Programming with Cluttermm

Murray Cumming

Daniel Elstner

Programming with Cluttermm

by Murray Cumming and Daniel Elstner

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).

Table of Contents

List of Figures

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 Cluttermm 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 Cluttermm

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

We would very much like to hear of any problems you have learning Cluttermm 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.

1.3. Cluttermm

Cluttermm is a language binding for C++ on top of Clutter. It has the same functionality and concepts as plain Clutter, but provides C++ programmers with an interface that uses language features and common concepts of C++, such as static type safety, class inheritance and (optionally) exception handling.

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

2.2. Installing from Source

After you've installed all of the dependencies, download the Cluttermm source code, unpack it, and change to the newly created directory. Cluttermm 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, Cluttermm will be installed under the `/usr/local` directory.

If you want to help develop Cluttermm or experiment with new features, you can also install Cluttermm from SVN. Details are available at the gtkmm website (http://www.gtkmm.org/).

2.2.1. Dependencies

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

- GTK+
- libgl (Mesa)
- clutter
- clutter-gtk

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 Cluttermm 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 --cflags cluttermm-1.0
pkg-config --libs cluttermm-1.0
```

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], [cluttermm-1.0 >= 1.0.0])
```

You should then use the generated `$(EXAMPLE_CFLAGS)` and `$(EXAMPLE_LIBS)` variables 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 additional Cluttermm libraries, such as `clutter-gtkmm-1.0`.

Chapter 4. The Stage

4.1. Stage Basics

Each Cluttermm application contains at least one `Clutter::Stage`. 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 Cluttermm. You may then get the application's stage with `Clutter::Stage::get_default()`. This method always returns the same instance, with its own window. You could instead use a `Clutter::Gtk::Embed` widget inside a more complicated GTK+ window -- see the Stage Widget section.

`Clutter::Stage` is derived from the `Clutter::Actor` class so many of that class' methods are useful for the stage. For instance, call `Clutter::Actor::show()` to make the stage visible.

`Clutter::Stage` also implements the `Clutter::Container` interface, allowing it to contain child actors via calls to `Clutter::Container::add_actor()`.

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

`Clutter::Stage` class reference
(http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Stage.html)

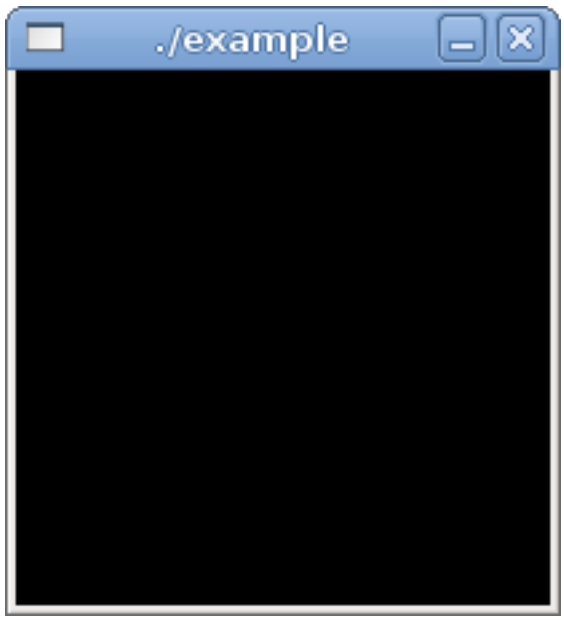
4.1.1. Example

The following example shows a `Clutter::Stage` 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.

```
c++ -Wall -g `pkg-config --cflags --libs cluttermm-1.0` -o example main.cc
```

Figure 4-1. Stage



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

File: `main.cc`

```
#include <cluttermm.h>
#include <iostream>

namespace
{

bool on_stage_button_press(Clutter::ButtonEvent* event)
{
  float x = 0;
  float y = 0;
  // TODO: Wrap properly
  clutter_event_get_coords(reinterpret_cast<Clutter::Event*>(event), &x, &y);

  std::cout << "Stage clicked at (" << x << ", " << y << ")\n";

  return true; // stop further handling of this event
}

} // anonymous namespace

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

```
{
  try
  {
    Clutter::init(&argc, &argv);

    // Get the stage and set its size and color:
    Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
    stage->set_size(200, 200);
    stage->set_color(Clutter::Color(0, 0, 0)); // black

    stage->show();

    // Connect a signal handler to handle mouse clicks:
    stage->signal_button_press_event().connect(&on_stage_button_press);

    // Start the main loop, so we can respond to events:
    Clutter::main();
  }
  catch (const Glib::Error& error)
  {
    std::cerr << "Exception: " << error.what() << std::endl;
    return 1;
  }

  return 0;
}
```

4.2. Stage Widget

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

When using the `Clutter::Gtk::Embed` widget you should use `Clutter::Gtk::init()` instead of `Clutter::init()` and `Gtk::Main` to initialize Clutter and gtkmm. And you should use the regular `Gtk::Main::run()` method 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 `Clutter::Gtk::Embed` widget.

`Clutter::Embed` class reference
(http://library.gnome.org/devel/clutter-gtkmm/unstable/classClutter_1_1Gtk_1_1Embed.html)

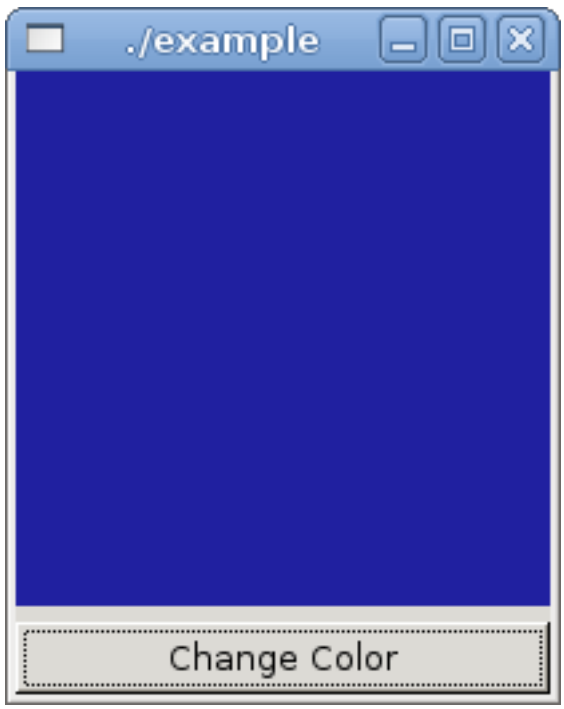
4.2.1. Example

The following example shows a `Clutter::Gtk::Embed` GTK+ widget and changes the stage color when a button is clicked.

Note that this example requires the `clutter-gtkmm-1.0` library as well as the standard `cluttermm-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.

```
c++ -Wall -g `pkg-config --cflags --libs clutter-gtkmm-1.0` -o example main.cc
```

Figure 4-2. Stage Widget



Source Code (../../../examples/stage_widget)

File: main.cc

```
#include <gtkmm.h>
#include <clutter-gtkmm.h>
#include <iostream>

namespace
{

class StageWindow : public Gtk::Window
{
public:
  StageWindow();
  virtual ~StageWindow();

private:
  Clutter::Gtk::Embed* embed_;
  Glib::RefPtr<Clutter::Stage> stage_;
  bool colored_;

  void on_button_clicked();
  static bool on_stage_button_press(Clutter::ButtonEvent* event);
};

StageWindow::StageWindow()
:
  embed_(0),
  colored_(false)
{
  Gtk::Box *const box = new Gtk::VBox(false, 6);
  add(*Gtk::manage(box));

  Gtk::Button *const button = new Gtk::Button("Change color");
  box->pack_end(*Gtk::manage(button), Gtk::PACK_SHRINK);

  embed_ = new Clutter::Gtk::Embed();
  box->pack_start(*Gtk::manage(embed_), Gtk::PACK_EXPAND_WIDGET);
  embed_->set_size_request(200, 200);

  button->signal_clicked().connect(sigc::mem_fun(*this, &StageWindow::on_button_clicked));

  stage_ = embed_->get_stage();
  stage_->reference();
  stage_->set_color(Clutter::Color(0, 0, 0)); // black
  stage_->signal_button_press_event().connect(&StageWindow::on_stage_button_press);

  show_all();
  stage_->show();
}

StageWindow::~StageWindow()
{}
```

```
void StageWindow::on_button_clicked()
{
  colored_ = !colored_;
  stage_->set_color((colored_) ? Clutter::Color(32, 32, 160)
    : Clutter::Color(0, 0, 0));
}

bool StageWindow::on_stage_button_press(Clutter::ButtonEvent* event)
{
  float x = 0;
  float y = 0;
  // TODO: Wrap properly
  clutter_event_get_coords(reinterpret_cast<Clutter::Event*>(event), &x, &y);

  std::cout << "Stage clicked at (" << x << ", " << y << ")\n";

  return true; // stop further handling of this event
}

} // anonymous namespace


int main(int argc, char** argv)
{
  try
  {
    Clutter::Gtk::init(&argc, &argv);
    Gtk::Main kit (&argc, &argv);

    StageWindow window;
    Gtk::Main::run(window);
  }
  catch (const Glib::Error& error)
  {
    std::cerr << "Exception: " << error.what() << std::endl;
    return 1;
  }

  return 0;
}
```

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:

- Clutter::Stage (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Stage.html): The stage itself, mentioned already
- Clutter::Rectangle (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Rectangle.html): A rectangle.
- Clutter::Text (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Text.html): Displays and edits text.
- Clutter::Texture (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Texture.html): An image.

Each actor should be added to the stage with `Clutter::Container::add_actor()` and its positions should then be specified. All actors derive from `Clutter::Actor` 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 all objects of the `Glib::Object` class, Cluttermm actors are reference counted. The reference count starts at one when they are first instantiated with a method such as `Clutter::Rectangle::create()`. This reference is then managed by the `Glib::RefPtr<Clutter::Rectangle>` smart pointer returned by the method. If the smart pointer goes out of scope, the reference count is decremented again, and the actor will be destroyed if the reference count reached zero. However, if the actor is added to a container such as the stage, the container will obtain another reference to the actor, which is not released until the actor has been removed again from the container. This means you do not need to keep around the

`Glib::RefPtr<Clutter::Rectangle>` after the actor was added to a container, and neither do you have to manually reference or unreference any Cluttermm object.

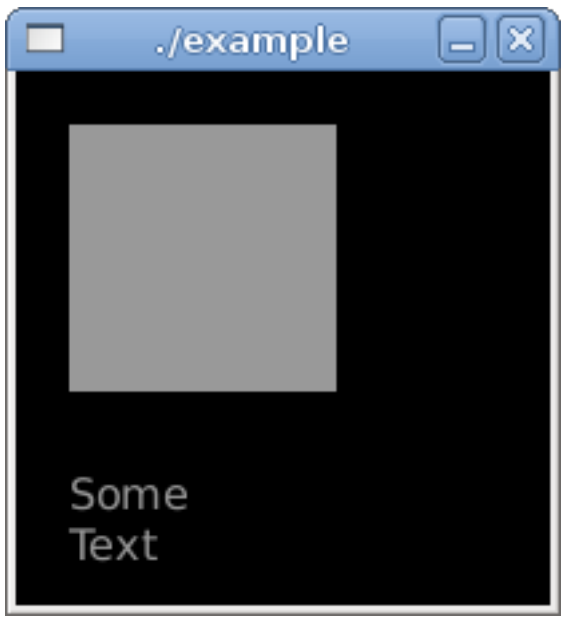
Actors may also be transformed by scaling or rotation, and may be made partly transparent.

`Clutter::Actor` class reference (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Actor.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.cc`

```
#include <cluttermm.h>

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);
```

```
  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0, 0, 0, 255)); // black

  Clutter::Color actor_color (0xff, 0xff, 0xff, 0x99);

  // Add a rectangle to the stage:
  const Glib::RefPtr<Clutter::Rectangle> rect = Clutter::Rectangle::create(actor_color);
  rect->set_size(100, 100);
  rect->set_position(20, 20);
  stage->add_actor(rect);
  rect->show();

  // Add a label to the stage:
  const Glib::RefPtr<Clutter::Actor> label =
    Clutter::Text::create("Sans 12", "Some Text", actor_color);
  label->set_size(500, 500);
  label->set_position(20, 150);
  stage->add_actor(label);
  label->show();

  // Show the stage:
  stage->show();

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

  return 0;
}
```

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

Cluttermm does not have a translation method 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 methods, `Clutter::Actor::move_by()` does change the result of methods 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.cc`

```
#include <cluttermm.h>

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0, 0, 0, 0xFF)); // black

  const Clutter::Color actor_color (0xff, 0xff, 0xff, 0x99);

  // Add a rectangle to the stage:
  const Glib::RefPtr<Clutter::Rectangle> rect = Clutter::Rectangle::create(actor_color);
  rect->set_size(100, 100);
  rect->set_position(20, 20);
  stage->add_actor(rect);
  rect->show();

  // Rotate it 20 degrees away from us around the x axis
```

```
  // (around its top edge):
  rect->set_rotation(Clutter::X_AXIS, -20, 0, 0, 0);

  // Add a label to the stage:
  const Glib::RefPtr<Clutter::Text> label =
      Clutter::Text::create("Sans 12", "Some Text", actor_color);
  label->set_size(500, 500);
  label->set_position(20, 150);
  stage->add_actor(label);
  label->show();

  // Scale it 300% along the x axis:
  label->set_scale(3.0, 1.0);

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

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

  // Show the stage:
  stage->show();

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

  return 0;
}
```

5.3. Containers

Some clutter actors implement the `Clutter::Container` 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_actor()` method.

The main `Clutter::Stage` is itself a container, allowing it to contain all the child actors. The only other container in core Cluttermm is `Clutter::Group`, which can contain child actors, with positions relative to the parent `Clutter::Group`. 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.

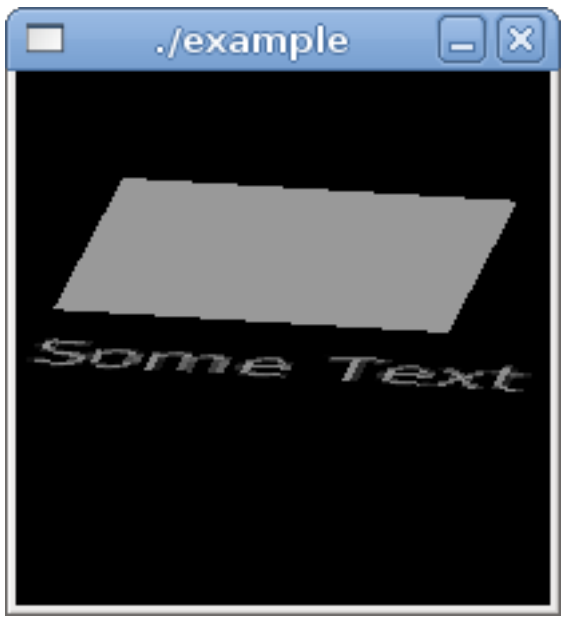
`Clutter::Container` class reference
(http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Container.html)

`Clutter::Group` class reference
(http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Group.html)

5.3.1. Example

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

Figure 5-3. Group



Source code (../../../examples/actor_group)

File: `main.cc`

```
#include <cluttermm.h>

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
```

```
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0, 0, 0, 0xFF)); // black

  // Add a group to the stage:
  const Glib::RefPtr<Clutter::Group> group = Clutter::Group::create();
  group->set_position(40, 40);
  stage->add_actor(group);
  group->show();

  const Clutter::Color actor_color (0xFF, 0xFF, 0xFF, 0x99);

  // Add a rectangle to the group:
  const Glib::RefPtr<Clutter::Rectangle>
    rect = Clutter::Rectangle::create(actor_color);
  rect->set_size(50, 50);
  rect->set_position(0, 0);
  group->add_actor(rect);
  rect->show();

  // Add a label to the group:
  const Glib::RefPtr<Clutter::Text>
    label = Clutter::Text::create("Sans 9", "Some Text", actor_color);
  label->set_position(0, 60);
  group->add_actor (label);
  label->show();

  // Scale the group 120% along the x axis:
  group->set_scale(3.00, 1.0);

  // Rotate it around the z axis:
  group->set_rotation(Clutter::Z_AXIS, 10, 0, 0, 0);

  // Show the stage:
  stage->show();

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

  return 0;
}
```

5.4. Events

The base `Clutter::Actor` has several signals that are emitted when the user interacts with the actor:

- `signal_button_press_event()`: Emitted when the user presses the mouse over the actor.
- `signal_button_release_event()`: Emitted when the user releases the mouse over the actor.

- `signal_motion_event()`: Emitted when the user moves the mouse over the actor.
- `signal_enter_event()`: Emitted when the user moves the mouse in to the actor's area.
- `signal_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:

```
rect->signal_button_press_event().connect(&on_rect_button_press);
```

Alternatively, you might just handle signals from the parent `Clutter::Stage` 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 (`signal_motion_event()`, `signal_enter_event()` and `signal_leave_event()`), you may call the global `Clutter::set_motion_events_enabled()` function with `false` to further optimize performance.

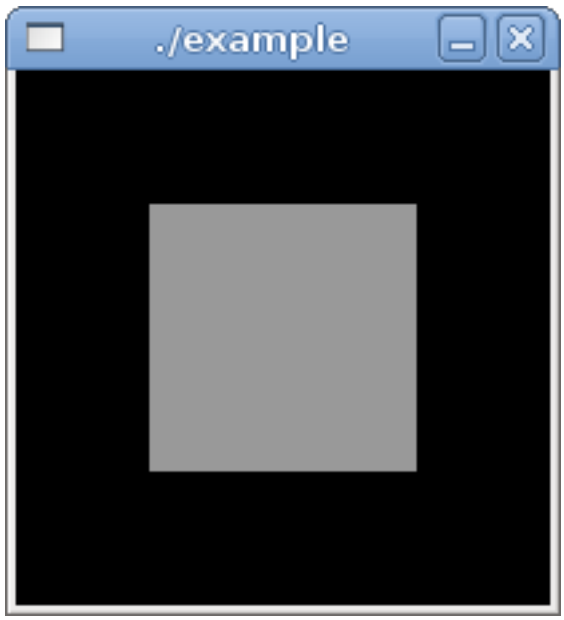
Your event signal handler should return `true` when it has fully handled the event, or `false` if you want the event to be sent also to the next actor in the event chain. Cluttermm first allows the stage to handle each event via the `signal_captured_event()` signal. But if the stage does not handle the 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 `signal_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 `signal_button_press_event()` or `signal_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)

Chapter 6. Timelines

6.1. Using Timelines

A `Clutter::Timeline` 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 effect or behaviour, as you will see in the following chapters.

The timeline object emits its `signal_new_frame()` signal for each frame that should be drawn, for as many frames per second as specified. 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::create()` constructor function takes a number of frames, and a number of frames per second, so the entire timeline will have a duration of *`n_frames`* / *`fps`*. You might therefore choose the number of frames based on a desired duration, by dividing the duration by the desired frames per second.

You may also use `Clutter::Timeline::set_loop()` to cause the timeline to repeat forever, or until you call `Clutter::Timeline::stop()`. The timeline does not start until you call `Clutter::Timeline::start()`.

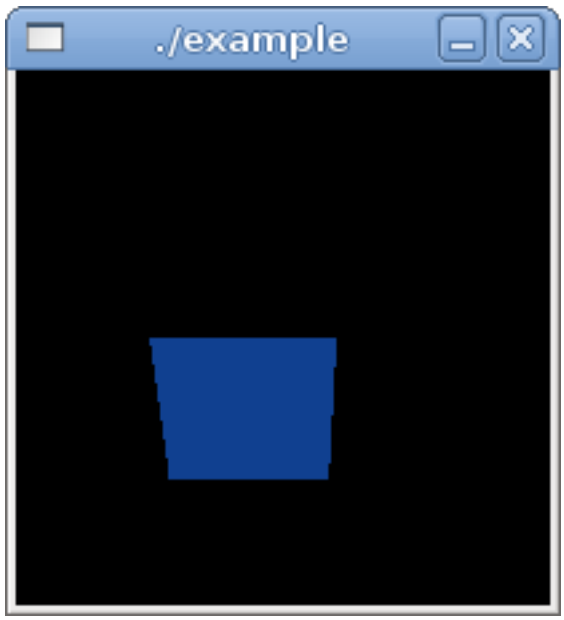
When a timeline has completed, it emits the `signal_completed()` signal.

`Clutter::Timeline` class reference (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Timeline.html)

6.2. 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.cc`

```
#include <cluttermm.h>

namespace
{

static Glib::RefPtr<Clutter::Rectangle> rect;

static int rotation_angle = 0;
static int color_change_count = 0;

static void on_timeline_new_frame(int /* frame_num */,
  Glib::RefPtr<Clutter::Timeline> /* timeline */)
{
  if(++rotation_angle >= 360)
    rotation_angle = 0;

  // Rotate the rectangle clockwise around the z axis, around
  // it's top-left corner:
  rect->set_rotation(Clutter::X_AXIS, rotation_angle, 0, 0, 0);

  // Change the color
```

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

  if(color_change_count == 0)
    rect->set_color(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  else if(color_change_count == 50)
    rect->set_color(Clutter::Color(0x10, 0x40, 0x90, 0xFF));
}

} // anonymous namespace

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF));

  // Add a rectangle to the stage:
  rect = Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  rect->set_size(70, 70);
  rect->set_position(50, 100);
  stage->add_actor(rect);
  rect->show();

  // Show the stage:
  stage->show();

  const Glib::RefPtr<Clutter::Timeline>
    timeline = Clutter::Timeline::create(5000 /* milliseconds */);
  timeline->signal_new_frame()
    .connect(sigc::bind(&on_timeline_new_frame, timeline));
  timeline->set_loop(true);
  timeline->start();

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

  return 0;
}
```

6.3. Grouping TimeLines in a Score

A `Clutter::Score` 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()` without the *`parent`* argument. All such timelines will be started when you call `Clutter::Score::start()`.

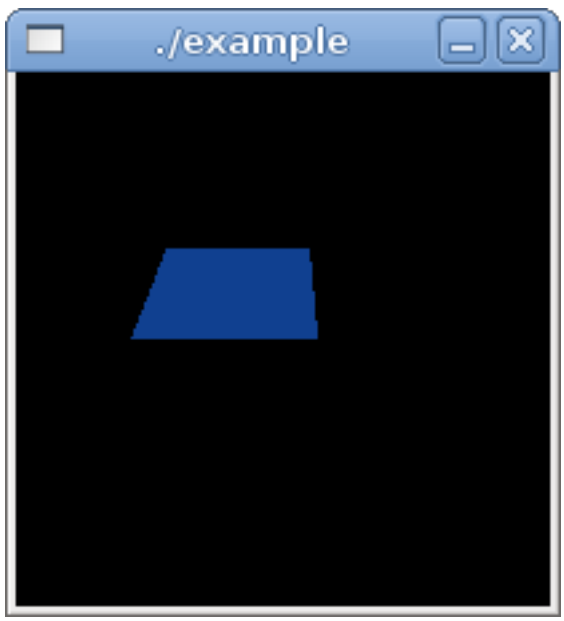
To add a timeline that should be started when a previously added timeline stops, call `Clutter::Score::append()` with the first timeline for the *`parent`* argument.

`Clutter::Score` class reference (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Score.html)

6.4. 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.cc`

```
#include <cluttermm.h>
```

```
namespace
{

static Glib::RefPtr<Clutter::Rectangle> rect;

static int rotation_angle = 0;
static int color_change_count = 0;

/*
 * Rotate the rectangle and alternate its color.
 */
static void on_timeline_rotation_new_frame(int /* frame_num */,
  Glib::RefPtr<Clutter::Timeline> /* timeline */)
{
  if(++rotation_angle >= 360)
    rotation_angle = 0;

  // Rotate the rectangle clockwise around the z axis, around
  // it's top-left corner:
  rect->set_rotation(Clutter::X_AXIS, rotation_angle, 0, 0, 0);

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

  if(color_change_count == 0)
    rect->set_color(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  else if(color_change_count == 50)
    rect->set_color(Clutter::Color(0x10, 0x40, 0x90, 0xFF));
}

/*
 * Move the rectangle.
 */
static void on_timeline_move_new_frame(int /* frame_num */,
  Glib::RefPtr<Clutter::Timeline> /* timeline */)
{
  int x_position = rect->get_x();

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

  rect->set_x(x_position);
}

} // anonymous namespace

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);
```

```
  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF));

  // Add a rectangle to the stage:
  rect = Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  rect->set_size(70, 70);
  rect->set_position(50, 100);
  stage->add_actor(rect);
  rect->show();

  // Show the stage:
  stage->show();

  // Create a score and add two timelines to it,
  // so the second timeline starts when the first one stops:
  const Glib::RefPtr<Clutter::Score> score = Clutter::Score::create();
  score->set_loop(true);

  const Glib::RefPtr<Clutter::Timeline>
    timeline_rotation = Clutter::Timeline::create(5000 /* milliseconds */);
  timeline_rotation->signal_new_frame()
    .connect(sigc::bind(&on_timeline_rotation_new_frame, timeline_rotation));
  score->append(timeline_rotation);

  const Glib::RefPtr<Clutter::Timeline>
    timeline_move = Clutter::Timeline::create(5000 /* milliseconds */);
  timeline_move->signal_new_frame()
    .connect(sigc::bind(&on_timeline_move_new_frame, timeline_move));
  score->append(timeline_rotation, timeline_move);

  score->start();

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

  return 0;
}
```

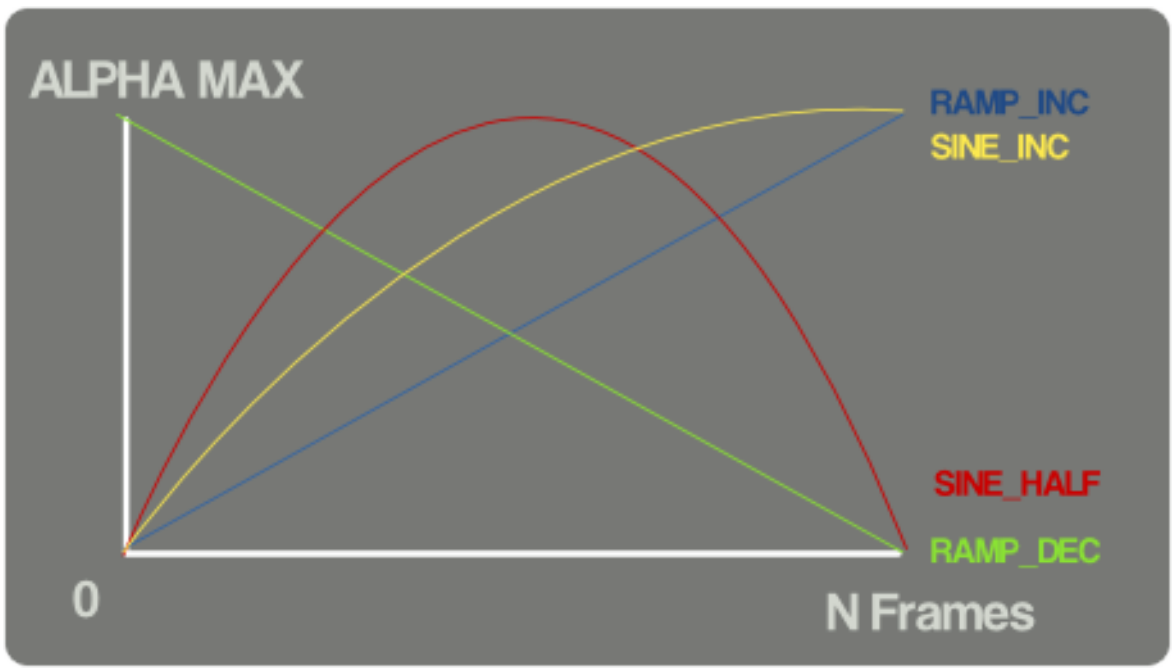
Chapter 7. Effects

7.1. Using Effects

Cluttermm provides several effects methods that can be used together with a timeline to change the properties of a single actor over time, using a simple numeric calculation. In many cases this is an easier way to implement animation. For instance, `Clutter::EffectTemplate::fade()` gradually changes the opacity of an actor or `Clutter::EffectTemplate::rotate()` gradually changes the rotation of an actor, calculating the opacity or rotation by calling the supplied *`alpha_func`* slot.

To use a clutter effect, you should first create a `Clutter::EffectTemplate`, specifying your timeline object and an *`alpha_func`* function slot. This function will need to call `Clutter::Alpha::get_timeline()` so it can return a value based on the timeline's current frame number and total number of frames, using `Clutter::Timeline::get_current_frame()` and `Clutter::Timeline::get_n_frames()`. The result should be between `0` and `Clutter::Alpha::MAX_ALPHA`, with the meaning of the result depending on the effect used. For instance, `Clutter::Alpha::MAX_ALPHA` would be 100% opacity when using `Clutter::EffectTemplate::fade()`. Several built-in callbacks, such as `Clutter::Alpha::sine_func()`, allow you to easily specify natural movement.

Figure 7-1. Graphic representation of some alpha functions.



You should then use one of the effect methods of this `Clutter::EffectTemplate`, and pass along the actor and any extra parameters required by that method.

To make it easier to use different timelines with different effects, you can use `Clutter::EffectTemplate::set_timeline_clone()` to cause the effect to clone (copy instead of just referencing) the timeline, allowing you to change the original timeline and supply it to a second effect, without influencing the first effect.

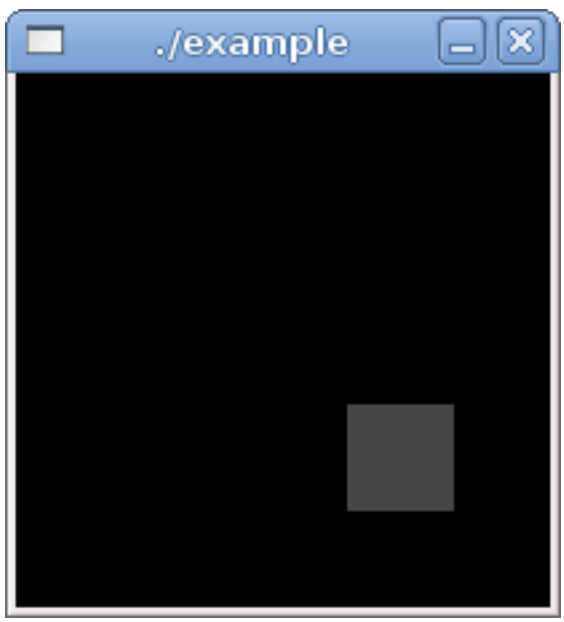
The effects API is actually a simplified API that wraps the `Clutter::Behaviour` objects. However, the effect methods can only control one actor at a time and do not allow you to change the effects while the timeline is running. To do this you can use the behaviours directly, as described in the Behaviours section.

`Clutter::EffectTemplate` class reference (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1EffectTemplate.html)

7.2. Example

The following example demonstrates the use of both a fade effect and a path effect on the same actor, changing a rectangle's opacity while it is moved a long a straight line:

Figure 7-2. Behaviour



Source code (../../../examples/effects)

File: `main.cc`

```
#include <cluttermm.h>
#include <vector>

namespace
{

Glib::RefPtr<Clutter::Rectangle> rect;

/*
 * This must return a value between 0 and Clutter::Alpha::MAX_ALPHA,
 * where 0 means the start of the path, and Clutter::Alpha::MAX_ALPHA
 * is the end of the path.
 *
 * This will be called as many times per seconds as specified in our
 * call to Clutter::Timeline::create().
 *
 * See also, for instance Clutter::Alpha::sine_half_func for a useful
 * built-in callback.
 */
static guint32 on_alpha(const Glib::RefPtr<Clutter::Alpha>& alpha)
{
  // Get the position in the timeline, so we can base our value upon it:
  const Glib::RefPtr<Clutter::Timeline> timeline = alpha->get_timeline();
  const int current_frame_num = timeline->get_current_frame();
  const int n_frames = timeline->get_n_frames();

  // Return a value that is simply proportional to the frame position:
  return Clutter::Alpha::MAX_ALPHA * current_frame_num / n_frames;
}

} // anonymous namespace

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF));

  // Add a rectangle to the stage:
  rect = Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  rect->set_size(40, 40);
  rect->set_position(10, 10);
  stage->add_actor(rect);
  rect->show();
```

```
  // Show the stage:
  stage->show();

  {
    const Glib::RefPtr<Clutter::Timeline>
      timeline = Clutter::Timeline::create(100 /*frames*/, 30 /*fps*/);
    timeline->set_loop();
    timeline->start();

    // Instead of our custom callback, we could use a standard callback.
    // For instance, Clutter::Alpha::sine_inc_func.
    const Glib::RefPtr<Clutter::EffectTemplate>
      effect = Clutter::EffectTemplate::create(timeline, &on_alpha);

    std::vector<Clutter::Knot> knots (2);
    knots[0].set_xy(10, 10);
    knots[1].set_xy(150, 150);

    // Move the actor along the path:
    effect->path(rect, knots);

    // Also change the actor's opacity while moving it along the path:
    // (You would probably want to use a different ClutterEffectTemplate,
    // so you could use a different alpha callback for this.)
    effect->fade(rect, 50);
  }

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

  return 0;
}
```

Chapter 8. Behaviours

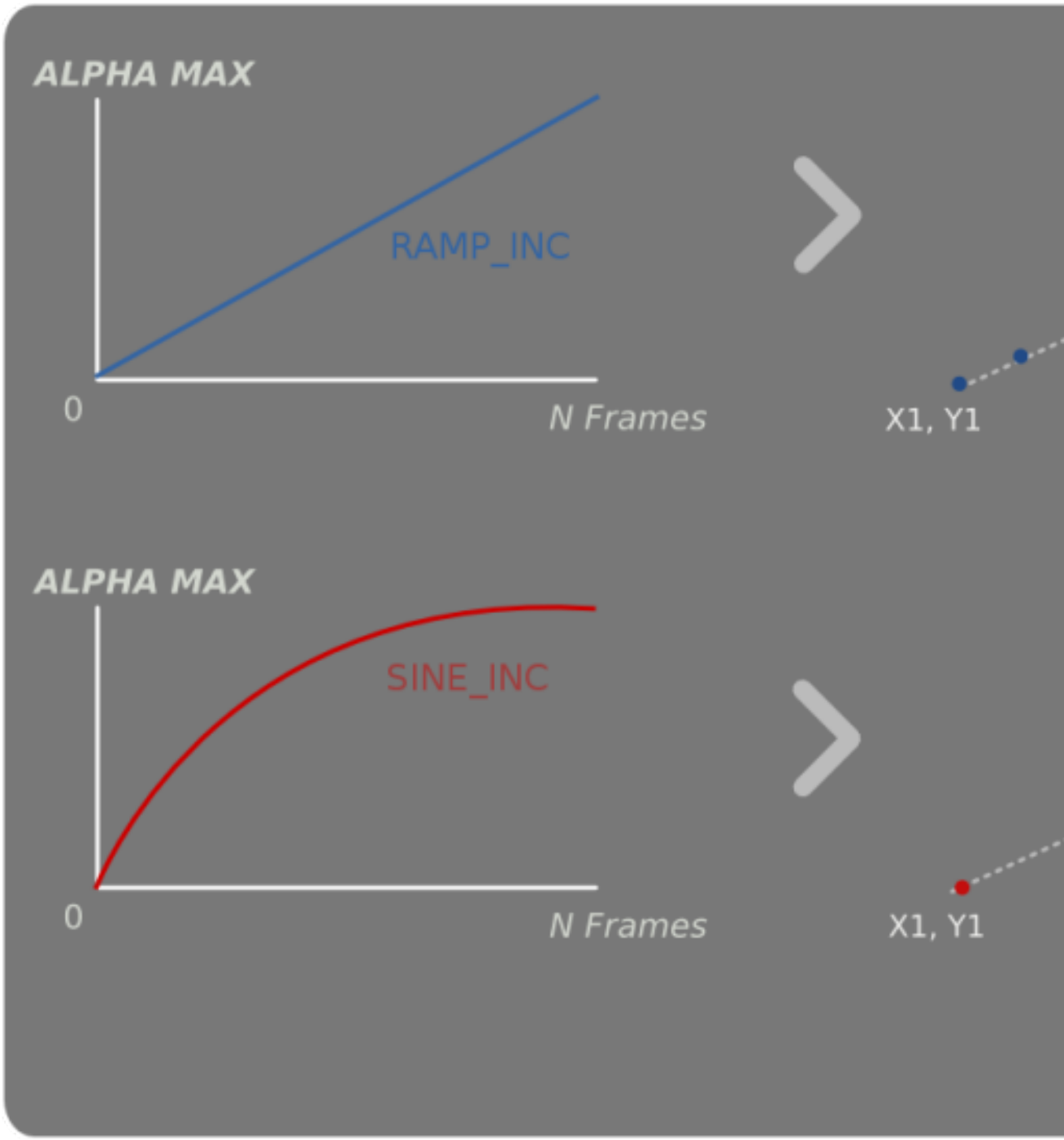
8.1. Using Behaviours

The Effects API is simple but you will often need to use behaviours directly to have more control.

Although the `Clutter::Timeline`'s `signal_new_frame()` signal allows you to set actor properties for each frame, Cluttermm also provides `Behaviour`s which can change specific properties of one specific actor over time, using a simple numeric calculation. However, unlike the simplified Effects 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, `Clutter::BehaviourPath` moves the actor along a specified path, calculating the position on the path once per frame by calling a supplied *`alpha_func`* slot. The `Clutter::Alpha` object is constructed with this slot and a `Clutter::Timeline` which tells it when a new frame needs a new value to be calculated.

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



Your *alpha_func* slot will need to call `Clutter::Alpha::get_timeline()` so it can return a value based on the timeline's current frame number and total number of frames, using `Clutter::Timeline::get_current_frame()` and `Clutter::Timeline::get_n_frames()`. Several built-in callbacks, such as `Clutter::Alpha::sine_func`, allow you to easily specify natural movement.

If the behaviour's timeline is started and not stopped then the end point of the behaviour will always be reached and it will end there unless the timeline is set to loop. For instance, an actor will move along a path until it has reached the end, taking as much time as specified by the timeline's number of frames and frames per second.

Like containers, behaviours maintain a reference to the supplied `Clutter::Alpha` object. For this reason it is not necessary to keep around a reference to the alpha object yourself. However, you should clear the smart pointers to the behaviours when you are finished with them.

`Clutter::BehaviourPath` class reference (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Behaviour.html)

`Clutter::Alpha` class reference (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1Alpha.html)

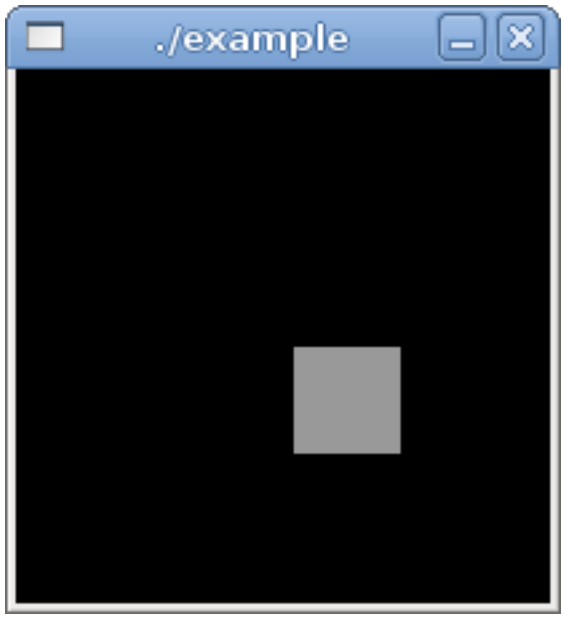
The following standard behaviours are available in Cluttermm:

- `Clutter::BehaviourDepth` (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1BehaviourDepth.html): Moves the actor along the z axis.
- `Clutter::BehaviourEllipse` (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1BehaviourEllipse.html): Moves the actor around an ellipse.
- `Clutter::BehaviourOpacity` (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1BehaviourOpacity.html): Changes the opacity of the actor.
- `Clutter::BehaviourPath` (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1BehaviourPath.html): Moves the actor along straight lines and bezier curves.
- `Clutter::BehaviourRotate` (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1BehaviourRotate.html): Rotates the actor.
- `Clutter::BehaviourScale` (http://library.gnome.org/devel/cluttermm/unstable/classClutter_1_1BehaviourScale.html): Scales the actor.

8.2. Example

The following example demonstrates the use of a `Clutter::BehaviourPath` with a simple custom alpha function. 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.cc`

```
#include <cluttermm.h>

namespace
{

static Glib::RefPtr<Clutter::Rectangle> rect;

/*
 * This must return a value between 0 and Clutter::Alpha::MAX_ALPHA, where
 * 0 means the start of the path, and Clutter::Alpha::MAX_ALPHA is the end
 * of the path.
 *
 * This will be called as many times per seconds as specified in our call
 * to Clutter::Timeline::create().
```

```
 *
 * See also, for instance Clutter::Alpha::sine_half_func for a useful
 * built-in callback.
 */
static guint32 on_alpha(Glib::RefPtr<Clutter::Alpha> alpha)
{
  // Get the position in the timeline, so we can base our value upon it:
  const Glib::RefPtr<Clutter::Timeline> timeline = alpha->get_timeline();

  // Return a value that is simply proportional to the frame position:
  return timeline->get_progress();
}

} // anonymous namespace

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF)); // black

  // Add a rectangle to the stage:
  rect = Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  rect->set_size(40, 40);
  rect->set_position(10, 10);
  stage->add_actor(rect);
  rect->show();

  // Show the stage:
  stage->show();

  {
    const Glib::RefPtr<Clutter::Timeline>
      timeline = Clutter::Timeline::create(5000 /* milliseconds */);
    timeline->set_loop(true);
    timeline->start();

    // Instead of our custom callback, we could use a standard callback.
    // For instance, Clutter::Alpha::sine_inc_func.
    const Glib::RefPtr<Clutter::Alpha>
      alpha = Clutter::Alpha::create(timeline, &on_alpha);

    std::vector<Clutter::Knot> knots (2);
    knots[0].set_xy(10, 10);
    knots[1].set_xy(150, 150);

    const Glib::RefPtr<Clutter::Behaviour>
      behaviour = Clutter::BehaviourPath::create_with_knots(alpha, knots);
    behaviour->apply(rect);
  }
```

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

  return 0;
}
```

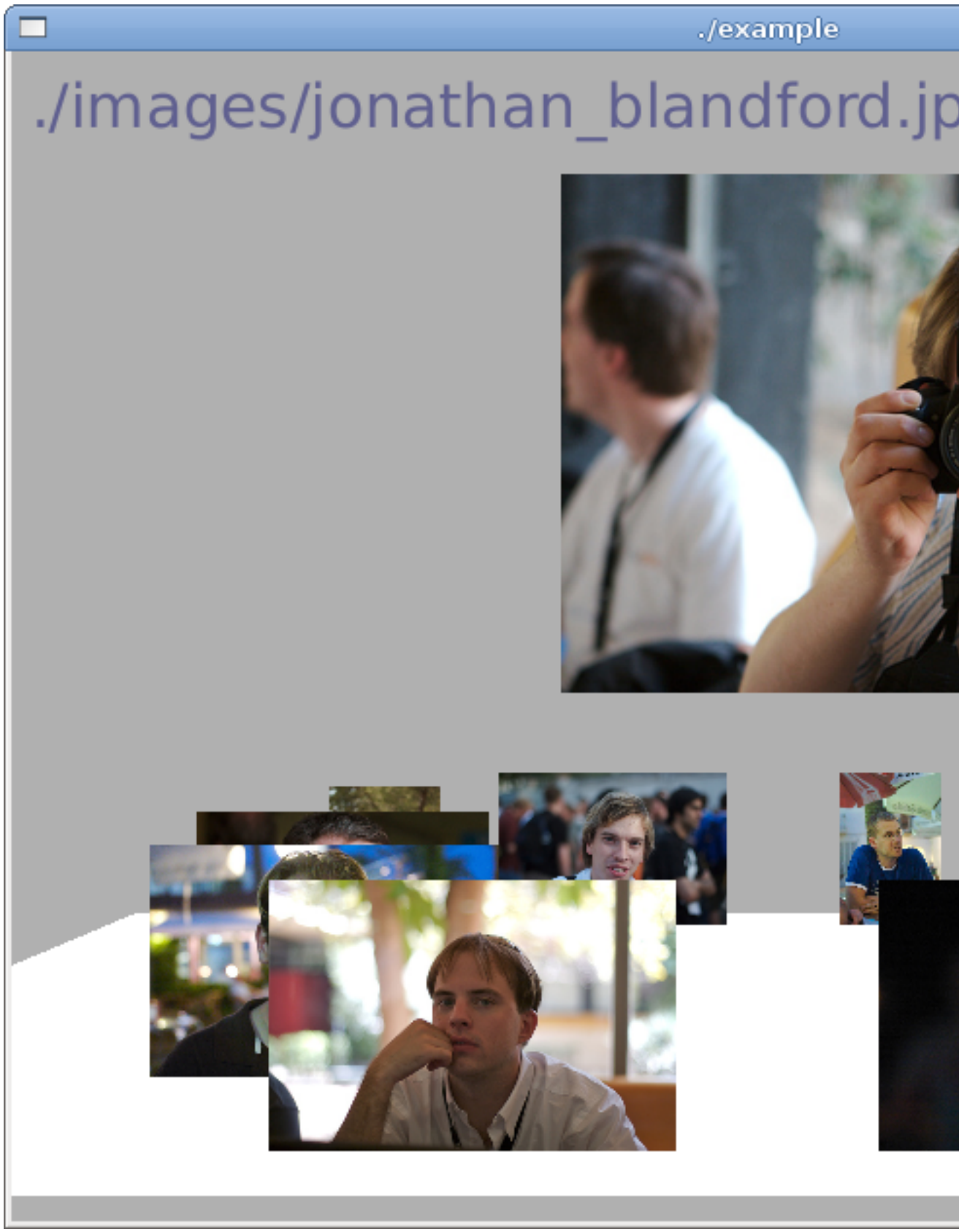
Chapter 9. Full Example

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 9-1. Full Example



Source code (../../../examples/full_example)

File: `main.cc`

```
#include <cluttermm.h>
#include <glibmm.h>
#include <algorithm>
#include <list>
#include <string>
#include <iostream>
#include <cmath>

namespace
{

enum
{
  ELLIPSE_Y = 390, // The y position of the ellipse of images.
  ELLIPSE_HEIGHT = 450, // The distance from front to back when it's rotated 90 degrees.
  IMAGE_HEIGHT = 100
};

const double angle_step = 30.0;

class Item
{
public:
  std::string filepath;
  Glib::RefPtr<Clutter::Texture> texture;
  Glib::RefPtr<Clutter::BehaviourEllipse> behaviour;

  Item() {}

  //This can throw an exception if the file could not be found:
  explicit Item(const std::string& path)
    : filepath (path), texture (Clutter::Texture::create_from_file(path)) {}
};

class Example : public sigc::trackable
{
public:
  Example();
  virtual ~Example();

private:
  std::list<Item> items_;
  std::list<Item>::iterator front_item_;

  Glib::RefPtr<Clutter::Stage> stage_;
```

```
  // For showing the filename:
  Glib::RefPtr<Clutter::Text> label_filename_;

  // For rotating all images around an ellipse::
  Glib::RefPtr<Clutter::Timeline> timeline_rotation_;

  // For moving one image up and scaling it::
  Glib::RefPtr<Clutter::Timeline> timeline_moveup_;
  Glib::RefPtr<Clutter::Behaviour> behaviour_scale_;
  Glib::RefPtr<Clutter::Behaviour> behaviour_path_;
  Glib::RefPtr<Clutter::Behaviour> behaviour_opacity_;

  void load_images(const std::string& directory_path);
  void add_image_actors();

  bool on_texture_button_press(Clutter::ButtonEvent* event, std::list<Item>::iterator pitem
  void on_timeline_moveup_completed();
  void on_timeline_rotation_completed();
  void rotate_item_to_front(std::list<Item>::iterator pitem);
};

static void scale_texture_default(const Glib::RefPtr<Clutter::Texture>& texture)
{
  int width = 0;
  int height = 0;
  texture->get_base_size(width, height);

  if(height > 0)
  {
    const double scale = IMAGE_HEIGHT / double(height);
    texture->set_scale(scale, scale);
  }
}

Example::Example()
:
  items_             (),
  front_item_        (items_.end()),
  stage_             (Clutter::Stage::get_default()),
  label_filename_    (Clutter::Text::create()),
  timeline_rotation_ (Clutter::Timeline::create(2000 /* milliseconds */))
{
  stage_->set_size(800, 600);
  stage_->set_color(Clutter::Color(0xB0, 0xB0, 0xB0, 0xFF)); // light gray

  // Create and add a label actor, hidden at first:
  label_filename_->set_color(Clutter::Color(0x60, 0x60, 0x90, 0xFF)); // blueish
  label_filename_->set_font_name("Sans 24");
  label_filename_->set_position(10, 10);
  label_filename_->set_opacity(0);
  stage_->add_actor(label_filename_);
  label_filename_->show();
```

```
  // Add a plane under the ellipse of images:
  const Glib::RefPtr<Clutter::Actor> rect =
      Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0xFF)); // white

  rect->set_width(stage_->get_width() + 100);
  rect->set_height(ELLIPSE_HEIGHT + 20);
  // Position it so that its center is under the images:
  rect->set_position((stage_->get_width() - rect->get_width()) / 2,
                     ELLIPSE_Y + IMAGE_HEIGHT - rect->get_height() / 2);

  // Rotate it around its center:
  rect->set_rotation(Clutter::X_AXIS, -90.0, 0, rect->get_height() / 2, 0);
  stage_->add_actor(rect);
  rect->show();

  // Show the stage:
  stage_->show();

  timeline_rotation_->signal_completed()
    .connect(sigc::mem_fun(*this, &Example::on_timeline_rotation_completed));

  // Add an actor for each image:
  load_images("images");
  add_image_actors();
#if 0 //TODO: What's this?
  timeline_rotation_->set_loop(true);
#endif
  // Move them a bit to start with:
  if(!items_.empty())
    rotate_item_to_front(items_.begin());
}

Example::~Example()
{}

void Example::load_images(const std::string& directory_path)
{
  // Clear any existing images
  items_.clear();

  Glib::Dir dir(directory_path);

  for(Glib::Dir::iterator p = dir.begin(); p != dir.end(); ++p)
  {
    const std::string filename = *p;

    //Use only .jpg files:
    const std::string::size_type size = filename.size();

    const std::string suffix = ".jpg";
    const std::string::size_type suffix_size = suffix.size();
    if(size < suffix_size)
```

```
      continue;

    const std::string possible_suffix = filename.substr(size - suffix_size);
    if(possible_suffix != suffix)
      continue;

    //Use the file:
    const std::string path = Glib::build_filename(directory_path, *p);
    try
    {
      const Item item (path);

      scale_texture_default(item.texture);
      items_.push_back(item);
    }
    catch (const Glib::Error& ex)
    {
      std::cerr << "Exception when loading image file: " << ex.what() << std::endl;
    }
  }
}

void Example::add_image_actors()
{
  int x = 20;
  int y = 0;
  double angle = 0.0;

  for(std::list<Item>::iterator p = items_.begin(); p != items_.end(); ++p)
  {
    const Glib::RefPtr<Clutter::Actor> actor = p->texture;

    // Add the actor to the stage:
    stage_->add_actor(actor);

    // Set an initial position:
    actor->set_position(x, y);
    y += 100;

    // Allow the actor to emit events.  By default only the stage does this.
    actor->set_reactive(true);

    actor->signal_button_press_event().connect(
        sigc::bind(sigc::mem_fun(*this, &Example::on_texture_button_press), p));

    const Glib::RefPtr<Clutter::Alpha> alpha =
        Clutter::Alpha::create(timeline_rotation_, CLUTTER_EASE_OUT_SINE);

    double angle_end = 0;
    if(angle > 0)
      angle_end = angle - 1; //We want a full rotation, but we can't have > 360 degrees.
    p->behaviour = Clutter::BehaviourEllipse
        ::create(alpha, 320, ELLIPSE_Y, // x, y
```

```
                ELLIPSE_HEIGHT, ELLIPSE_HEIGHT, // width, height
                Clutter::ROTATE_CW,
                angle, angle_end);

    p->behaviour->set_angle_tilt(Clutter::X_AXIS, -90.0);
    p->behaviour->apply(actor);
    actor->show();

    angle += angle_step;

    //This property may not be > 360:
    if(angle > 360)
     angle -= 360;
  }
}

/*
 * This signal handler is called when the item has finished
 * moving up and increasing in size.
 */
void Example::on_timeline_moveup_completed()
{
  // Forget this timeline because we have now finished with it:
  timeline_moveup_.reset();
  behaviour_scale_.reset();
  behaviour_path_.reset();
  behaviour_opacity_.reset();
}

/*
 * This signal handler is called when the items have completely
 * rotated around the ellipse.
 */
void Example::on_timeline_rotation_completed()
{
  // 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:
  const Glib::RefPtr<Clutter::Actor> actor = front_item_->texture;
  timeline_moveup_ = Clutter::Timeline::create(1000 /* milliseconds */);
  const Glib::RefPtr<Clutter::Alpha> alpha =
      Clutter::Alpha::create(timeline_moveup_, CLUTTER_EASE_OUT_SINE);

  // Scale the item from its normal scale to approximately twice the normal scale:
  double scale_start = 0.0;
  actor->get_scale(scale_start, scale_start);
  const double scale_end = scale_start * 1.8;

  behaviour_scale_ = Clutter::BehaviourScale::create(alpha,
                                                     scale_start, scale_start,
                                                     scale_end, scale_end);
```

```
  behaviour_scale_->apply(actor);

  // Move the item up the y axis:
  std::vector<Clutter::Knot> knots (2);
  knots[0].set_xy(actor->get_x(), actor->get_y());
  knots[1].set_xy(knots[0].get_x(), knots[0].get_y() - 250);

  behaviour_path_ = Clutter::BehaviourPath::create_with_knots(alpha, knots);
  behaviour_path_->apply(actor);

  // Show the filename gradually:
  label_filename_->set_text(front_item_->filepath);
  behaviour_opacity_ = Clutter::BehaviourOpacity::create(alpha, 0, 255);
  behaviour_opacity_->apply(label_filename_);

  // Start the timeline and handle its "completed" signal so we can unref it:
  timeline_moveup_->signal_completed()
    .connect(sigc::mem_fun(*this, &Example::on_timeline_moveup_completed));
  timeline_moveup_->start();
}

void Example::rotate_item_to_front(std::list<Item>::iterator pitem)
{
  g_return_if_fail(pitem != items_.end());

  timeline_rotation_->stop();

  // Stop the other timeline in case that is active at the same time:
  if(timeline_moveup_)
    timeline_moveup_->stop();

  label_filename_->set_opacity(0);

  // Get the item's position in the list:
  const int pos = std::distance(items_.begin(), pitem);

  if(front_item_ == items_.end())
    front_item_ = items_.begin();

  const int pos_front = std::distance(items_.begin(), front_item_);

  // Calculate the end angle of the first item:
  const double angle_front = 180.0;
  double angle_start = std::fmod(angle_front - (angle_step * pos_front), 360.0);
  double angle_end   = angle_front - (angle_step * pos);

  double angle_diff = 0.0;

  // Set the end angles:
  for(std::list<Item>::iterator p = items_.begin(); p != items_.end(); ++p)
  {
    // Reset its size:
    scale_texture_default(p->texture);
```

```
    angle_start = std::fmod(angle_start, 360.0);
    angle_end   = std::fmod(angle_end,   360.0);

    // Move 360° instead of 0° when moving for the first time,
    // and when clicking on something that is already at the front.
    if(front_item_ == pitem)
      angle_end += 360.0;

    p->behaviour->set_angle_start(angle_start);
    p->behaviour->set_angle_end(angle_end);

    if(p == pitem)
    {
      if(angle_start < angle_end)
        angle_diff = angle_end - angle_start;
      else
        angle_diff = 360 - (angle_start - angle_end);
    }

    // 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;
  }

  timeline_rotation_->set_duration(angle_diff * 0.2);

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

  timeline_rotation_->start();
}

bool Example::on_texture_button_press(Clutter::ButtonEvent* /* event */, std::list<Item>::i
{
  // Ignore the events if the timeline_rotation is running (meaning,
  // if the objects are moving), to simplify things.
  if(timeline_rotation_ && timeline_rotation_->is_playing())
  {
    std::cout << "on_texture_button_press(): ignoring." << std::endl;
    return false;
  }
  std::cout << "on_texture_button_press(): handling." << std::endl;

  rotate_item_to_front(pitem);
  return true;
}

} // anonymous namespace
```

```
int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  Example example;

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

  return 0;
}
```

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 Glib::Object subclass.

```
class Triangle : public Clutter::Actor
{
  // ...
};
```

You should then override the `Clutter::Actor::on_paint()` virtual method in your class:

```
class Triangle : public Clutter::Actor
{
  // ...
protected:
  virtual void on_paint();
};

void Triangle::on_paint()
{
  // ...
}
```

Your `Clutter::Actor::on_paint()` implementation should use the OpenGL API to actually paint something. You will probably need some information from your object's generic `Clutter::Actor` 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 testing for the preprocessor macro `CLUTTER_COGL_HAS_GL` or `CLUTTER_COGL_HAS_GLES`.

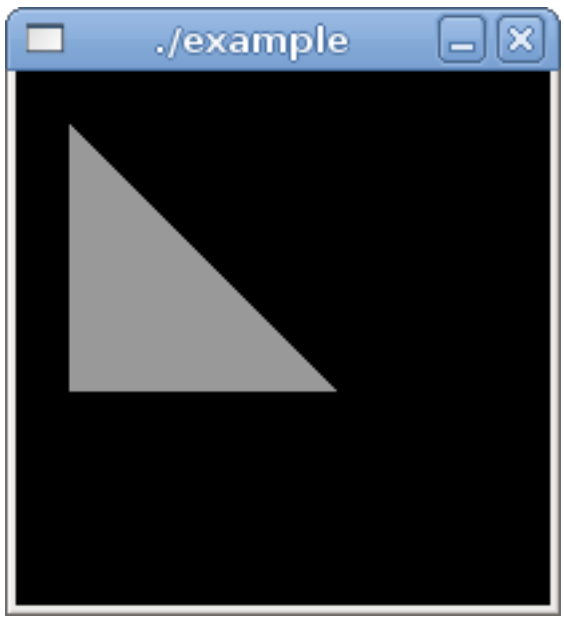
You should also implement the `ClutterActor::pick_vfunc()` virtual function, painting a silhouette of your actor in the provided color. Clutter uses this to draw each actor's silhouette offscreen in a unique color, using the color to quickly identify the actor under the cursor. If your actor is simple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

Most of the rest of `Clutter::Actor`'s virtual functions don't need to be reimplemented, because a suitable default implemention exists in `Clutter::Actor`. For instance, `on_show()`, `on_show_all()`, `on_hide()`, `on_hide_all()`, `allocate_vfunc()`.

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 <cluttermm.h>

namespace Tutorial
{

class Triangle : public Clutter::Actor
{
```

```
public:
  static Glib::RefPtr<Triangle> create();
  static Glib::RefPtr<Triangle> create(const Clutter::Color& color);

  virtual ~Triangle();

  void set_color(const Clutter::Color& color);
  Clutter::Color get_color() const;

protected:
  Triangle();
  explicit Triangle(const Clutter::Color& color);

  virtual void on_paint();
  virtual void pick_vfunc(const Clutter::Color& color);

private:
  Clutter::Color color_;

  void do_triangle_paint(const CoglColor* color);
};

} // namespace Tutorial

#endif /* !CLUTTER_TUTORIAL_TRIANGLE_ACTOR */
```

File: `main.cc`

```
#include "triangle_actor.h"
#include <cluttermm.h>

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF)); // black

  // Add our custom actor to the stage:
  Glib::RefPtr<Tutorial::Triangle> actor =
    Tutorial::Triangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  actor->set_size(100, 100);
  actor->set_position(20, 20);
  stage->add_actor(actor);
  actor->show();

  // Show the stage:
  stage->show();

  // Start the main loop, so we can respond to events:
```

```
  Clutter::main();

  return 0;
}
```

File: `triangle_actor.cc`

```
#include "triangle_actor.h"
#include <cogl/cogl.h>

namespace Tutorial
{

Glib::RefPtr<Triangle> Triangle::create()
{
  return Glib::RefPtr<Triangle>(new Triangle());
}

Glib::RefPtr<Triangle> Triangle::create(const Clutter::Color& color)
{
  return Glib::RefPtr<Triangle>(new Triangle(color));
}

Triangle::Triangle()
:
  color_ (0xFF, 0xFF, 0xFF, 0xFF)
{}

Triangle::Triangle(const Clutter::Color& color)
:
  color_ (color)
{}

Triangle::~Triangle()
{}

void Triangle::do_triangle_paint(const CoglColor* color)
{
  const Clutter::Geometry geom = get_geometry();

  cogl_push_matrix();
  cogl_set_source_color(color);

  Cogl::Fixed coords[6];

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

  coords[2] = COGL_FIXED_FROM_INT(0);
  coords[3] = COGL_FIXED_FROM_INT(geom.get_height());
```

```
  coords[4] = COGL_FIXED_FROM_INT(geom.get_width());
  coords[5] = coords[3];

  cogl_path_polygon((float*)coords, G_N_ELEMENTS(coords) / 2);
  cogl_path_fill();

  cogl_pop_matrix();
}

void Triangle::on_paint()
{
  CoglColor coglcolor;

  // Paint the triangle with the actor's color:
  cogl_color_set_from_4ub(&coglcolor,
    color_.get_red(),
    color_.get_green(),
    color_.get_blue(),
    get_opacity());

  do_triangle_paint(&coglcolor);
}

void Triangle::pick_vfunc(const Clutter::Color& color)
{
  // 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.
  CoglColor coglcolor;
  cogl_color_set_from_4ub(&coglcolor,
                          color.get_red(),
                          color.get_green(),
                          color.get_blue(),
                          color.get_alpha());
  do_triangle_paint(&coglcolor);
}

/**
 * Tutorial::Triangle::get_color:
 *
 * @returns the color of the triangle.
 */
Clutter::Color Triangle::get_color() const
{
  return color_;
}

/**
 * Tutorial::Triangle::set_color:
 * @color: a Clutter::Color
 *
 * Sets the color of the triangle.
```

```
 */
void Triangle::set_color(const Clutter::Color& color)
{
  color_ = color;
  set_opacity(color_.get_alpha());

  if(is_visible())
    queue_redraw();
}

} // namespace Tutorial
```

A.3. Implementing Container Actors

You can create container actors that arrange child actors by implementing the Clutter::Container interface in your actor. This is done by using multiple inheritance to specify that the type is derived from `Clutter::Actor` and also implements the `Clutter::Container` interface. For instance:

```
class ExampleContainer : public Clutter::Actor, public Clutter::Container
{
  ExampleContainer();
  // ...
};

ExampleContainer::ExampleContainer()
:
  // Create a named GObject type for the custom container class
  Glib::ObjectBase(typeid(ExampleContainer))
{}
```

A.3.1. Clutter::Actor virtual functions to implement

If your container should control the position or size of its children then it should implement the `Clutter::Actor`'s `allocate_vfunc()`, `get_preferred_width_vfunc()` and `get_preferred_height_vfunc()` virtual methods.

For instance, your `allocate_vfunc()` 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_vfunc()` and `get_preferred_height_vfunc()` 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 `property_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 `on_paint()` method, usually calling `Clutter::Actor::paint()` on the child actors. All containers should also implement the `pick_vfunc()` method, calling `Clutter::Actor::pick()` on each child actor.

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

A.3.2. Clutter::Container virtual methods to implement

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

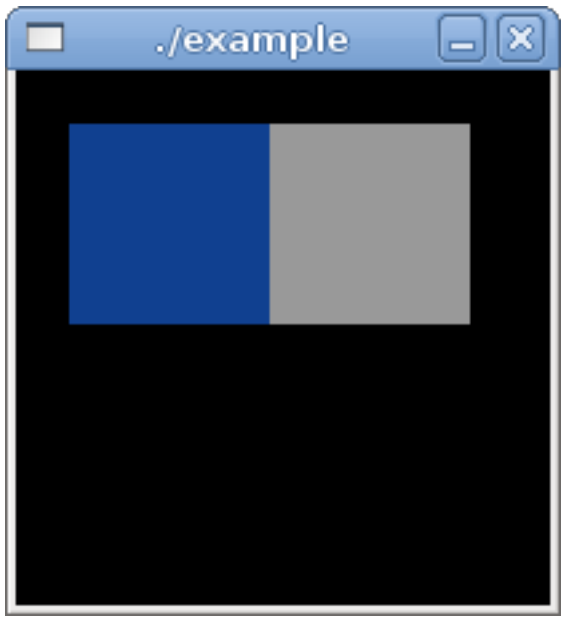
For instance, your `add_vfunc()` and `remove_vfunc()` implementions should manage your container's internal list of child actors and might need to trigger repositioning or resizing of the child actors by calling `Clutter::Actor::queue_relayout()`.

Your `foreach_vfunc()` 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_vfunc(), remove_vfunc() and foreach_vfunc().

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 CLUTTER_TUTORIAL_EXAMPLEBOX_H
#define CLUTTER_TUTORIAL_EXAMPLEBOX_H

#include <cluttermm.h>
#include <list>

namespace Tutorial
{

class Box : public Clutter::Actor, public Clutter::Container
{
public:
  virtual ~Box();
  static Glib::RefPtr<Box> create();

  void remove_all();

protected:
  Box();

  // Clutter::Container interface:
```

```
  virtual void add_vfunc(const Glib::RefPtr<Actor>& actor);
  virtual void remove_vfunc(const Glib::RefPtr<Actor>& actor);
  virtual void raise_vfunc(const Glib::RefPtr<Actor>& actor, const Glib::RefPtr<Actor>& sib
  virtual void lower_vfunc(const Glib::RefPtr<Actor>& actor, const Glib::RefPtr<Actor>& sib
  virtual void sort_depth_order_vfunc();
  virtual void foreach_vfunc(ClutterCallback callback, gpointer user_data);

  // Clutter::Actor interface:
  virtual void on_paint();
  virtual void show_all_vfunc();
  virtual void hide_all_vfunc();
  virtual void pick_vfunc(const Clutter::Color& color);
  virtual void get_preferred_width_vfunc(float for_height,
    float& min_width_p, float& natural_width_p);
  virtual void get_preferred_height_vfunc(float for_width,
    float& min_height_p, float& natural_height_p);
  virtual void allocate_vfunc(const Clutter::ActorBox& box, Clutter::AllocationFlags absolu

private:
  typedef std::list< Glib::RefPtr<Clutter::Actor> > ChildrenList;
  ChildrenList children_;
};

} // namespace Tutorial

#endif /* !CLUTTER_TUTORIAL_EXAMPLEBOX_H */
```

File: `examplebox.cc`

```
#include "examplebox.h"
#include <cogl/cogl.h>
#include <algorithm>

namespace Tutorial
{

/*
 * Simple example of a container actor.
 *
 * Tutorial::Box imposes a specific layout on its children, unlike
 * Clutter::Group which is a free-form container.
 *
 * Specifically, Tutorial::Box lays out its children along an imaginary
 * horizontal line.
 */

Box::Box()
:
  // Create a named GObject type for the custom container class:
  Glib::ObjectBase(typeid(Box))
{
  property_request_mode() = Clutter::REQUEST_WIDTH_FOR_HEIGHT;
```

```
}

Box::~Box()
{
  remove_all();
}

Glib::RefPtr<Box> Box::create()
{
  return Glib::RefPtr<Box>(new Box());
}

void Box::remove_all()
{
  while (!children_.empty())
    remove_actor(children_.front());
}

void Box::add_vfunc(const Glib::RefPtr<Clutter::Actor>& actor)
{
  children_.push_front(actor);

  // Ugly but necessary:  Explicitely acquire an additional reference
  // because Glib::RefPtr assumes ownership.
  actor->set_parent(Glib::RefPtr<Clutter::Actor>((reference(), this)));
  actor_added(actor);
  queue_relayout();
}

void Box::remove_vfunc(const Glib::RefPtr<Clutter::Actor>& actor)
{
  const Glib::RefPtr<Clutter::Actor> element = actor;
  const ChildrenList::iterator p = std::find(children_.begin(), children_.end(), element);

  if(p != children_.end())
  {
    element->unparent();
    children_.erase(p);
    actor_removed(element);
    queue_relayout();
  }
}

void Box::raise_vfunc(const Glib::RefPtr<Clutter::Actor>&, const Glib::RefPtr<Clutter::Acto
{
  g_assert_not_reached();
}

void Box::lower_vfunc(const Glib::RefPtr<Clutter::Actor>&, const Glib::RefPtr<Clutter::Acto
{
  g_assert_not_reached();
}
```

```
void Box::sort_depth_order_vfunc()
{
  g_assert_not_reached();
}

void Box::foreach_vfunc(ClutterCallback callback, gpointer user_data)
{
  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
    callback((*p)->gobj(), user_data);
}

void Box::on_paint()
{
  cogl_push_matrix();

  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
  {
    if((*p)->is_mapped())
      (*p)->paint();
  }

  cogl_pop_matrix();
}

void Box::show_all_vfunc()
{
  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
    (*p)->show();

  show();
}

void Box::hide_all_vfunc()
{
  hide();

  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
    (*p)->hide();
}

void Box::pick_vfunc(const Clutter::Color& color)
{
  // Call the base class so we get  a bounding box painted (if we are reactive):
  Clutter::Actor::pick_vfunc(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(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
  {
    if((*p)->is_mapped())
      (*p)->paint();
```

```
  }
}

/*
 * 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.
 */
void Box::get_preferred_width_vfunc(float  for_height,
                                    float& min_width_p,
                                    float& natural_width_p)
{
  float min_width     = 0;
  float natural_width = 0;

  // Calculate the preferred width for this container,
  // based on the preferred width requested by the children.
  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
    if((*p)->is_visible())
    {
      float child_min_width     = 0;
      float child_natural_width = 0;

      (*p)->get_preferred_width(for_height, child_min_width, child_natural_width);

      min_width     += child_min_width;
      natural_width += child_natural_width;
    }

  min_width_p     = min_width;
  natural_width_p = natural_width;
}

/*
 * For this container, the preferred height is the maximum height
 * of the children. The preferred height is independent of the given width.
 */
void Box::get_preferred_height_vfunc(float /* for_width */,
                                     float& min_height_p,
                                     float& natural_height_p)
{
  float min_height     = 0;
  float natural_height = 0;

  // Calculate the preferred height for this container,
  // based on the preferred height requested by the children.
  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
    if((*p)->is_visible())
    {
      float child_min_height     = 0;
      float child_natural_height = 0;

      (*p)->get_preferred_height(-1, child_min_height, child_natural_height);
```

```
      min_height     = std::max(min_height,     child_min_height);
      natural_height = std::max(natural_height, child_natural_height);
    }

  min_height_p     = min_height;
  natural_height_p = natural_height;
}

void Box::allocate_vfunc(const Clutter::ActorBox& box, Clutter::AllocationFlags absolute_or
{
  float child_x = 0;

  for(ChildrenList::iterator p = children_.begin(); p != children_.end(); ++p)
  {
    float min_width    = 0;
    float min_height   = 0;
    float child_width  = 0;
    float child_height = 0;

    (*p)->get_preferred_size(min_width, min_height, child_width, child_height);

    // Calculate the position and size that the child may actually have.
    // Position the child just after the previous child, horizontally.
    const Clutter::ActorBox child_box (child_x, 0, child_x + child_width, child_height);
    child_x += child_width;

    // Tell the child what position and size it may actually have:
    (*p)->allocate(child_box, absolute_origin_changed);
  }

  Clutter::Actor::allocate_vfunc(box, absolute_origin_changed);
}

} // namespace Tutorial
```

File: `main.cc`

```
#include "examplebox.h"
#include <cluttermm.h>

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF)); // black

  // Add our custom container to the stage:
  const Glib::RefPtr<Tutorial::Box> box = Tutorial::Box::create();
```

```
  // Set the size to the preferred size of the container:
  box->set_size(-1, -1);
  box->set_position(20, 20);

  // Add some actors to our container:
  const Glib::RefPtr<Clutter::Rectangle>
    rect1 = Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xFF, 0x99));
  rect1->set_size(75, 75);
  box->add_actor(rect1);

  const Glib::RefPtr<Clutter::Rectangle>
    rect2 = Clutter::Rectangle::create(Clutter::Color(0x10, 0x40, 0x90, 0xFF));
  rect2->set_size(75, 75);
  box->add_actor(rect2);

  stage->add_actor(box);
  box->show_all();
  stage->show();

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

  return 0;
}
```

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 `Gtk::ScrolledWindow` widget in the GTK+ toolkit, so you may need to implement this functionality in your application.

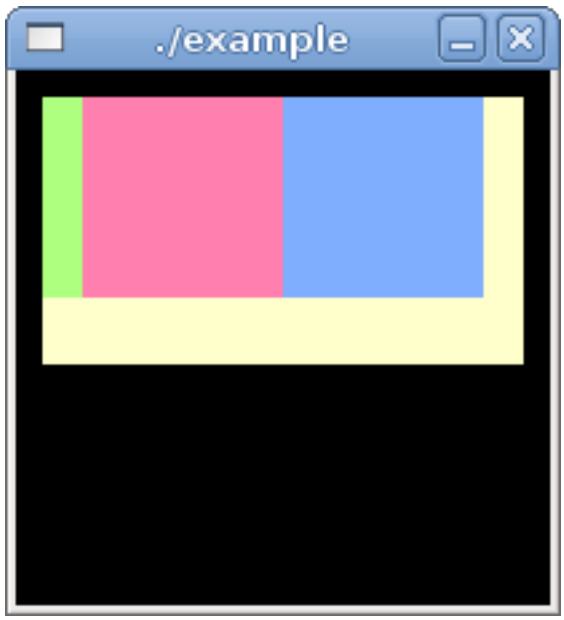
The Tidy project contains some suggested implementations for such actors, but here is a simpler example of the general technique. It creates the impression of scrolling by clipping a container so that it only shows a small area of the screen, while moving the child widgets in that container.

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 CLUTTER_TUTORIAL_SCROLLINGCONTAINER_H
#define CLUTTER_TUTORIAL_SCROLLINGCONTAINER_H

#include <cluttermm.h>

namespace Tutorial
{

class ScrollingContainer : public Clutter::Actor, public Clutter::Container
{
public:
  virtual ~ScrollingContainer();
  static Glib::RefPtr<ScrollingContainer> create();

  void scroll_left(int distance);

protected:
  ScrollingContainer();

  // Clutter::Container interface:
  virtual void add_vfunc(const Glib::RefPtr<Actor>& actor);
```

```
  virtual void remove_vfunc(const Glib::RefPtr<Actor>& actor);
  virtual void raise_vfunc(const Glib::RefPtr<Actor>& actor, const Glib::RefPtr<Actor>& sib
  virtual void lower_vfunc(const Glib::RefPtr<Actor>& actor, const Glib::RefPtr<Actor>& sib
  virtual void sort_depth_order_vfunc();
  virtual void foreach_vfunc(ClutterCallback callback, gpointer user_data);

  // Clutter::Actor interface:
  virtual void on_paint();
  virtual void on_show();
  virtual void on_hide();
  virtual void show_all_vfunc();
  virtual void hide_all_vfunc();
  virtual void pick_vfunc(const Clutter::Color& color);
  virtual void allocate_vfunc(const Clutter::ActorBox& box, Clutter::AllocationFlags absolu

private:
  Glib::RefPtr<Clutter::Rectangle> border_;
  Glib::RefPtr<Clutter::Group> children_;
  int offset_;
};

} // namespace Tutorial

#endif /* CLUTTER_TUTORIAL_SCROLLINGCONTAINER_H */
```

File: `scrollingcontainer.cc`

```
#include "scrollingcontainer.h"
#include <algorithm>

namespace
{

static void allocate_child(const Glib::RefPtr<Clutter::Actor>& actor,
  int& child_x, Clutter::AllocationFlags absolute_origin_changed)
{
  float min_width  = 0;
  float min_height = 0;
  float width      = 0;
  float height     = 0;

  actor->get_preferred_size(min_width, min_height, width, height);

  const Clutter::ActorBox child_box (child_x, 0,
    child_x + width, height);

  actor->allocate(child_box, absolute_origin_changed);

  child_x += width;
}

} // anonymous namespace
```

```
namespace Tutorial
{

/*
 * Tutorial::ScrollingContainer shows only a small area of its child
 * actors, and the child actors can be scrolled left under that area.
 */

ScrollingContainer::ScrollingContainer()
:
  Glib::ObjectBase(typeid(ScrollingContainer)),
  border_(Clutter::Rectangle::create(Clutter::Color(0xFF, 0xFF, 0xCC, 0xFF))),
  children_(Clutter::Group::create()),
  offset_(0)
{
  // Ugly but necessary: Explicitely acquire an additional reference
  // because Glib::RefPtr assumes ownership.
  const Glib::RefPtr<Clutter::Actor> self ((reference(), this));
  border_->set_parent(self);
  children_->set_parent(self);

  children_->signal_actor_added().connect(sigc::mem_fun(*this, &ScrollingContainer::actor_a
  children_->signal_actor_removed().connect(sigc::mem_fun(*this, &ScrollingContainer::actor
}

ScrollingContainer::~ScrollingContainer()
{
  children_->unparent();
  border_->unparent();
}

Glib::RefPtr<ScrollingContainer> ScrollingContainer::create()
{
  return Glib::RefPtr<ScrollingContainer>(new ScrollingContainer());
}

/*
 * Scroll all the child widgets left, resulting in some parts
 * being hidden, and some parts becoming visible.
 */
void ScrollingContainer::scroll_left(int distance)
{
  offset_ += distance;
  queue_relayout();
}

void ScrollingContainer::add_vfunc(const Glib::RefPtr<Clutter::Actor>& actor)
{
  children_->add_actor(actor);
  queue_relayout();
}
```

```
void ScrollingContainer::remove_vfunc(const Glib::RefPtr<Clutter::Actor>& actor)
{
  children_->remove_actor(actor);
  queue_relayout();
}

void ScrollingContainer::raise_vfunc(const Glib::RefPtr<Clutter::Actor>&,
  const Glib::RefPtr<Clutter::Actor>&)
{
  g_assert_not_reached();
}

void ScrollingContainer::lower_vfunc(const Glib::RefPtr<Clutter::Actor>&,
  const Glib::RefPtr<Clutter::Actor>&)
{
  g_assert_not_reached();
}

void ScrollingContainer::sort_depth_order_vfunc()
{
  g_assert_not_reached();
}

void ScrollingContainer::foreach_vfunc(ClutterCallback callback, gpointer user_data)
{
  clutter_container_foreach(children_->Clutter::Container::gobj(), callback, user_data);
}

void ScrollingContainer::on_paint()
{
  border_  ->paint();
  children_->paint();
}

void ScrollingContainer::on_show()
{
  border_  ->show();
  children_->show();
  Clutter::Actor::on_show();
}

void ScrollingContainer::on_hide()
{
  Clutter::Actor::on_hide();
  children_->hide();
  border_  ->hide();
}

void ScrollingContainer::show_all_vfunc()
{
  children_->show_all();
  show();
}
```

```
void ScrollingContainer::hide_all_vfunc()
{
  hide();
  children_->hide_all();
}

void ScrollingContainer::pick_vfunc(const Clutter::Color& color)
{
  // Call the base class so we get a bounding box painted (if we are reactive):
  Clutter::Actor::pick_vfunc(color);
}

void ScrollingContainer::allocate_vfunc(const Clutter::ActorBox& box, Clutter::AllocationFl
{
  const float width  = std::max<float>(0, box.get_x2() - box.get_x1());
  const float height = std::max<float>(0, box.get_y2() - box.get_y1());

  Clutter::ActorBox child_box (0, 0, width, height);

  // Position the child at the top of the container:
  children_->allocate(child_box, absolute_origin_changed);

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

  // Show a rectangle border to show the area:
  border_->allocate(child_box, absolute_origin_changed);

  int child_x = -offset_;
  children_->foreach(sigc::bind(&allocate_child, sigc::ref(child_x), absolute_origin_change

  Clutter::Actor::allocate_vfunc(box, absolute_origin_changed);
}

} // namespace Tutorial
```

File: `main.cc`

```
#include "scrollingcontainer.h"
#include <cluttermm.h>
#include <iostream>

namespace
{

static bool on_stage_button_press(Clutter::ButtonEvent*,
  Glib::RefPtr<Tutorial::ScrollingContainer> scrolling)
{
  std::cout << "Scrolling\n";

  scrolling->scroll_left(10);
```

```
  return true; // stop further handling of this event
}

} // anonymous namespace

int main(int argc, char** argv)
{
  Clutter::init(&argc, &argv);

  // Get the stage and set its size and color:
  const Glib::RefPtr<Clutter::Stage> stage = Clutter::Stage::get_default();
  stage->set_size(200, 200);
  stage->set_color(Clutter::Color(0x00, 0x00, 0x00, 0xFF)); // black

  // Add our scrolling container to the stage
  const Glib::RefPtr<Tutorial::ScrollingContainer>
    scrolling = Tutorial::ScrollingContainer::create();
  scrolling->set_size(180, 100);
  scrolling->set_position(10, 10);
  stage->add_actor(scrolling);

  // Add some actors to our container:
  {
    const Glib::RefPtr<Clutter::Actor>
      actor = Clutter::Rectangle::create(Clutter::Color(0x7F, 0xAE, 0xFF, 0xFF));
    actor->set_size(75, 75);
    scrolling->add_actor(actor);
  }

  {
    const Glib::RefPtr<Clutter::Actor>
      actor = Clutter::Rectangle::create(Clutter::Color(0xFF, 0x7F, 0xAE, 0xFF));
    actor->set_size(75, 75);
    scrolling->add_actor(actor);
  }

  {
    const Glib::RefPtr<Clutter::Actor>
      actor = Clutter::Rectangle::create(Clutter::Color(0xAE, 0xFF, 0x7F, 0xFF));
    actor->set_size(75, 75);
    scrolling->add_actor(actor);
  }

  scrolling->show_all();
  stage->show();

  // Connect signal handlers to handle mouse clicks on the stage:
  stage->signal_button_press_event().connect(sigc::bind(&on_stage_button_press, scrolling))

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

```
  return 0;
}
```

Chapter 10. 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 cluttermm_tutorial (http://git.gnome.org/cgit/cluttermm_tutorial/) module in GNOME's git repository.