emit_by_name (container, "actor-removed", actor);

          /* queue a relayout of the container */
  clutter_actor_queue_relayout (CLUTTER_ACTOR (box));

          break;
        }
    }

  g_object_unref (actor);
}

/* An implementation for the ClutterContainer::foreach() vfunc: */
static void
example_box_foreach (ClutterContainer *container,
                     ClutterCallback   callback,
                     gpointer          user_data)
{
  ExampleBox *box = EXAMPLE_BOX (container);
  GList *l;

  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      (* callback) (child, user_data);
    }
}

static void
clutter_container_iface_init (ClutterContainerIface *iface)
{
  /* Provide implementations for ClutterContainer vfuncs: */
  iface->add = example_box_add;
  iface->remove = example_box_remove;
  iface->foreach = example_box_foreach;
}

/* An implementation for the ClutterActor::show_all() vfunc,
   showing all the child actors: */
static void
example_box_show_all (ClutterActor *actor)
{
  ExampleBox *box = EXAMPLE_BOX (actor);
  GList *l;

  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      clutter_actor_show (child);
```

```
    }

  clutter_actor_show (actor);
}

/* An implementation for the ClutterActor::hide_all() vfunc,
   hiding all the child actors: */
static void
example_box_hide_all (ClutterActor *actor)
{
  ExampleBox *box = EXAMPLE_BOX (actor);
  GList *l;

  clutter_actor_hide (actor);

  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      clutter_actor_hide (child);
    }
}

/* An implementation for the ClutterActor::paint() vfunc,
   painting all the child actors: */
static void
example_box_paint (ClutterActor *actor)
{
  ExampleBox *box = EXAMPLE_BOX (actor);
  GList *l;

  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      if (CLUTTER_ACTOR_IS_MAPPED (child))
        clutter_actor_paint (child);
    }
}

/* An implementation for the ClutterActor::pick() vfunc,
   picking all the child actors: */
static void
example_box_pick (ClutterActor *actor,
                  const ClutterColor *color)
{
  ExampleBox *box = EXAMPLE_BOX (actor);
  GList *l = NULL;

  /* Chain up so we get a bounding box painted (if we are reactive) */
  CLUTTER_ACTOR_CLASS (example_box_parent_class)->pick (actor, color);
```

```
  /* TODO: Do something with the color?
   * In clutter 0.8 we used it to call clutter_actor_pick() on the children,
   * but now we call clutter_actor_paint() instead.
   */
  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      if (CLUTTER_ACTOR_IS_MAPPED (child))
        clutter_actor_paint (child);
    }
}

/* An implementation for the ClutterActor::get_preferred_width() vfunc: */
static void
example_box_get_preferred_width (ClutterActor *actor,
                                 float for_height,
                                 float *min_width_p,
                                 float *natural_width_p)
{
  ExampleBox *box = EXAMPLE_BOX (actor);
  GList *l;
  float min_width = 0, natural_width = 0;

  /* For this container, the preferred width is the sum of the widths
   * of the children. The preferred width depends on the height provided
   * by for_height.
   */

  /* Calculate the preferred width for this container,
   * based on the preferred width requested by the children: */
  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      if (CLUTTER_ACTOR_IS_VISIBLE (child))
        {
          float child_min_width = 0;
          float child_natural_width = 0;
          clutter_actor_get_preferred_width (child, for_height, &child_min_width, &child_na

          min_width += child_min_width;
          natural_width += child_natural_width;
        }
    }

  if (min_width_p)
    *min_width_p = min_width;

  if (natural_width_p)
    *natural_width_p = natural_width;
}
```

```
/* An implementation for the ClutterActor::get_preferred_height() vfunc: */
static void
example_box_get_preferred_height (ClutterActor *actor,
                                  float for_width,
                                  float *min_height_p,
                                  float *natural_height_p)
{
  ExampleBox *box = EXAMPLE_BOX (actor);
  GList *l;
  float min_height = 0, natural_height = 0;

  /* For this container, the preferred height is the maximum height
   * of the children. The preferred height is independent of the given width.
   */

  /* Calculate the preferred height for this container,
   * based on the preferred height requested by the children: */
  for (l = box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      if (CLUTTER_ACTOR_IS_VISIBLE (child))
        {
          float child_min_height = 0;
          float child_natural_height = 0;
          clutter_actor_get_preferred_height (child, -1, &child_min_height, &child_natural_

          min_height = MAX (min_height, child_min_height);
          natural_height = MAX (natural_height, child_natural_height);
        }
    }

  if (min_height_p)
    *min_height_p = min_height;

  if (natural_height_p)
    *natural_height_p = natural_height;
}

/* An implementation for the ClutterActor::allocate() vfunc: */
static void
example_box_allocate (ClutterActor          *actor,
                      const ClutterActorBox *box,
                  ClutterAllocationFlags absolute_origin_changed)
{
  ExampleBox *ebox = EXAMPLE_BOX (actor);

  /* Look at each child actor: */
  float child_x = 0;
  GList *l = NULL;
  for (l = ebox->children; l; l = l->next)
    {
      ClutterActor *child = l->data;
```

```
      /* Discover what size the child wants: */
      float child_width = 0;
      float child_height = 0;
      clutter_actor_get_preferred_size (child, NULL, NULL, &child_width, &child_height);

      /* Calculate the position and size that the child may actually have: */

      /* Position the child just after the previous child, horizontally: */
      ClutterActorBox child_box = { 0, };
      child_box.x1 = child_x;
      child_box.x2 = child_x + child_width;
      child_x = child_box.x2;

      /* Position the child at the top of the container: */
      child_box.y1 = 0;
      child_box.y2 = child_box.y1 + child_height;

      /* Tell the child what position and size it may actually have: */
      clutter_actor_allocate (child, &child_box, absolute_origin_changed);
    }

  CLUTTER_ACTOR_CLASS (example_box_parent_class)->allocate (actor, box, absolute_origin_cha
}

static void
example_box_dispose (GObject *gobject)
{
  /* Destroy each child actor when this container is destroyed: */
  ExampleBox *box = EXAMPLE_BOX (gobject);
  GList *l;

  for (l =  box->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      clutter_actor_destroy (child);
    }

  g_list_free (box->children);
  box->children = NULL;

  G_OBJECT_CLASS (example_box_parent_class)->dispose (gobject);
}

static void
example_box_class_init (ExampleBoxClass *klass)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
  ClutterActorClass *actor_class = CLUTTER_ACTOR_CLASS (klass);

  gobject_class->dispose = example_box_dispose;
```

```
  /* Provide implementations for ClutterActor vfuncs: */
  actor_class->show_all = example_box_show_all;
  actor_class->hide_all = example_box_hide_all;
  actor_class->paint = example_box_paint;
  actor_class->pick = example_box_pick;
  actor_class->get_preferred_width = example_box_get_preferred_width;
  actor_class->get_preferred_height = example_box_get_preferred_height;
  actor_class->allocate = example_box_allocate;
}

static void
example_box_init (ExampleBox *box)
{
  /* The required width depends on a given height in this container */
  g_object_set (G_OBJECT (box),
                "request-mode", CLUTTER_REQUEST_WIDTH_FOR_HEIGHT,
                NULL);
}

/*
 * Public API
 */

/**
 * example_box_pack:
 * @box: a #ExampleBox
 * @actor: a #ClutterActor to pack into the box
 *
 * Packs @actor into @box.
 */
void
example_box_pack (ExampleBox           *box,
                  ClutterActor         *actor)
{

  g_return_if_fail (EXAMPLE_IS_BOX (box));
  g_return_if_fail (CLUTTER_IS_ACTOR (actor));

  box->children = g_list_prepend (box->children, actor);
  clutter_actor_set_parent (actor, CLUTTER_ACTOR (box));

  /* queue a relayout of the container */
  clutter_actor_queue_relayout (CLUTTER_ACTOR (box));
}


/**
 * example_box_remove_all:
 * @box: a #ExampleBox
 *
 * Removes all child actors from the #ExampleBox
 */
void
```

```
example_box_remove_all (ExampleBox *box)
{
  GList *children;

  g_return_if_fail (EXAMPLE_IS_BOX (box));

  children = box->children;
  while (children)
    {
      ClutterActor *child = children->data;
      children = children->next;

      clutter_container_remove_actor (CLUTTER_CONTAINER (box), child);
    }
}


/**
 * example_box_new:
 *
 * Creates a new box.
 *
 * Return value: the newly created #ExampleBox
 */
ClutterActor *
example_box_new (void)
{
  return g_object_new (EXAMPLE_TYPE_BOX, NULL);
}
```

Appendix B. Implementing Scrolling in a Window-like Actor

B.1. The Technique

There is not yet a standard container in Clutter to show and scroll through a small part of a set of widgets, like the `GtkScrolledWindow` widget in the GTK+ toolkit, so you may need to implement this functionality in your application.

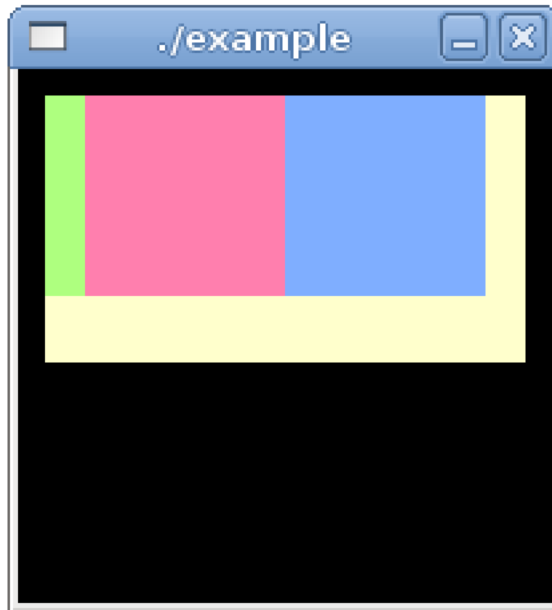
The Tidy project contains some suggested implementations for such actors, but here is a simpler example of the general technique. It creates the impression of scrolling by clipping a container so that it only shows a small area of the screen, while moving the child widgets in that container.

B.2. Example

This example places three rectangles in a custom container which scrolls its child widgets to the left when the user clicks on the stage.

Real-world applications will of course want to implement more specific behaviour, depending on their needs. For instance, adding scrollbars that show accurate scroll positions, scrolling smoothly with animation, efficiently drawing only objects that should be visible when dealing with large numbers of rows.

Figure B-1. Scrolling Container



Source Code (../../../examples/scrolling)

File: `scrollingcontainer.h`

```
#ifndef __EXAMPLE_SCROLLING_CONTAINER_H__
#define __EXAMPLE_SCROLLING_CONTAINER_H__

#include <clutter/clutter.h>

G_BEGIN_DECLS

#define EXAMPLE_TYPE_SCROLLING_CONTAINER                (example_scrolling_container_get_ty
#define EXAMPLE_SCROLLING_CONTAINER(obj)                (G_TYPE_CHECK_INSTANCE_CAST ((obj),
#define EXAMPLE_IS_SCROLLING_CONTAINER(obj)             (G_TYPE_CHECK_INSTANCE_TYPE ((obj),
#define EXAMPLE_SCROLLING_CONTAINER_CLASS(klass)        (G_TYPE_CHECK_CLASS_CAST ((klass),
#define EXAMPLE_IS_SCROLLING_CONTAINER_CLASS(klass)     (G_TYPE_CHECK_CLASS_TYPE ((klass),
#define EXAMPLE_SCROLLING_CONTAINER_GET_CLASS(obj)      (G_TYPE_INSTANCE_GET_CLASS ((obj),


typedef struct _ExampleScrollingContainerChild         ExampleScrollingContainerChild;
typedef struct _ExampleScrollingContainer              ExampleScrollingContainer;
typedef struct _ExampleScrollingContainerClass         ExampleScrollingContainerClass;

struct _ExampleScrollingContainer
{
```

```
  /*< private >*/
  ClutterActor parent_instance;

  /* List of ExampleScrollingContainerChild structures */
  GList *children;

  /* All the child actors are in this group: */
  ClutterActor *group;
  float offset;

  /* A rectange to show the bounds: */
  ClutterActor *rect;
};

struct _ExampleScrollingContainerClass
{
  /*< private >*/
  ClutterActorClass parent_class;
};


GType example_scrolling_container_get_type (void) G_GNUC_CONST;

ClutterActor *example_scrolling_container_new (void);
void example_scrolling_container_pack (ExampleScrollingContainer *self, ClutterActor *actor
void example_scrolling_container_remove_all (ExampleScrollingContainer *self);

void example_scrolling_container_scroll_left (ExampleScrollingContainer *self, gint distanc

G_END_DECLS

#endif /* __EXAMPLE_SCROLLING_CONTAINER_H__ */
```

File: `main.c`

```
#include <clutter/clutter.h>
#include "scrollingcontainer.h"
#include <stdlib.h>

ClutterActor *scrolling = NULL;


static gboolean
on_stage_button_press (ClutterStage *stage, ClutterEvent *event, gpointer data)
{
  printf ("Scrolling\n");

  /* Scroll the container: */
  example_scrolling_container_scroll_left (
    EXAMPLE_SCROLLING_CONTAINER (scrolling), 10);

  return TRUE; /* Stop further handling of this event. */
```

```
}

int main(int argc, char *argv[])
{
  ClutterColor stage_color = { 0x00, 0x00, 0x00, 0xff };
  ClutterColor actor_color = { 0x7f, 0xae, 0xff, 0xff };
  ClutterColor actor_color2 = { 0xff, 0x7f, 0xae, 0xff };
  ClutterColor actor_color3 = { 0xae, 0xff, 0x7f, 0xff };

  clutter_init (&argc, &argv);

  /* Get the stage and set its size and color: */
  ClutterActor *stage = clutter_stage_get_default ();
  clutter_actor_set_size (stage, 200, 200);
  clutter_stage_set_color (CLUTTER_STAGE (stage), &stage_color);

  /* Add our scrolling container to the stage: */
  scrolling = example_scrolling_container_new ();
  clutter_actor_set_size (scrolling, 180, 100);
  clutter_actor_set_position (scrolling, 10, 10);
  clutter_container_add_actor (CLUTTER_CONTAINER (stage), scrolling);
  clutter_actor_show (scrolling);

 /* Add some actors to our container: */
  ClutterActor *actor = clutter_rectangle_new_with_color (&actor_color);
  clutter_actor_set_size (actor, 75, 75);
  clutter_container_add_actor (CLUTTER_CONTAINER (scrolling), actor);
  clutter_actor_show (actor);

  ClutterActor *actor2 = clutter_rectangle_new_with_color (&actor_color2);
  clutter_actor_set_size (actor2, 75, 75);
  clutter_container_add_actor (CLUTTER_CONTAINER (scrolling), actor2);
  clutter_actor_show (actor2);

  ClutterActor *actor3 = clutter_rectangle_new_with_color (&actor_color3);
  clutter_actor_set_size (actor3, 75, 75);
  clutter_container_add_actor (CLUTTER_CONTAINER (scrolling), actor3);
  clutter_actor_show (actor3);


  /* Show the stage: */
  clutter_actor_show (stage);

  /* Connect signal handlers to handle mouse clicks on the stage: */
  g_signal_connect (stage, "button-press-event",
    G_CALLBACK (on_stage_button_press), NULL);

  /* Start the main loop, so we can respond to events: */
  clutter_main ();

  return EXIT_SUCCESS;

}
```

File: `scrollingcontainer.c`

```
#include "scrollingcontainer.h"
#include <clutter/clutter.h>

#include <string.h>
#include <stdio.h>

/**
 * SECTION:example-scrolling-container
 * @short_description: This container shows only a small area
 * of its child actors, and the child actors can be scrolled
 * left under that area.
 */


static void clutter_container_iface_init (ClutterContainerIface *iface);

G_DEFINE_TYPE_WITH_CODE (ExampleScrollingContainer,
                         example_scrolling_container,
                         CLUTTER_TYPE_ACTOR,
                         G_IMPLEMENT_INTERFACE (CLUTTER_TYPE_CONTAINER,
                                                clutter_container_iface_init));


/* An implementation for the ClutterContainer::add() vfunc: */
static void
example_scrolling_container_add (ClutterContainer *container,
                                 ClutterActor     *actor)
{
  example_scrolling_container_pack (EXAMPLE_SCROLLING_CONTAINER (container), actor);
}

/* An implementation for the ClutterContainer::remove() vfunc: */
static void
example_scrolling_container_remove (ClutterContainer *container,
                                    ClutterActor     *actor)
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (container);
  GList *l;

  g_object_ref (actor);

  for (l = self->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      if (child == actor)
        {
          clutter_container_remove_actor (CLUTTER_CONTAINER (self->group), child);

          self->children = g_list_remove_link (self->children, l);
          g_list_free (l);
```

```
          g_signal_emit_by_name (container, "actor-removed", actor);

          clutter_actor_queue_relayout (CLUTTER_ACTOR (self));

          break;
        }
    }

  g_object_unref (actor);
}

/* An implementation for the ClutterContainer::foreach() vfunc: */
static void
example_scrolling_container_foreach (ClutterContainer *container,
                                     ClutterCallback   callback,
                                     gpointer          user_data)
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (container);
  clutter_container_foreach (CLUTTER_CONTAINER (self->group), callback, user_data);
}

static void
clutter_container_iface_init (ClutterContainerIface *iface)
{
  /* Provide implementations for ClutterContainer vfuncs: */
  iface->add = example_scrolling_container_add;
  iface->remove = example_scrolling_container_remove;
  iface->foreach = example_scrolling_container_foreach;
}

/* An implementation for the ClutterActor::show_all() vfunc,
   showing all the child actors: */
static void
example_scrolling_container_show_all (ClutterActor *actor)
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (actor);
  GList *l;

  for (l = self->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      clutter_actor_show (child);
    }

  clutter_actor_show (actor);
}

/* An implementation for the ClutterActor::hide_all() vfunc,
   hiding all the child actors: */
static void
example_scrolling_container_hide_all (ClutterActor *actor)
```

```
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (actor);
  GList *l;

  clutter_actor_hide (actor);

  for (l = self->children; l; l = l->next)
    {
      ClutterActor *child = l->data;

      clutter_actor_hide (child);
    }
}

/* An implementation for the ClutterActor::paint() vfunc,
   painting all the child actors: */
static void
example_scrolling_container_paint (ClutterActor *actor)
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (actor);
  clutter_actor_paint (self->group);
}

/* An implementation for the ClutterActor::pick() vfunc,
   drawing outlines of all the child actors: */
static void
example_scrolling_container_pick (ClutterActor *actor,
                                  const ClutterColor *color)
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (actor);

  /* Chain up so we get a bounding box painted (if we are reactive) */
  CLUTTER_ACTOR_CLASS (example_scrolling_container_parent_class)->pick (actor, color);

  clutter_actor_paint (self->group);
}

/* An implementation for the ClutterActor::allocate() vfunc: */
static void
example_scrolling_container_allocate (ClutterActor          *actor,
                                      const ClutterActorBox *box,
                                      ClutterAllocationFlags absolute_origin_changed)
{
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (actor);

  /* Make sure that the children adapt their positions: */
  float width = box->x2 - box->x1;
  float height = box->y2 - box->y1;

  if (width < 0)
    width = 0;

  if (height < 0)
```

```
    height = 0;

  /* Arrange the group: */
  ClutterActorBox child_box = { 0, };
  child_box.x1 = 0;
  child_box.x2 = width;

  /* Position the child at the top of the container: */
  child_box.y1 = 0;
  child_box.y2 = height;

  clutter_actor_allocate (self->group, &child_box, absolute_origin_changed);

  /* Make sure that the group only shows the specified area, by clipping: */
  clutter_actor_set_clip (self->group, 0, 0, width, height);

  /* Show a rectangle border to show the area: */
  clutter_actor_allocate (self->rect, &child_box, absolute_origin_changed);
  clutter_actor_lower (self->rect, NULL);

  /* Look at each child actor: */
  float child_x = -(self->offset);
  GList *l = NULL;
  for (l = self->children; l; l = l->next)
    {
      ClutterActor *child = l->data;
      float width = 0;
      float height = 0;
      clutter_actor_get_preferred_size (child, NULL, NULL, &width, &height);

      child_box.x1 = child_x;
      child_box.y1 = 0;
      child_box.x2 = child_box.x1 + width;
      child_box.y2 = child_box.y1 + height;

      clutter_actor_allocate (child, &child_box, absolute_origin_changed);

      child_x += width;
    }

  CLUTTER_ACTOR_CLASS (example_scrolling_container_parent_class)->allocate (actor, box, abs
}


static void
example_scrolling_container_dispose (GObject *gobject)
{
  /* Destroy each child actor when this container is destroyed: */
  ExampleScrollingContainer *self = EXAMPLE_SCROLLING_CONTAINER (gobject);
  GList *l;

  for (l =  self->children; l; l = l->next)
    {
```

```
        ClutterActor *child = l->data;

        clutter_actor_destroy (child);
      }

  g_list_free (self->children);
  self->children = NULL;

  if (self->group)
    {
      clutter_actor_destroy (self->group);
      self->group = NULL;
  }

  G_OBJECT_CLASS (example_scrolling_container_parent_class)->dispose (gobject);
}

static void
example_scrolling_container_class_init (ExampleScrollingContainerClass *klass)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
  ClutterActorClass *actor_class = CLUTTER_ACTOR_CLASS (klass);

  gobject_class->dispose = example_scrolling_container_dispose;

  /* Provide implementations for ClutterActor vfuncs: */
  actor_class->show_all = example_scrolling_container_show_all;
  actor_class->hide_all = example_scrolling_container_hide_all;
  actor_class->paint = example_scrolling_container_paint;
  actor_class->pick = example_scrolling_container_pick;
  actor_class->allocate = example_scrolling_container_allocate;
}

static void
example_scrolling_container_init (ExampleScrollingContainer *self)
{
  self->group = clutter_group_new ();
  clutter_actor_show (self->group);
  self->offset = 0;

  /* A rectange to show the bounds: */
  ClutterColor actor_color = { 0xff, 0xff, 0xcc, 0xff };
  self->rect = clutter_rectangle_new_with_color (&actor_color);
  clutter_container_add_actor (CLUTTER_CONTAINER (self->group), self->rect);
  clutter_actor_show (self->rect);
}

/*
 * Public API
 */

/**
 * example_scrolling_container_pack:
```

```
 * @self: a #ExampleScrollingContainer
 * @actor: a #ClutterActor to pack into the self
 *
 * Packs @actor into @self.
 */
void
example_scrolling_container_pack (ExampleScrollingContainer           *self,
                  ClutterActor         *actor)
{

  g_return_if_fail (EXAMPLE_IS_SCROLLING_CONTAINER (self));
  g_return_if_fail (CLUTTER_IS_ACTOR (actor));

  self->children = g_list_prepend (self->children, actor);
  clutter_container_add_actor (CLUTTER_CONTAINER (self->group), actor);

  clutter_actor_queue_relayout (CLUTTER_ACTOR (self));
}


/**
 * example_scrolling_container_remove_all:
 * @self: a #ExampleScrollingContainer
 *
 * Removes all child actors from the #ExampleScrollingContainer
 */
void
example_scrolling_container_remove_all (ExampleScrollingContainer *self)
{
  GList *children;

  g_return_if_fail (EXAMPLE_IS_SCROLLING_CONTAINER (self));

  children = self->children;
  while (children)
    {
      ClutterActor *child = children->data;
      children = children->next;

      clutter_container_remove_actor (CLUTTER_CONTAINER (self), child);
    }
}


/**
 * example_scrolling_container_new:
 *
 * Creates a new self.
 *
 * Return value: the newly created #ExampleScrollingContainer
 */
ClutterActor *
example_scrolling_container_new (void)
```

```
{
  return g_object_new (EXAMPLE_TYPE_SCROLLING_CONTAINER, NULL);
}

/**
 * example_scrolling_container_scroll_left:
 * @self: a #ExampleScrollingContainer
 * @distance: The number of pixels by which to scroll left.
 *
 * Scroll all the child widgets left,
 * resulting in some parts being hidden,
 * and some parts becoming visible.
 */
void example_scrolling_container_scroll_left (ExampleScrollingContainer *self, gint distanc
{
  g_return_if_fail (EXAMPLE_IS_SCROLLING_CONTAINER (self));

  self->offset += distance;

  clutter_actor_queue_relayout (CLUTTER_ACTOR (self));
}
```

Chapter 11. Contributing

If you find errors in this documentation or if you would like to contribute additional material, you are encouraged to write an email to murrayc@openismus.com. Thanks.

The DocBook and C source code for this documentation is in the clutter-tutorial (http://git.gnome.org/cgit/clutter-tutorial/) module in GNOME's git repository.