Design of Multimedia Applications

Assignment 2: GStreamer ! application development

Deadline: 2PM Mar. 15th 2019

1 Introduction

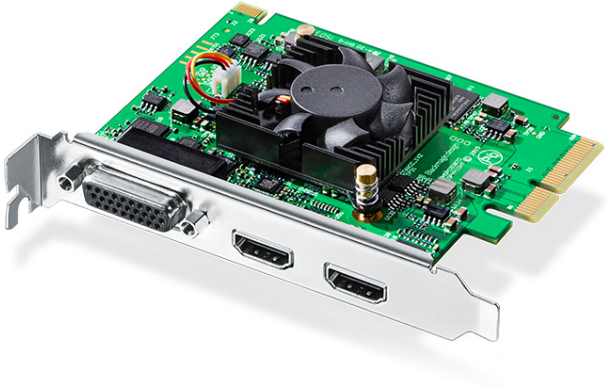
Figure 1: Example of a hardware, 4 channel Roland V4EX videomixer.

During this assignment, the plan is to make a rudimentary version of a 2 channel video mixer. An example of a hardware video mixer is visualized in Figure 1. With such a video mixer, a crossfade can be made between different live video channels combined with different effects applied on the output video. During this assignment, you will have the freedom to apply any GStreamer effect on the video as long as one of the effects is the visulization of a logo.

Due to hardware constraints, the video mixer is also simplified towards video file sources instead of a live HDMI captured video stream. The only changes this would require, compared to the full fledged product, is the change of the file source with a live source

from a possible video capture card, like the Blackmagic Intensity Pro 4K (see Figure 2). Replacing the `filesrc` with a `decklinksrc` and the `autovideosink` with a `decklinksink` would then suffice to make our solution a complete product.

Figure 2: Blackmagic Intensity Pro 4K.



2 Tutorials

The following description is a modified version of:
- the GStreamer SDK Basic tutorials
(`https://gstreamer.freedesktop.org/documentation/tutorials/basic/index.html`).

More help on the Gstreamer API in python can be found on
`https://lazka.github.io/pgi-docs/`

2.1 Tutorial 1: Hello world!

Nothing better to get a first impression about a software library than to print Hello World on the screen!

But since we are dealing with multimedia frameworks, we are going to play a video instead.

Take a look at `basic-tutorial-1.py`. Do not be scared by the amount of code; there are only 4 lines which do real work. The rest is initialization and cleanup code.

```
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, GObject, GLib
```

```
# initialize GStreamer
Gst.init(None)
```

Among other things, `Gst.init()`:

1. Initializes all internal structures
2. Checks what plug-ins are available
3. Executes any command-line option intended for GStreamer

```
# build the pipeline
pipeline = Gst.parse_launch(
    "playbin uri=http://users.datasciencelab.be/MM/tears_of_steel_720p.mp4"
)
```

In GStreamer you usually build the pipeline by manually assembling the individual elements, but, when the pipeline is easy enough, and you do not need any advanced features, you can take the shortcut: `Gst.parse_launch()`.

This function takes a textual representation of a pipeline and turns it into an actual pipeline, which is very handy. In fact, this function is so handy there is a tool built completely around it which you are very acquainted with, namely `gst-launch-1.0`.

If you mistype the URI, or the file does not exist, or you are missing a plug-in, GStreamer provides several notification mechanisms, but the only thing we are doing in this example is exiting on error, so do not expect much feedback.

```
# start playing
pipeline.set_state(Gst.State.PLAYING)
```

This line highlights another interesting concept: the state. Every GStreamer element has an associated state, which you can more or less think of as the Play/Pause button in your regular DVD player. For now, suffice to say that playback will not start unless you set the pipeline to the PLAYING state.

In this line, `pipeline.set_state()` is setting pipeline (our only element, remember) to the PLAYING state, thus initiating playback.

```
# wait until EOS or error
bus = pipeline.get_bus()
msg = bus.timed_pop_filtered(
    Gst.CLOCK_TIME_NONE,
    Gst.MessageType.ERROR | Gst.MessageType.EOS
)
```

These lines will wait until an error occurs or the end of the stream is found. `pipeline.get_bus()` retrieves the pipeline's bus, and `bus.timed_pop_filtered()` will block until you receive either an ERROR or an EOS (End-Of-Stream) through that bus.

And that's it! From this point onwards, GStreamer takes care of everything. Execution will end when the media reaches its end (EOS) or an error is encountered (try closing the video window, or unplugging the network cable). The application can always be stopped by pressing control-C in the console.

Before terminating the application, though, you need to tidy up correctly after ourselves.

```
# free resources
pipeline.set_state(Gst.State.NULL)
```

2.2 Tutorial 2: GStreamer concepts

The previous tutorial showed how to build a pipeline automatically. Now we are going to build a pipeline manually by instantiating each element and linking them all together. In the process, we will learn:

1. What is a GStreamer element and how to create one.
2. How to connect elements to each other.
3. How to customize an element's behavior.
4. How to watch the bus for error conditions and extract information from GStreamer messages.

Run and take a look at `basic-tutorial-2.py`.

```
# create the elements
source = Gst.ElementFactory.make("videotestsrc", "source")
sink = Gst.ElementFactory.make("autovideosink", "sink")
```

As seen in this code, new elements can be created with `Gst.ElementFactory.make()`. The first parameter is the type of element to create. The second parameter is the name we want to give to this particular instance. Naming your elements is useful to retrieve them later if you didn't keep a pointer (and for more meaningful debug output). If you pass NULL for the name, however, GStreamer will provide a unique name for you.

```
# create the empty pipeline
pipeline = Gst.Pipeline.new("test-pipeline")
...
```

```
# build the pipeline
pipeline.add(source)
pipeline.add(sink)
if not source.link(sink):
    print("ERROR: Could not link source to sink")
    sys.exit(1)
```

All elements in GStreamer must typically be contained inside a pipeline before they can be used, because it takes care of some clocking and messaging functions.

A pipeline is a particular type of bin, which is the element used to contain other elements. Therefore all methods which apply to bins also apply to pipelines. In our case, we call `pipeline.add()` to add the elements to the pipeline.

These elements, however, are not linked with each other yet. For this, we need to use `source.link(sink)`. The order counts, because links must be established following the data flow (this is, from source elements to sink elements). Keep in mind that only elements residing in the same bin can be linked together.

```
# modify the source's properties
source.set_property("pattern", 0)
```

Most GStreamer elements have customizable properties: named attributes that can be modified to change the element's behavior (writable properties) or inquired to find out about the element's internal state (readable properties).

The line of code above changes the pattern property of videotestsrc, which controls the type of test video the element outputs. Try different values!

The names and possible values of all the properties an element exposes can be found using the `gst-inspect-1.0` tool.

```
# wait for EOS or error
bus = pipeline.get_bus()
msg = bus.timed_pop_filtered(
    Gst.CLOCK_TIME_NONE,
    Gst.MessageType.ERROR | Gst.MessageType.EOS
)
if msg:
    t = msg.type
    if t == Gst.MessageType.ERROR:
        err, dbg = msg.parse_error()
        print("ERROR:", msg.src.get_name(), " ", err.message)
```

```
        if dbg:
            print("debugging info:", dbg)
    elif t == Gst.MessageType.EOS:
        print("End-Of-Stream reached")
    else:
        # this should not happen. we only asked for ERROR and EOS
        print("ERROR: Unexpected message received.")
```

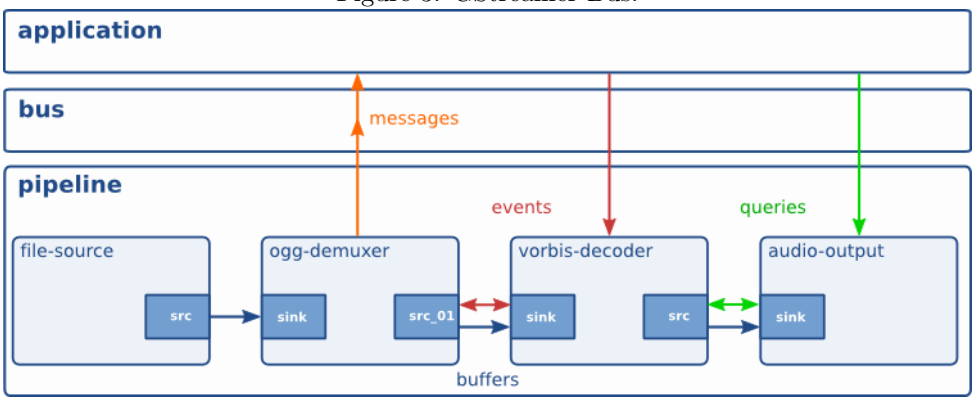
At this point, we have the whole pipeline built and setup, and the rest of the tutorial is very similar to the previous one, but we are going to add more error checking.

`bus.timed_pop_filtered()` waits for execution to end and returns with a GstMessage which we previously ignored. We asked `bus.timed_pop_filtered()` to return when GStreamer encountered either an error condition or an EOS, so we need to check which one happened, and print a message on screen (Your application will probably want to undertake more complex actions).

GstMessage is a very versatile structure which can deliver virtually any kind of information. Fortunately, GStreamer provides a series of parsing functions for each kind of message.

In this case, once we know the message contains an error (by using the `Gst.MessageType`), we can use `msg.parse_error()` which returns a GLib GError error structure and a string useful for debugging.

Figure 3: GStreamer Bus.



At this point it is worth introducing the GStreamer bus a bit more formally (remember Figure 3). It is the object responsible for delivering to the application the GstMessages generated by the elements, in order and to the application thread. This last point is important, because the actual streaming of media is done in another thread than the

application.

Messages can be extracted from the bus synchronously with `bus.timed_pop_filtered()` and its siblings, or asynchronously, using signals. Your application should always keep an eye on the bus to be notified of errors and other playback-related issues.

In the `basic-tutorial-2-ex-vertigo.py` file, you can find an extended example where a video filter element (the vertigotv effect) is added in between the source and the sink of this pipeline. This element needs to be created, added to the pipeline, and linked with the other elements.

2.3 Tutorial 3: Dynamic pipelines

This tutorial shows the rest of the basic concepts required to use GStreamer, which allow building the pipeline "on the fly", as information becomes available, instead of having a monolithic pipeline defined at the beginning of your application.

You can find the code in `basic-tutorial-3.py`. For an explanation of the code, have a look at the tutorial website:

`https://gstreamer.freedesktop.org/documentation/tutorials/basic/dynamic-pipelines.html`

Take into account the code discussed there is C++ instead of python, so try to extract the commonalities.

2.4 Tutorial 4: GUI toolkit integration

This tutorial shows how to integrate GStreamer in a Graphical User Interface (GUI) toolkit like GTK+. Basically, GStreamer takes care of media playback while the GUI toolkit handles user interaction. The most interesting parts are those in which both libraries have to interact: Instructing GStreamer to output video to a GTK+ window and forwarding user actions to GStreamer.

In particular, you will learn:

1. How to tell GStreamer to output video to a particular window (instead of creating its own window).
2. How to continuously refresh the GUI with information from GStreamer.
3. How to update the GUI from the multiple threads of GStreamer.

Figure 4: GUI toolkit integration

4. A mechanism to subscribe only to the messages you are interested in, instead of being notified of all of them.

We are going to build a media player using the GTK+ toolkit, but the concepts apply to other toolkits like QT, for example.

The main point of this tutorial is about telling GStreamer to output the video to a window of our choice. The specific mechanism depends on the operating system (or rather, on the windowing system), but GStreamer provides a layer of abstraction for the sake of platform independence. This independence comes through the XOverlay interface, that allows the application to tell a video sink the handler of the window that should receive the rendering.

Another issue is that GUI toolkits usually only allow manipulation of the graphical widgets through the main (or application) thread, whereas GStreamer usually spawns multiple threads to take care of different tasks. Calling GTK+ functions from within callbacks will usually fail, because callbacks execute in the calling thread, which does not need to be the main thread. This problem can be solved by posting a message on the GStreamer bus in the callback: The messages will be received by the main thread which will then react accordingly.

Let's write a very simple media player based on playbin, this time, with a GUI! You can find the code in `basic-tutorial-5.py`.

This tutorial is composed mostly of callback functions, which will be called from GStreamer or GTK+, so let's review the main function, which registers all these callbacks.

```
# connect to interesting signals in playbin
```

```
self.playbin.connect("video-tags-changed",self.on_tags_changed)
self.playbin.connect("audio-tags-changed",self.on_tags_changed)
self.playbin.connect("text-tags-changed",self.on_tags_changed)
```

We are interested in being notified when new tags (metadata) appear on the stream. We are going to handle all kinds of tags (video, audio and text) from the same callback `self.on_tags_changed`.

```
# create the GUI
self.build_ui()
```

All GTK+ widget creation and signal registration happens in this function. It contains only GTK-related function calls, so we will skip over its definition. Try to get to understand this section by browsing the documentation written on `https://lazka.github.io/pgi-docs/`.

```
# instruct the bus to emit signals for each received message
# and connect to the interesting signals
bus = self.playbin.get_bus()
bus.add_signal_watch()
bus.connect("message::error", self.on_error)
bus.connect("message::eos", self.on_eos)
bus.connect("message::state-changed", self.on_state_changed)
bus.connect("message::application", self.on_application_message)
```

In this application, we can achieve a fine granularity by using signals instead of messages, which allow us to register only to the messages we are interested in. By calling `bus.add_signal_watch()` we instruct the bus to emit a signal every time it receives a message. This signal has the name `message::detail` where detail is the message that triggered the signal emission. For example, when the bus receives the EOS message, it emits a signal with the name `message::eos`. In contrast, if we had registered to the message signal, we would be notified of every single message.

```
# start the GTK main loop. we will not regain control until
# Gtk.main_quit() is called
Gtk.main()
```

Keep in mind that, in order for the bus watches to work, there must be GLib Main Loop running. In this case, it is hidden inside the GTK+ main loop.

```
# register a function that GLib will call every second
GLib.timeout_add_seconds(1, self.refresh_ui)
```

Before transferring control to GTK+, we use `GLib.timeout_add_seconds()` to register yet another callback, this time with a timeout, so it gets called every second. We are going to use it to refresh the GUI from the `refresh_ui` function.

After this, we are done with the setup and can start the GTK+ main loop. We will regain control from our callbacks when interesting things happen. Review the different callbacks by inspecting the comments in the code. Each callback has a different signature, depending on who will call it. You can look up the signature (the meaning of the parameters and the return value) in the documentation of the signal.

3 Assignment

In a television studio, there needs to be a possibility to fade between different videostreams (camera's). You can assume that the system works at a fixed resolution (640x480). Additionally, the editor needs to be able to add some effects to the video whenever these are needed. Therefore, we are going to make a GUI application which is able to mix (cross dissolve/fade) two different video sources and introduce a variety of effects on the mixed video in real-time.

Hand in your solution using the following file containing the different assignment solutions:
Package name: DMA_2_XX.tar.gz (with XX being your groupnr)

1. (DMA_2_1.sh) Write a command line pipeline performing a visualization (autovideosink) and a mix of two video streams after which one effect is applied on the mixed video. After the effect has been added, the video must be displayed (autovideosink) and stored as an H.264/AVC compressed video stream inside an MKV container. Audio can be ignored. Use the following videos and store them in the Downloads folder.

   Video 1: `http://users.datasciencelab.ugent.be/MM/sintel_SD.mp4`
   Video 2: `http://users.datasciencelab.ugent.be/MM/sita_SD.mp4`

2. (DMA_2_2.py) Write a python GUI application in which two video's are visualized and mixed. The transparency (alpha) of each video stream can be set using one horizontal slider on the screen. The output after mixing must be both displayed in the GUI window and stored to disk simultaneously. The path to the video files can be hard coded and video seeking capabilities should not be present because a live stream is supposed to enter the system.

3. (logo.png, DMA_2_3.py) Add some buttons on the GUI such that different effects can be applied to the resulting video (at least 2). One of the effects should be able

to put your channel's logo in the upper left corner of the video. Take care the user interface is convincing as a proof of concept towards the outside "non-engineering" world.

Hint: Effects inherit from BaseTransform, therefore you get the possibility to use the passthrough option for every effect available.

As a source of inspiration, these are the effects gstreamer supports:

1. geometrictransforms
   (a) bulge Adds a protuberance in the center point
   (b) circle Warps the picture into an arc shaped form
   (c) diffuse Diffuses the image by moving its pixels in random directions
   (d) fisheye Simulate a fisheye lens by zooming on the center of the image and compressing the edges
   (e) kaleidoscope Applies 'kaleidoscope' geometric transform to the image
   (f) marble Applies a marbling effect to the image
   (g) mirror Split the image into two halves and reflect one over each other
   (h) perspective Apply a 2D perspective transform
   (i) pinch Applies 'pinch' geometric transform to the image
   (j) rotate Rotates the picture by an arbitrary angle
   (k) sphere Applies 'sphere' geometric transform to the image
   (l) square Distort center part of the image into a square
   (m) stretch Stretch the image in a circle around the center point
   (n) tunnel Light tunnel effect
   (o) twirl Twists the image from the center out
   (p) waterripple Creates a water ripple effect on the image
2. gaudieffects
   (a) burn Burn adjusts the colors in the video signal.
   (b) chromium Chromium breaks the colors of the video signal.
   (c) dilate Dilate copies the brightest pixel around.
   (d) dodge Dodge saturates the colors in the video signal.
   (e) exclusion Exclusion excludes the colors in the video signal.

(f) gaussianblur Perform Gaussian blur/sharpen on a video

(g) solarize Solarize tunable inverse in the video signal.

3. Effect

(a) agingtv AgingTV adds age to video input using scratches and dust

(b) dicetv 'Dices' the screen up into many small squares

(c) edgetv Apply edge detect on video

(d) optv Optical art meets real-time video effect

(e) quarktv Motion dissolver

(f) radioactv motion-enlightment effect

(g) revtv A video waveform monitor for each line of video processed

(h) rippletv RippleTV does ripple mark effect on the video input

(i) shagadelictv Oh behave, ShagedelicTV makes images shagadelic!

(j) streaktv StreakTV makes after images of moving objects

(k) vertigotv A loopback alpha blending effector with rotating and scaling

(l) warptv WarpTV does realtime goo'ing of the video input