Capítulo
8

Programando aplicações multimídia no GStreamer

Guilherme F. Lima, Rodrigo C. M. Santos, Roberto G. de A. Azevedo
Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil

Abstract

This short course is an introduction to GStreamer, one of the main free/open-source frameworks for multimedia processing. We start presenting GStreamer, its architecture and the dataflow programming model, and then adopt a hands-on approach. Starting with an example, a simple video player, we introduce the main concepts of GStreamer's basic C API and implement them over the initial example incrementally, so that at the end of the course we get a complete video player with support for the usual playback operations (start, stop, pause, seek, fast-forward, and rewind). We also discuss sample filters—processing elements that manipulate audio and video samples—and present some of the filters natively available in GStreamer. Moreover, we show how one can extend the framework by creating a plugin with a custom filter that manipulates video samples. The prerequisite for the short course is a basic knowledge of the C programming language. At the end of the course, we expect that participants acquire a general view of GStreamer, and be able to create simple multimedia applications and explore its more advanced features.

Resumo

Este minicurso é uma introdução ao GStreamer, um dos principais frameworks *de código livre/aberto para processamento de dados multimídia. Começamos apresentando o GStreamer, sua arquitetura e modelo de programação baseado em* dataflow, *e em seguida, adotamos uma abordagem prática. Partindo de um exemplo inicial, um* player *de vídeo, introduzimos cada conceito da API C básica do GStreamer e o implementamos sobre o exemplo, incrementando-o, de forma que ao final do minicurso obtemos um* player *de vídeo completo, com suporte às operações usuais de reprodução de vídeo (*start, stop, seek, fast-forward *e* rewind*). Discutimos também filtros—elementos que manipulam amostras de áudio e vídeo—e apresentamos os diversos filtros disponíveis nativamente no GStreamer. Além disso, mostramos como estender o* framework *criando um* plugin *com um filtro simples que manipula amostras de vídeo. O único pré-requisito para o minicurso é um conhecimento básico da linguagem de programação C. Ao final do minicurso, esperamos que os participantes tenham uma visão geral do GStreamer, e estejam aptos a criar aplicações simples e explorar os recursos mais avançados do* framework.

8.1. Introdução

O GStreamer [GStreamer, 2016] é um dos principais *frameworks* de código livre/aberto para processamento de dados multimídia. Além de robusto e flexível, ele suporta diversos formatos de áudio e vídeo, e é amplamente utilizado na indústria e na academia [GStreamer Developers, 2016]. O *framework* em si consiste de um conjunto de bibliotecas C e ferramentas relacionadas. Neste minicurso, apresentamos tanto a parte conceitual quanto a prática do GStreamer.

Na parte conceitual, discutimos o modelo de computação *dataflow* no qual o GStreamer se baseia, e que também é adotado por outros *frameworks* e linguagens multimídia, por exemplo, DirectShow [Chatterjee e Maltz, 1997], Pure Data [Puckette, 2007], CLAM [Amatriain et al., 2008], ChucK [Wang e Cook, 2003], Faust [Orlarey et al., 2009], etc. Nesse modelo, uma aplicação multimídia estrutura-se como um grafo em que os nós são elementos processadores e as arestas representam conexões entre elementos por onde fluem amostras de áudio e vídeo e dados de controle. O modelo de *dataflow* é particularmente interessante para multimídia porque possibilita implementações naturalmente paralelas, modulares e escaláveis.

Na parte prática, apresentamos os principais conceitos da API C [Kernighan e Ritchie, 1988] básica (de baixo nível) do GStreamer 1.8, sua versão estável mais atual, e ilustramos o uso dessa API a partir da construção de um reprodutor (*player*) de vídeo. Apesar de aqui estarmos interessados apenas na reprodução (decodificação e apresentação) de fluxos de mídia, essa mesma API pode ser utilizada para capturar fluxos de áudio e vídeo, codificá-los e transmiti-los na rede. O GStreamer possui uma grande variedade de componentes para tratar cada uma dessas fases de processamento e, portanto, pode ser usado para construir diversos tipos de aplicações multimídia tais como editores de vídeo, transcodificadores, transmissores de fluxos de mídia, *players* de mídia e motores de renderização de linguagens multimídia.

O restante do capítulo está organizado da seguinte forma. Na Seção 8.2 apresentamos uma versão preliminar do exemplo base do minicurso: um *player* de vídeo simples, porém funcional. Na Seção 8.3 discutimos o que está por trás do código aparentemente simples do exemplo anterior e reconstruímos o mesmo exemplo usando a API básica do GStreamer; essa versão reconstruída é o ponto de partida dos incrementos posteriores. Na Seção 8.4 apresentamos alguns do principais filtros de áudio e vídeo disponíveis no GStreamer e discutimos como integrá-los ao exemplo. Na Seção 8.5 adicionamos ao exemplo suporte às operações usuais de controle de reprodução (*start*, *stop*, *pause*, *seek*, *fast-foward* e *rewind*). Na Seção 8.6 apresentamos a arquitetura de *plugins* do GStreamer e mostramos como implementar e integrar ao exemplo um *plugin* simples contendo um elemento que manipula amostras de vídeo. Finalmente, na Seção 8.7 discutimos funcionalidades avançadas do *framework* e listamos algumas referências para estudos posteriores.

Antes de escrever nossa primeira aplicação GStreamer, porém, é preciso entender o modelo de computação *dataflow*, no qual o *framework* se baseia. No restante desta seção apresentamos esse modelo e discutimos como ele é instanciado no GStreamer.

O modelo *dataflow* e sua instanciação no GStreamer

No modelo de computação *dataflow* os dados são processados enquanto "fluem" através de uma rede. Essa rede estrutura-se como um grafo dirigido em que os nós representam elementos de processamento, ou atores, e as arestas representam conexões unidirecionais por onde fluem os dados. Os atores recebem dados através de suas portas de entrada e emitem dados através de suas porta de saída. Um *pipeline* é um *dataflow* em que os dados fluem através das arestas na mesma ordem em que foram produzidos [Kahn e MacQueen, 1977, Lee e Parks, 1995]. O modelo de computação *dataflow*, e em especial o modelo de *pipeline*, é interessante para multimídia porque aproxima a estrutura real do sistema da sua descrição abstrata, idealizada na forma de diagrama de blocos [Yviquel et al., 2015]. Além disso, o modelo de *dataflow* induz implementações flexíveis e eficientes já que ele é naturalmente paralelo, modular e escalável: (1) paralelo porque conceitualmente os atores executam independentemente uns dos outros; (2) modular porque a lógica de processamento está encapsulada nos atores e pode ser reusada em diversos pontos do *dataflow*; e (3) escalável porque a estrutura é naturalmente composicional (um *dataflow* pode ser visto como um ator e vice versa).

A Figura 8.1 apresenta o leiaute de um *pipeline* típico para processamento multimídia. O leiaute nesse caso é uma versão simplificada do *pipeline* de uma aplicação GStreamer que reproduz um vídeo Ogg [Pfeiffer, 2003] . Os nós da figura representam atores e as arestas representam as conexões por onde fluem as amostras de áudio e vídeo e dados de controle. Na terminologia do GStreamer os atores são chamados de "elementos" e as portas de "*pads*". Há dois tipos de *pads*: *sink pads* e *source pads*. As *sink pads* são as portas de entrada através das quais o elemento consome dados, e as *source pads* são as portas de saída através das quais o elemento produz dados. Os elementos são classificados de acordo com o tipo de suas *pads*. Elementos produtores (*sources*) possuem apenas *source pads*. Elementos processadores possuem ambos os tipos, *source pads* e *sink pads*. E, elementos consumidores (*sinks*) possuem apenas *sink pads*. Consequentemente, produtores apenas produzem dados, processadores consomem e produzem dados, e consumidores apenas consomem dados.

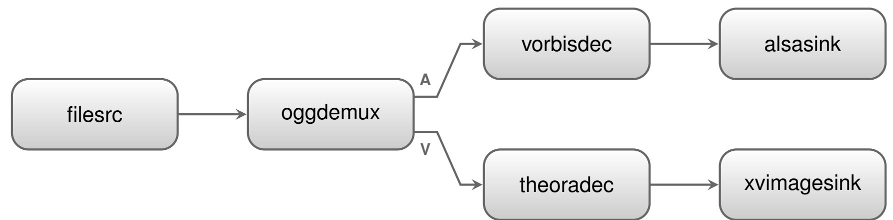


Figura 8.1. Um *pipeline* GStreamer que reproduz um arquivo de vídeo Ogg.

Na Figura 8.1 há um elemento produtor ("filesrc"), três processadores ("oggdemux", "vorbisdec" e "theoradec") e dois consumidores ("alsasink" e "xvimagesink"). O elemento "filesrc" possui uma única *source pad* que está conectada à *sink pad* do elemento subsequente, "oggdemux". Ou seja, os dados produzidos pelo elemento "filesrc" fluem através da sua *source pad* para a *sink pad* do elemento "oggdemux". No diagrama, essa conexão entre as *pads* é representada pela aresta entre os elementos "filesrc" e "oggdemux". Similarmente, as demais arestas denotam conexões entre *source pads* e *sink pads*.

O processo de desenvolvimento de uma aplicação GStreamer com *pipeline* estático (cuja topologia não muda em tempo de execução) é relativamente simples. Basta instanciar os elementos necessários, interconectar suas *pads* e iniciar o *pipeline* resultante. O *pipeline* da Figura 8.1, por exemplo, após iniciado opera da seguinte forma.

1. O elemento "filesrc" lê um arquivo Ogg do disco e escreve o fluxo de bytes resultante na sua *source pad*. Ogg é [Pfeiffer, 2003] um formato para multiplexação de fluxos de áudio, vídeo e texto. Vamos assumir que o arquivo Ogg nesse caso contém apenas dois fluxos multiplexados: um fluxo de áudio (sequência de amostras de áudio) codificado no formato Vorbis [Xiph.Org, 2015]; e um fluxo de vídeo (sequência de quadros de imagem) codificado no formato Theora [Xiph.Org, 2011].

2. O elemento "oggdemux" lê da sua *sink pad* um fluxo de bytes codificado no formato Ogg, demultiplexa-o e escreve os fluxos Vorbis (áudio) e Theora (vídeo) resultantes nas *source pads* correspondentes. Na figura 8.1, a *source pad* que recebe o fluxo de áudio codificado possui o rótulo "A" e a *source pad* que recebe o fluxo de vídeo codificado possui o rótulo "V".

3. O elemento "vorbisdec" lê da sua *sink pad* um fluxo de bytes codificado no formato Vorbis, decodifica-o e escreve o fluxo de áudio PCM resultante na sua *source pad*. Um fluxo de áudio PCM é o que chamamos de "áudio descomprimido" (*raw*), ou seja, é uma sequência de amostras obtidas via *pulse-code modulation* que representa digitalmente o sinal analógico original.

4. O elemento "theoradec" opera de maneira análoga. Ele lê da sua *sink pad* um fluxo de bytes codificado no formato Theora, decodifica-o e escreve o fluxo de vídeo descomprimido (*raw*) resultante na sua *source pad*. Um fluxo de vídeo descomprimido é uma sequência de amostras (quadros) de vídeo em que cada quadro é uma matriz de *pixels* codificados em algum modelo de cor. No modelo de cor RGB (*red-green-blue*), por exemplo, cada *pixel* é codificado como uma sequência de três inteiros que representam tonalidades de vermelho, verde e azul.

5. O elemento "alsasink" lê um fluxo de áudio descomprimido da sua *sink pad* e utiliza a biblioteca ALSA [ALSA, 2016] para reproduzir as amostras do fluxo nos alto-falantes.

6. O elemento "xvimagesink" lê um fluxo de vídeo descomprimido da sua *sink pad* e utiliza a biblioteca X11 [X.Org, 2016] para reproduzir os quadros do fluxo na tela.

A função de um *pipeline*, isto é, o que ele faz ou computa, é o resultado da combinação da função dos seus elementos que, conceitualmente, operam em paralelo. O *pipeline* anterior, portanto, (1) lê um arquivo Ogg, (2) demultiplexa-o, (3–4) decodifica os fluxos de áudio e vídeo resultantes e (5–6) reproduz os fluxos decodificados nos dispositivos de saída correspondentes (alto-falantes e tela). Conceitualmente, tudo isso acontece em paralelo, ou seja, podemos assumir que enquanto os *sinks* "alsasink" e "xvimagesink" estão exibindo amostras, o *source* "filesrc" está lendo bytes do disco, o demultiplexador "oggdemux" está demultiplexando dados Ogg e os decodificadores "vorbisdec" e "theoradec" estão decodificando dados Vorbis e Theora.

A descrição anterior seria suficiente se estivéssemos interessados apenas em exibir as amostras decodificadas o mais rápido possível. Mas esse não é caso. Queremos "tocar" o vídeo original em tempo real, isto é, reproduzi-lo nas mesmas condições em que ele foi gravado (amostrado). Sendo assim, para que o vídeo seja reproduzido corretamente, suas amostras de áudio e vídeo devem ser exibidas na taxa correta. Valores típicos para essas taxas são 44100Hz para amostras de áudio e 30Hz para amostras de vídeo; ou seja, a cada 22,67µs uma nova amostra de áudio deve ser enviada ao dispositivo de saída de áudio, e a cada 33,33ms uma nova amostra de vídeo deve ser enviada ao dispositivo de saída de vídeo.

No GStreamer, em geral, são os elementos *sink* os responsáveis por controlar as taxas de exibição de amostras. Por exemplo, no*pipeline* da Figura 8.1, para manter a taxa de reprodução necessária, os *sinks* "alsasink" e "xvimagesink" armazenam as amostras recebidas numa fila interna e exibem-nas (consomem-nas) apenas no momento adequado. Já os outros elementos operam livres—consomem e produzem dados em taxas arbitrárias. Se durante a execução os elementos que precedem os *sinks* operarem rápido o bastante, as filas dos *sinks* nunca ficarão vazias e todas as amostras serão exibidas no momento correto, mas esse nem sempre é o caso.

Se os elementos que precedem os *sinks* operarem abaixo da taxa de consumo dos *sinks*, pode ser que alguma das filas internas fique vazia e, caso chegue o momento de exibir uma amostra, o *sink* simplesmente não tenha o que exibir, causando interrupções ou saltos na reprodução (amostras "atrasadas" quando chegarem serão descartadas). Há ainda o problema inverso. Se os elementos que precedem os *sink* operarem muito acima da taxa de consumo dos *sinks*, pode ser que a capacidade da fila interna seja excedida e amostras sejam perdidas. Para evitar ambos os problemas, esvaziamento ou estouro das filas, ou mesmo para limitar o uso de CPU, os *sinks* enviam aos elementos que os precedem eventos de QoS (*quality of service*), que indicam o quão atrasada ou adiantada está cada amostra que chega. Os elementos anteriores (produtores e processadores) podem capturar esses eventos e usar a informação de retardo para ajustar a sua taxa de operação.

Dois tipos de dados trafegam através das conexões (arestas) de um *pipeline* GStreamer: segmentos de dados (*buffers*) e eventos (*events*). Os *buffers* carregam segmentos do conteúdo processado entre *pads* (por exemplo, amostras de áudio e vídeo codificadas ou *raw*) e fluem exclusivamente na direção das conexões, ou seja, de *source pads* para *sink pads*. Já os eventos carregam informação de controle; eles também fluem entre conexões, mas podem percorrê-las em ambos os sentidos: fluxo abaixo (*downstream*), de *source pads* para *sink pads*, ou fluxo acima (*upstream*), de *sink pads* para *source pads*. Além de eventos de QoS, elementos podem emitir eventos de EOS (*end-of-stream*) que indicam o fim do fluxo, eventos de *seek* que indicam deslocamentos no fluxo e eventos de *flush* que sinalizam que *caches* internos devem ser descarregados.

Buffers e eventos percorrem as conexões em paralelo. Ou seja, conceitualmente cada conexão entre as *pads* dos elementos pode ser entendida como consistindo de dois canais, um canal unidirecional para *buffers* e outro canal bidirecional para eventos. A Figura 8.2 ilustra a estrutura conceitual de uma conexão entre *pads*. Na figura,"B" é o canal de *buffers* e "E" é o canal de eventos.

Figura 8.2. Estrutura conceitual de uma conexão entre *pads* no GStreamer.

A discussão do modelo *dataflow* e da sua instanciação no GStreamer se encerra aqui. Na maior parte do tempo, programar no GStreamer consiste em montar *pipelines* e controlar seu funcionamento. Até agora discutimos de maneira abstrata os conceitos envolvidos nessa montagem e controle. A partir da próxima seção, veremos como utilizar esses conceitos na prática.

8.2. Olá mundo: Tocando um vídeo

Nosso objetivo nesta seção é usar o GStreamer para tocar um vídeo. Para tal, poderíamos implementar o *pipeline* da Figura 8.1, mas há uma maneira mais simples: basta usar o elemento “playbin”. A Listagem 8.1 apresenta um programa C que faz exatamente isso.

```
1  #include <glib.h>
2  #include <gst/gst.h>
3
4  int main (int argc, char *argv[])
5  {
6    GstElement *playbin;
7    char *uri;
8    GstStateChangeReturn ret;
9    GstBus *bus;
10   GstMessage *msg;
11
12   gst_init (&argc, &argv);
13
14   playbin = gst_element_factory_make ("playbin", "hello");
15   g_assert_nonnull (playbin);
16
17   uri = gst_filename_to_uri ("bunny.ogg", NULL);
18   g_assert_nonnull (uri);
19   g_object_set (G_OBJECT (playbin), "uri", uri, NULL);
20   g_free (uri);
21
22   ret = gst_element_set_state (playbin, GST_STATE_PLAYING);
23   g_assert (ret != GST_STATE_CHANGE_FAILURE);
24
25   bus = gst_element_get_bus (playbin);
26   msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
27                                     GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
28   gst_message_unref (msg);
29   gst_object_unref (bus);
30   gst_element_set_state (playbin, GST_STATE_NULL);
31   gst_object_unref (playbin);
32
33   return 0;
34 }
```

Listagem 8.1. Tocando um vídeo no GStreamer usando o elemento “playbin”.

Vejamos o propósito de cada linha da Listagem 8.1. A linha 1 inclui as declarações da GLib [GLib, 2016a], a biblioteca de portabilidade do projeto GNOME [GNOME, 2016], e a linha 2 inclui as declarações do GStreamer—o GStreamer depende do *framework* GObject [GLib, 2016b] da GLib para programação orientada a objetos em C. Na listagem, as chamadas com prefixo “`g_`” ou “`G_`” e referem-se à funções e macros da GLib, e as chamadas com prefixo “`gst_`” ou “`GST_`” e as declarações com prefixo “`Gst`” referem-se à funções, macros e tipos do GStreamer.

A chamada `gst_init`, linha 12, inicializa o GStreamer e trata os argumentos com prefixo "- -gst-" em `argv`.

A próxima chamada, linha 14, cria um elemento "playbin" que é também o *pipeline* da aplicação. No GStreamer, o *pipeline* é, ele próprio, um elemento—que contém outros elementos mas não possui *pads*. Tais elementos contêiner são chamados de "*bins*". A função `gst_element_factory_make` aloca e retorna um novo elemento (`GstElement`) do tipo especificado. O primeiro parâmetro da função é o nome da fábrica do tipo e o segundo parâmetro (opcional) é o nome a ser atribuído ao elemento retornado—esse nome pode ser usado mais tarde para obter uma referência para o elemento a partir do *pipeline*. No exemplo da Listagem 8.1, a chamada cria e retorna um elemento do tipo "playbin" chamado "hello". Um elemento "playbin" nada mais é do que um *pipeline* que se autoconfigura. Dada uma URI, assim que o "playbin" é iniciado ele determina o tipo do conteúdo da URI e constrói um *pipeline* apropriado para reproduzi-la.

A chamada `g_assert_nonnull`, linha 15, é uma asserção que garante que a chamada anterior funcionou corretamente, isto é, retornou um elemento válido (não nulo).

A chamada `gst_filename_to_uri`, linha 17, constrói uma URI a partir do caminho de um arquivo. Nesse caso, "bunny.ogg" é caminho do arquivo de vídeo Ogg que queremos tocar. O vídeo em si é o curta de animação "Big Buck Bunny" [Blender, 2008] produzido pelo Blender Institute e licenciado via Creative Commons.

A chamada `g_object_set`, linha 19, atribui a URI criada anteriormente à propriedade "uri" do elemento "playbin". Note que `g_object_set` é uma função do GObject. Todo elemento (`GstElement`) é também um objeto (`GObject`), e todo objeto possui propriedades cujos valores podem ser obtidos via `g_object_get` e alterados via `g_object_set`.

A chamada `g_free`, linha 20, libera a *string* alocada na linha 16. Nesse ponto, a *string* já não é mais necessária pois foi copiada pela chamada `g_object_set` anterior.

A chamada `gst_element_set_state`, linha 22, inicia o *pipeline* da aplicação, isto é, transiciona o elemento "playbin" do seu estado inicial *null* (`GST_STATE_NULL`) para o estado *playing* (`GST_STATE_PLAYING`). Na Seção 8.3, vamos discutir em detalhes as consequências internas dessa transição. Por enquanto, basta dizer que nesse ponto o elemento "playbin" (1) inspeciona o conteúdo apontado pela URI configurada, (2) instancia e interconecta os elementos necessários para reproduzir esse conteúdo, (3) inicia a sua reprodução numa *thread* separada e (4) devolve o controle à *thread* principal da aplicação.

A chamada `g_assert`, linha 23, é uma asserção que garante que a requisição de transição de estado anterior foi bem sucedida. Após essa linha, podemos assumir que o vídeo está tocando e que aplicação possui pelo menos duas *threads*: a *thread* principal, que está prestes a executar a linha 23, e uma ou mais *threads* secundárias, que operam o *pipeline*. A *thread* principal é chamada de "*thread* da aplicação" e as *threads* secundárias são chamadas de "*streaming threads*". As *streaming threads* se comunicam com a *thread* da aplicação através de mensagens assíncronas postadas num barramento (*bus*) associado ao *pipeline*. Essas mensagens informam a *thread* da aplicação sobre acontecimentos internos do *pipeline*, por exemplo, mudança de estado de elementos, fim do fluxo, erros, *warnings*, etc., e permitem que a aplicação reaja da maneira apropriada.

A chamada gst_element_get_bus, linha 25, obtém uma referência para o barramento de mensagens (*bus*) do *pipeline* do “playbin”.

A próxima chamada, linha 26, bloqueia a *thread* da aplicação até que uma mensagem de erro ou de EOS (*end-of-stream*, “fim do fluxo”) seja postada no barramento. Ou seja, a *thread* da aplicação aguarda nessa chamada até que um erro aconteça ou até que o vídeo seja reproduzido completamente. A função gst_bus_timed_pop_filtered recebe o *bus*, o tempo máximo de espera e a máscara dos tipos das mensagens a serem aguardadas, e bloqueia até que o tempo máximo seja atingido ou até que uma mensagem de um dos tipos esperados seja postada no *bus*. A função retorna NULL caso o tempo máximo seja atingido ou retorna uma referência para a mensagem recebida. Na Listagem 8.1, estamos aguardando por um tempo ilimitado (GST_TIME_CLOCK_NONE) uma mensagem de erro (GST_MESSAGE_ERROR) ou uma mensagem de EOS (GST_MESSAGE_EOS) que após a chamada é armazenada na variável msg.

A chamada gst_message_unref, linha 27, libera a mensagem retornada na chamada da linha anterior. Nesse ponto, ou houve um erro ou a reprodução chegou ao fim.

A chamada gst_object_unref, linha 28, libera a referência para o *bus* do “playbin”, obtida na linha 25. A função gst_object_unref é um pseudônimo (*alias*) para a função g_object_unref, que libera uma referência para um GObject. No GStreamer, a maioria dos tipos que são “objeto” herdam deGstObject que, por sua vez, herda de GObject.

A chamada gst_element_set_state, linha 30, para o *pipeline* da aplicação (caso ele ainda não esteja parado) e libera os recursos utilizados durante o seu processamento, isto é, transiciona o elemento “playbin” de volta para o estado inicial *null*.

Finalmente, a chamada gst_object_unref, linha 31, libera o próprio elemento “playbin”, alocado na linha 14.

Se tudo correr bem, ao executar o programa “Olá mundo” anterior uma nova janela aparece na tela e o vídeo é reproduzido do inicio ao fim. A Figura 8.3 apresenta um quadro do vídeo em questão. Para rodar o exemplo, porém, primeiro é preciso compilá-lo.

Figura 8.3. Quadro do vídeo *Big Buck Bunny* [Blender, 2008].

Compilando o programa "Olá mundo" no GNU/Linux

Para compilar o programa da Listagem 8.1 precisamos informar ao compilador C o caminho do diretório contendo os cabeçalhos da GLib e do GStreamer. Além disso, precisamos informar ao *linker* o caminho e o nome das bibliotecas dinâmicas correspondentes. No GNU/Linux, a maneira mais fácil de se fazer isso é através da ferramenta *pkg-config*. Por exemplo:

```
$ cc hello.c -o hello `pkg-config --cflags --libs glib-2.0\
        gstreamer-1.0`
```

Esse comando compila e *link*-edita o programa da Listagem 8.1 ("hello.c") e gera um executável chamado "hello". As opções "- -cflags" e "- -libs" do comando *pkg-config* instruem-no a emitir as *flags* de compilação e *link*-edição apropriadas para os módulos listados, no caso, "glib-2.0" e "gstreamer-1.0". Observe que a chamada do comando *pkg-config* aparece entre crases. Ou seja, antes de avaliar o comando mais externo, *cc*, que é a chamada do compilador C, o interpretador de comandos (*shell*) executa o comando *pkg-config* e substitui o texto entre crases pelo resultado dessa execução. Por exemplo, no nosso ambiente, o comando anterior avalia para:

```
$ cc hello.c -o hello\
      -pthread\
      -I/usr/include/gstreamer-1.0\
      -I/usr/lib/gstreamer-1.0/include\
      -I/usr/include/glib-2.0\
      -I/usr/lib/glib-2.0/include\
      -lgstreamer-1.0 -lgobject-2.0 -lglib-2.0
```

Uma forma de tornar o processo de compilação menos trabalhoso é escrever um *makefile* com a descrição da sequência de comandos que constrói o programa. A Listagem 8.2 apresenta o *makefile* (arquivo texto chamado "Makefile") que usamos para construir os exemplos do minicurso. Na listagem, a variável `PROGRAMS`, linha 1, contém o nome dos programas a serem construídos. A variável `MODULES`, linha 2, contém os nomes dos módulos do *pkg-config* necessários. E as variáveis `CFLAGS` e `LDFLAGS`, linhas 3 e 4, contém as *flags* de compilação e *link*-edição correspondentes. O restante do arquivo, linhas 5–8, define os alvos "all" e "clean". (Na linha 7, o primeiro caractere é um TAB.) Uma vez escrito o *makefile*, para construir o programa basta executar o comando "`make`" (ou "`make all`") e para remover os arquivos gerados pelo compilador basta executar "`make clean`". (Veja [Gough, 2005, Mecklenburg, 2005] para mais informações sobre como compilar programas e escrever *makefiles* no GNU/Linux.)

```
1  PROGRAMS= hello
2  MODULES= glib-2.0 gstreamer-1.0
3  CFLAGS= -g -Wall -Wextra `pkg-config --cflags $(MODULES)`
4  LDFLAGS= `pkg-config --libs $(MODULES)`
5  all: $(PROGRAMS)
6  clean:
7          -rm -f $(PROGRAMS)
8  .PHONY: all clean
```

Listagem 8.2. *Makefile* que constrói o programa "Olá mundo".

8.3. Tocando áudio e vídeo a partir de elementos básicos

Na Seção 8.2 utilizamos o elemento "playbin" para tocar um vídeo Ogg no GStreamer. Nesta seção, nosso objetivo é construir a mesma aplicação, mas dessa vez sem usar o elemento "playbin". Usando apenas elementos básicos (*sources*, demultiplexadores, decodificadores e *sinks*) vamos construir um *pipeline* semelhante ao da Figura 8.1, Seção 8.1, que lê um arquivo Ogg do disco, demultiplexa-o, e decodifica e reproduz os fluxos de áudio e vídeo resultantes. Antes disso, porém, vejamos um exemplo mais simples.

Tocando um arquivo de áudio MP3

A Listagem 8.3 apresenta um programa que reproduz um arquivo de áudio MP3 [ISO/IEC, 1993] do disco. Para tal, o programa monta um *pipeline* com três elementos: "filesrc", "mad" e "alsasink". Como vimos, no GStreamer o próprio *pipeline* é um elemento—um elemento sem *pads* que contém outros elementos. Na listagem, a chamada da linha 15 aloca o elemento *pipeline*, que inicialmente está vazio, e as chamadas das linhas 16–18, alocam os elementos "filesrc", "mad" e "alsasink". (As chamadas anteriores, linhas 11–13, inicializam o GStreamer e testam se o programa foi chamado corretamente, isto é, se o caminho para o arquivo MP3 está em `argv[1]`.)

```
1  #include <glib.h>
2  #include <gst/gst.h>
3
4  int main (int argc, char *argv[])
5  {
6    GstElement *pipeline, *src, *dec, *sink;
7    GstStateChangeReturn ret;
8    GstBus *bus;
9    GstMessage *msg;
10
11   gst_init (&argc, &argv);
12   if (argc != 2)
13     return (g_printerr ("usage: %s FILE\n", g_get_prgname ()), 1);
14
15   pipeline = gst_element_factory_make ("pipeline", "mp3");
16   src = gst_element_factory_make ("filesrc", "src");
17   dec = gst_element_factory_make ("mad", "dec");
18   sink = gst_element_factory_make ("alsasink", "sink");
19   g_assert (pipeline && src && dec && sink);
20
21   gst_bin_add_many (GST_BIN (pipeline), src, dec, sink, NULL);
22   gst_element_link_many (src, dec, sink, NULL);
23   g_object_set (G_OBJECT (src), "location", argv[1], NULL);
24
25   ret = gst_element_set_state (pipeline, GST_STATE_PLAYING);
26   g_assert (ret != GST_STATE_CHANGE_FAILURE);
27
28   bus = gst_element_get_bus (pipeline);
29   msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
30                                     GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
31   gst_message_unref (msg);
32   gst_object_unref (bus);
33   gst_element_set_state (pipeline, GST_STATE_NULL);
34   gst_object_unref (pipeline);
35
36   return 0;
37 }
```

Listagem 8.3. Tocando um arquivo de áudio MP3.

Na Seção 8.1, discutimos a operação dos elementos “filesrc” e “alsasink”. O primeiro lê um arquivo do disco e gera um fluxo de bytes correspondente, e o segundo lê um fluxo de amostras de áudio PCM e as reproduz no dispositivo de saída de áudio. Aqui a novidade é o elemento processador, mais precisamente, o decodificador “mad”. Ele lê da sua *source pad* um fluxo de bytes no formato MP3, decodifica-o via MAD [Underbit, 2016] (biblioteca para decodificação de áudio MP3) e escreve na sua *sink pad* o fluxo de áudio PCM resultante. Apesar de os três elementos, “filesrc”, “alsasink” e “mad”, fazerem parte do GStreamer, na prática, eles pertencem a pacotes diferentes, distribuídos separadamente.

A distribuição oficial do GStreamer possui cinco pacotes principais: (1) *gstreamer*, contendo o núcleo do *framework*; (2) *gst-plugins-base*, contendo apenas os elementos básicos; (3) *gst-plugins-good*, contendo elementos cuja implementação é considerada de boa qualidade e cuja licença é LGPL (*Lesser GNU General Public License*); (4) *gst-plugins-ugly*, contendo elementos cuja implementação é também considerada de boa qualidade mas que têm licenças problemáticas; e (5) *gst-plugins-bad*, contendo elementos de qualidade inferior.[1] O único pacote absolutamente necessário é o *gstreamer*. Os demais incrementam-no instalando bibliotecas e *plugins* contendo elementos especializados. O elemento “filesrc”, por exemplo, está no *plugin* “coreelements” do pacote *gstreamer*. Já o elemento “alsasink” está no *plugin* “alsa” do pacote *gst-plugins-base*, e o elemento “mad” está no *plugin* “mad” do pacote *gst-plugins-ugly*. Você pode descobrir os elementos e *plugins* disponíveis na sua instalação através do comando “`gst-inspect`”—voltaremos a falar desse comando no final da seção.

De volta à Listagem 8.3, a chamada `g_assert` na linha 19 assegura que as chamadas `gst_element_factory_make` anteriores foram bem sucedidas. Isto é, testa se os elementos “pipeline”, “filesrc”, “mad” e “alsasink” foram de fato alocados. Um erro comum é tentar instanciar um elemento de um *plugin* que não está instalado. Na listagem, se isso acontecer a chamada `gst_element_factory_make` retornará `NULL` e a asserção falhará, abortando o programa.

Continuando, a chamada da `gst_bin_add_many`, linha 21, adiciona três elementos ao *pipeline*: “filesrc”, “mad” e “alsasink”. Essa função recebe um *bin* (`GstBin`) e uma sequência de elementos (`GstElement`) terminada por `NULL`, e adiciona os elementos da sequência ao *bin*. Note, que antes da chamada precisamos usar a macro `GST_BIN` para converter a variável `pipeline` para o tipo da sua superclasse `GstBin`.

A chamada `gst_element_link_many`, linha 22, conecta os elementos apontados pelas variáveis `src`, `filter` e `sink` em série. Isto é, conecta a *source pad* do “filesrc” à *sink pad* do “mad”, e conecta a *source pad* do “mad” à *sink pad* do “videosink”. A função `gst_element_link_many` recebe uma sequência de elementos terminada por `NULL` e conecta-os em série apenas se suas *pads* são compatíveis e se todos possuem o mesmo elemento pai (isto é, se todos foram adicionados ao mesmo *bin*). A Figura 8.4 apresenta a estrutura interna do elemento “pipeline”, variável `pipeline`, após a chamada da linha 22.

A chamada `g_object_set`, linha 23, atribui o caminho do arquivo MP3 a ser tocado (primeiro argumento do programa) à propriedade “location” do elemento “filesrc”. Essa propriedade indica ao elemento o arquivo que servirá de fonte de bytes.

[1] Essa terminologia é inspirada clássico de faroeste “The Good, the Bad and the Ugly” (1966).

Figura 8.4. *Pipeline* que reproduz um arquivo de áudio MP3.

A chamada `gst_element_set_state`, linha 25, transiciona o "pipeline" do seu estado atual (*null*) para o estado *playing* o que, como vimos na Seção 8.2, faz com que os seus filhos "filesrc", "mad" e "alsasink" comecem a operar. No Gstreamer, todo elemento, incluindo o *pipeline*, possui um estado que pode ser nulo (*null*, identificado pela constante simbólica `GST_STATE_NULL`), pronto (*ready*, `GST_STATE_READY`), pausado (*paused*, `GST_STATE_PAUSED`) ou tocando (*playing*, `GST_STATE_PLAYING`). No estado inicial, *null*, o elemento não possui recursos alocados. No estado *ready*, o elemento aloca recursos globais que não dependem do conteúdo a ser processado. No estado *paused*, o elemento aloca recursos que dependem do conteúdo a ser processado e se prepara para processá-lo. Finalmente, no estado *playing*, o elemento inicia o processamento.

Para chegar do estado *null* (inicial) ao estado *playing* (tocando) todo elemento tem que passar primeiro pelo estados *ready* e *paused*, nessa ordem. De forma análoga, para sair do estado *playing* e voltar ao estado inicial *null*, o elemento tem que passar pelos estados *paused* e *ready*. A Figura 8.5 apresenta a máquina de estados de um elemento GStreamer. Mudanças de estado do *pipeline*, na verdade de *bins* em geral, são propagadas nos elementos filhos, e a direção da propagação é sempre dos *sinks* para os *sources*—dessa forma, dados não são gerados antes que os elementos posteriores no *pipeline* estejam prontos para recebê-los. Por exemplo, na Listagem 8.3, se bem sucedida, a chamada `gst_element_set_state` (linha 25) implica na seguinte sequência transições (sempre dos *sinks* para os *sources*): (1) os elementos filhos "alsasink", "mad" e "filesrc" transicionam de *null* para *ready* e, em seguida, o *pipeline* transiciona para *ready*; (2) os filhos transicionam de *ready* para *paused* e, em seguida, o *pipeline* transiciona para *paused*; e (3) os filhos transicionam de *paused* para *playing* e, finalmente, o *pipeline* transiciona para *playing*. A sequência de transições pode ser realizada sincronamente ou assincronamente. Se a sequência for realizada sincronamente, a *thread* da aplicação bloqueia até a que todas as transições sejam realizadas. Caso contrário, se a sequência for realizada assincronamente, a chamada da linha 25 retorna imediatamente e as transições são realizadas em paralelo (*background*). Em ambos os casos, podemos assumir que nesse ponto a *thread* da aplicação está prestes a executar a asserção da linha 26, que aborta o programa em caso de falha da chamada anterior, e que as *streaming threads* já começaram a operar o *pipeline*, ou seja, os elementos "filesrc", "mad" e "alsasink" já estão produzindo, processando e consumindo dados.

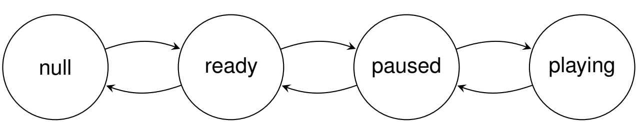


Figura 8.5. Máquina de estados de um elemento (`GstElement`).

A chamada seguinte, linha 28, obtém uma referência para o barramento do *pipeline* e a próxima, linha 29, bloqueia a *thread* da aplicação até que um erro aconteça ou até que o arquivo de áudio seja reproduzido completamente (EOS).

Assim que a chamada `gst_bus_timed_pop_filtered` retorna, a *thread* da aplicação libera as referências do *bus* e da mensagem retornada (linhas 31–32), transiciona o *pipeline* (linha 33) de volta para o estado *null* e libera a sua referência (linha 34). Nesse caso, a chamada `gst_element_set_state` (linha 33) também propaga a transição para os elementos filhos, só que agora as transições são no sentido contrário—de *playing* para *paused*, *ready* e *null*. Finalmente, observe que a chamada `gst_object_unref` (linha 34) quando aplicada a um *bin* libera não apenas uma referência para o *bin* mas também libera uma referência para cada elemento filho, ou seja, nesse ponto os elementos alocados nas linhas 15–28 são liberados.

De volta ao problema inicial

Agora que vimos como alocar, adicionar e interconectar elementos num *pipeline* podemos voltar ao nosso problema inicial: construir um *pipeline* similar ao da Seção 8.1 que usa apenas elementos básicos para tocar um vídeo Ogg. Há uma diferença importante entre esse *pipeline* Ogg (Figura 8.1) e o *pipeline* MP3 (Figura 8.4) que construímos anteriormente. Enquanto no *pipeline* MP3 o número total de *pads* é fixo (conhecido em tempo de compilação) no *pipeline* Ogg esse número é variável. Em particular, no *pipeline* Ogg o número de *sink pads* do elemento “oggdemux” depende do número de subfluxos multiplexados no fluxo Ogg de entrada. Por exemplo, se o fluxo Ogg possuir apenas um subfluxo Vorbis, o elemento “oggdemux” terá uma única *sink pad* que deverá ser conectada à *source pad* do elemento “vorbisdec”. No entanto, se o fluxo Ogg possuir dois subfluxos, Vorbis e Theora, o elemento “oggdemux” terá duas *sink pads*, uma para o subfluxo Vorbis que deverá ser conectada ao elemento “vorbisdec”, e outra para o subfluxo Theora que deverá ser conectada ao elemento “theoradec”. Logo, o número de subfluxos multiplexados no Ogg determina o número de *sink pads* no demultiplexador. *Pads* desse tipo, criadas sob demanda pelo elemento, são chamadas de *sometimes pads*.

No GStreamer, toda *pad* é instanciada de acordo com um *template* que indica a sua direção, capacidade e disponibilidade. A direção (*source* ou *sink*) determina se a *pad* produz ou consome *buffers*. A capacidade, ou *caps*, determina os tipos de *buffers* que podem atravessá-la. E a disponibilidade (*always*, *sometimes* ou *request*) determina o momento em que a *pad* é criada: (1) *always pads* são criadas assim que o elemento é criado; (2) *sometimes pads* são criadas pelo próprio elemento sob condições específicas que normalmente envolvem o conteúdo processado; e (3) *request pads* são criada apenas quando requisitadas pela aplicação.

No *pipeline* MP3 (Figura 8.4) todas as *pads* têm disponibilidade *always*. Isso significa que podemos construir o *pipeline* inteiro estaticamente, conectando todos os seus elementos antes de iniciá-lo. Já no *pipeline* Ogg (Figura 8.1) essa abordagem não é possível. As *sink pads* do elemento “oggdemux” possuem disponibilidade *sometimes* e só serão criadas quando o elemento inspecionar o fluxo de entrada (transicionar para o estado *paused*). Nesse caso, precisamos usar uma abordagem incremental com dois passos. Primeiro, precisamos montar e iniciar um *pipeline* contendo apenas os elementos “filesrc” e

"oggdemux" conectados. Em seguida, precisamos esperar o elemento "oggdemux" notificar a criação de cada *sink pad* para então adicionar e conectar os elementos restantes de acordo com as capacidades da *pad* criada. Esse tipo de notificação assíncrona disparada por elementos é chamada de sinal. Todo sinal possui um nome ("pad-added", nesse caso) e pode ser capturado via um mecanismo de registro de *callbacks*. Ou seja, para capturar um sinal a aplicação registra no elemento uma função *callback* que é chamada sempre que o sinal é disparado.

A Listagem 8.4 apresenta um programa que utiliza a abordagem incremental anterior para montar um *pipeline* que reproduz um arquivo Ogg. A estrutura final desse *pipeline*, apresentada na Figura 8.6, é ligeiramente diferente daquela apresentada na Seção 8.1—a Figura 8.1 é na verdade uma versão simplificada (não funcional) da Figura 8.6. A diferença aqui está nos elementos adicionais "audioconvert", que aparece entre os elementos "vorbisdec" e "alsasink", e "queue", que aparece entre os elementos "oggdemux" e "theoradec".

O elemento "audioconvert" é necessário porque os elementos "vorbisdec" e "alsasink" não podem ser conectados diretamente—as *caps* da *source pad* do "vorbisdec" não são compatíveis com as *caps* do *sink pad* do "alsasink", ou seja, interseção das *caps* dessas *pads* é vazia (podemos entender as *caps* de uma *pad* como o conjunto dos possíveis formatos dos *buffers* que podem atravessá-la). O elemento "audioconvert" converte as amostras produzidas pelo elemento "vorbisdec" para um formato que "alsasink" aceita. Já o elemento "queue" entre os elementos "oggdemux" e "theoradec" é necessário porque esses elementos precisam operar em *streaming threads* separadas. O elemento "queue" é uma fila de *buffers*; quando inserido entre dois elementos, produtor e consumidor, o "queue" força a criação de duas *streaming threads*, uma que opera o produtor e outra que opera o consumidor, e ao mesmo tempo, age como elemento de sincronização dessas *threads*. Isto é, o "queue" bloqueia a *thread* produtora caso a fila fique cheia e bloqueia a *thread* consumidora caso a fila esvazie. Sem o elemento *queue* o *pipeline* da Figura 8.6 não funciona pois não completa a transição do estado *paused* para *playing*.

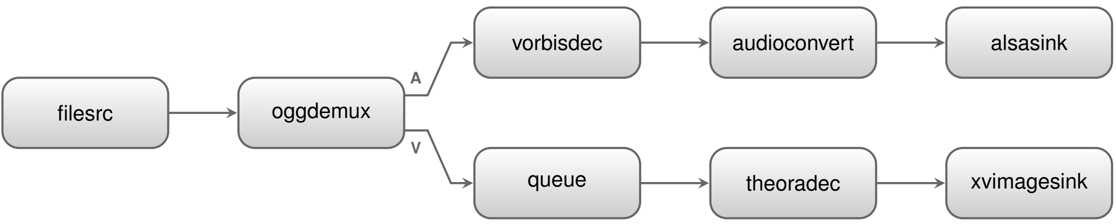


Figura 8.6. Estrutura final do *pipeline* da Listagem 8.4.

De volta à Listagem 8.4, vejamos os trechos relevantes do programa. Na função `main`, as chamadas das linhas 67–69 alocam os elementos "pipeline", "filesrc" e "oggdemux", e as chamadas seguintes, linhas 72–73, adicionam os dois últimos ao *pipeline* e interconecta-os. A chamada da linha 74 atribui o caminho do arquivo Ogg à propriedade "location" do elemento "filesrc". Nesse ponto, portanto, o *pipeline* possui apenas os elementos "filesrc" e "oggdemux", ambos estão conectados e o elemento "filesrc" está configurado com o caminho do arquivo a ser tocado.

A chamada `g_signal_connect`, linha 75, registra a *callback* `pad_added_cb` (linhas 4–54) como tratadora do sinal "pad-added" do elemento "oggdemux", ou seja, a função `pad_added_cb` será chamada sempre que o "oggdemux" criar uma *sometimes pad*.

A chamadas seguintes (linhas 77–85) fazem o mesmo que fizeram nos exemplos anteriores: transicionam o *pipeline* do estado *null* para *playing* (linha 77), bloqueiam a *thread* da aplicação esperando um erro ou o fim da reprodução (linhas 79–80) e, quando o controle retorna à `main`, transicionam o *pipeline* de volta ao estado *null* (linha 84) e liberam os elementos alocados (linha 85). A diferença aqui está no que acontece na transição intermediária do *pipeline* do estado *ready* para o estado *paused*, disparada pela chamada `gst_element_set_state`, linha 77.

Quando o *pipeline* da Listagem 8.4 transiciona para o estado *paused* seus elementos (nesse ponto, apenas "filesrc" e "oggdemux") já estão pausados, ou seja, já carregaram os dados iniciais do fluxo Ogg e estão prontos para começar a operar. Nesse momento, para cada subfluxo multiplexado no fluxo Ogg, o elemento "oggdemux" dispara um sinal "pad-added" que resulta numa chamada à função `pad_added_cb` (linhas 4–54). Por exemplo, se o fluxo Ogg possuir dois subfluxos, um Vorbis e um Theora, a função `pad_added_cb` será chamada uma vez para cada subfluxo. Em ambas as chamadas, o parâmetro `demux` contém o elemento que disparou o sinal ("oggdemux"), `srcpad` contém a *sometimes pad* criada pelo elemento, e `data` contém o dado adicional (*user data*) passado à função `g_signal_connect` (linha 75) no momento do registro da *callback* (no caso, `NULL`).

Na função `pad_added_cb`, a chamada da linha 13 obtém o *pipeline*, a chamada da linha 14 obtém as *caps* (`GstCaps`) associadas à *sometimes pad* criada (parâmetro `srcpad`) e a chamada da linha 15 obtém o tipo (*mime type*) do fluxo que escoará através da *pad*. Nesse ponto, há três possibilidades:

1. Se o tipo é Vorbis ("audio/x-vorbis") e uma *pad* Vorbis ainda não foi encontrada, a *callback* aloca os elementos "vorbisdec", "audioconvert" e "alsasink" (linhas 18–20), adiciona-os ao *pipeline* (linha 22), transiciona os elementos para o mesmo estado do *pipeline*, no caso *paused*, (linhas 23–25), conecta-os em série (linha 26)[2], obtém a *sink pad* do primeiro elemento da série, "vorbisdec", (linha 27), e marca a *flag* `has_vorbis_pad` (linha 28) para indicar que uma *pad* Vorbis foi encontrada.

2. Se o tipo é Theora ("video/x-theora") e uma *pad* Theora ainda não foi encontrada, a *callback* procede de forma análoga, mas agora para os elementos "queue", "theoradec" e "xvimagesink", ou seja, aloca-os, adiciona-os ao *pipeline*, transiciona-os para o mesmo estado do *pipeline*, conecta-os em série, obtém a *source pad* do elemento "queue" e marca a *flag* `has_theora_pad`.

3. Caso contrário, isto é, se o tipo não é Vorbis nem Theora ou se a *pad* em questão não é a primeira Vorbis ou Theora encontrada, a *pad* é simplesmente ignorada (linha 46).

Se a *pad* não for ignorada (casos 1 e 2 anteriores), na linha 48 a variável `sinkpad` conterá a *sink pad* do primeiro elemento da série correspondente à *sometimes pad* criada (parâmetro

[2]Observe que só é possível conectar elementos que estão no mesmo estado. Por isso, antes de conectar os elementos, a chamada `pad_added_cb` primeiro os transiciona para o mesmo estado do *pipeline*.

`srcpad`), ou seja, a *sink pad* do elemento “vorbisdec” se `srcpad` é do tipo Vorbis ou do elemento “queue” se `srcpad` é do tipo Theora. Finalmente, a chamada `gst_pad_link` (linha 48) conecta as *pads* `srcpad` e `sinkpad`, e as chamadas seguintes (linhas 50–53) liberam as referências utilizadas.

```
1  #include <glib.h>
2  #include <gst/gst.h>
3
4  static void pad_added_cb (GstElement *demux, GstPad *srcpad, gpointer data)
5  {
6    GstElement *pipeline;
7    GstCaps *caps;
8    const char *name;
9    GstPad *sinkpad;
10   GstPadLinkReturn ret;
11   static gboolean has_vorbis_pad = FALSE, has_theora_pad = FALSE;
12
13   pipeline = GST_ELEMENT (gst_element_get_parent (demux));
14   caps = gst_pad_query_caps (srcpad, NULL);
15   name = gst_structure_get_name (gst_caps_get_structure (caps, 0));
16   if (g_str_equal (name, "audio/x-vorbis") && !has_vorbis_pad)
17     {
18       GstElement *dec = gst_element_factory_make ("vorbisdec", NULL);
19       GstElement *conv = gst_element_factory_make ("audioconvert", NULL);
20       GstElement *sink = gst_element_factory_make ("alsasink", NULL);
21       g_assert (dec && conv && sink);
22       gst_bin_add_many (GST_BIN (pipeline), dec, conv, sink, NULL);
23       gst_element_sync_state_with_parent (dec);
24       gst_element_sync_state_with_parent (conv);
25       gst_element_sync_state_with_parent (sink);
26       gst_element_link_many (dec, conv, sink, NULL);
27       sinkpad = gst_element_get_static_pad (dec, "sink");
28       has_vorbis_pad = TRUE;
29     }
30   else if (g_str_equal (name, "video/x-theora") && !has_theora_pad)
31     {
32       GstElement *queue = gst_element_factory_make ("queue", NULL);
33       GstElement *dec = gst_element_factory_make ("theoradec", NULL);
34       GstElement *sink = gst_element_factory_make ("xvimagesink", NULL);
35       g_assert (queue && dec && sink);
36       gst_bin_add_many (GST_BIN (pipeline), queue, dec, sink, NULL);
37       gst_element_sync_state_with_parent (queue);
38       gst_element_sync_state_with_parent (dec);
39       gst_element_sync_state_with_parent (sink);
40       gst_element_link_many (queue, dec, sink, NULL);
41       sinkpad = gst_element_get_static_pad (queue, "sink");
42       has_theora_pad = TRUE;
43     }
44   else
45     {
46       goto done;
47     }
48   ret = gst_pad_link (srcpad, sinkpad);
49   g_assert (GST_PAD_LINK_SUCCESSFUL (ret));
50   gst_object_unref (sinkpad);
51  done:
52   gst_caps_unref (caps);
53   gst_object_unref (pipeline);
54 }
55
56 int main (int argc, char *argv[])
57 {
58   GstElement *pipeline, *src, *demux;
59   GstStateChangeReturn ret;
60   GstBus *bus;
61   GstMessage *msg;
62
63   gst_init (&argc, &argv);
```

```
64    if (argc != 2)
65      return (g_printerr ("usage: %s FILE\n", g_get_prgname ()), 1);
66
67    pipeline = gst_element_factory_make ("pipeline", "ogg");
68    src = gst_element_factory_make ("filesrc", "src");
69    demux = gst_element_factory_make ("oggdemux", "demux");
70    g_assert (pipeline && src && demux);
71
72    gst_bin_add_many (GST_BIN (pipeline), src, demux, NULL);
73    gst_element_link_many (src, demux, NULL);
74    g_object_set (G_OBJECT (src), "location", argv[1], NULL);
75    g_signal_connect (demux, "pad-added", G_CALLBACK (pad_added_cb), NULL);
76
77    ret = gst_element_set_state (pipeline, GST_STATE_PLAYING);
78    g_assert (ret != GST_STATE_CHANGE_FAILURE);
79    bus = gst_element_get_bus (pipeline);
80    msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
81                                      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
82    gst_message_unref (msg);
83    gst_object_unref (bus);
84    gst_element_set_state (pipeline, GST_STATE_NULL);
85    gst_object_unref (pipeline);
86    return 0;
87  }
```

Listagem 8.4. Tocando um arquivo de vídeo Ogg usando elementos básicos.

Quando iniciado, o programa da Listagem 8.4 demultiplexa o arquivo Ogg fornecido e reproduz os primeiros subfluxos Vorbis e Theora que encontrar. Se não houverem tais subfluxos, o elemento "oggdemux" posta uma mensagem de erro no *bus* (indicando que não há *sink pad* conectada) o que causa o desbloqueio da *thread* da aplicação e, consequentemente, o término do programa.

Um detalhe sutil, mas importante no código da Listagem 8.4 é que a *callback* `pad_added_cb` e a função `main` executam em *threads* diferentes. Consequentemente, caso elas compartilhem dados, para evitar condições de corrida, é necessário sincronizar o acesso a esses dados via *mutexes* e semáforos (tipos `GMutex` e `GCond` da GLib). Na Listagem 8.4 não precisamos fazer isso porque as chamadas `gst_element_*` e `gst_bin_*` que usamos na *callback* para manipular o "pipeline" (único dado compartilhado) já são protegidas internamente por *mutexes*. Além disso, no intervalo em que as *threads* da *callback* e da aplicação estão concorrendo, linhas 77–80, a *thread* da aplicação não modifica o *pipeline*.

Os comandos gst-launch e gst-inspect

Os programas que vimos até agora (Listagens 8.1, 8.3 e 8.4) têm uma característica em comum: a topologia do *pipeline* não muda depois que ele é iniciado (transiciona para o estado *playing*). O comando *gst-launch*, instalado pelo pacote *gstreamer*, permite construir *pipelines* desse tipo diretamente na linha de comando.[3] Por exemplo, os três comandos seguintes constroem e iniciam *pipelines* idênticos aos das Listagens 8.1, 8.3 e 8.4:

```
$ gst-launch playbin uri="file://$PWD/bunny.ogg"
$ gst-launch filesrc location=bunny.mp3 ! mad ! alsasink
$ gst-launch filesrc location=bunny.ogg ! oggdemux name=demux\
               demux. ! vorbisdec ! audioconvert ! alsasink\
               demux. ! queue     ! theoradec    ! xvimagesink
```

[3]Em algumas instalações o nome do comando é *gst-launch-1.0*.

O primeiro comando anterior cria um *pipeline* contendo apenas o elemento "playbin", atribui a URI do vídeo a ser tocado à propriedade "uri" do elemento e inicia o *pipeline*. O segundo comando cria um *pipeline* com os elementos "filesrc", "mad" e "alsasink" conectados em série, atribui o caminho do vídeo à propriedade "location" do "filesrc" e incia o *pipeline*. E o terceiro comando cria um *pipeline* preliminar contento apenas os elementos "filesrc" e "oggdemux" conectados, atribui o caminho do vídeo à propriedade "location" do "filesrc" e inicia o *pipeline*; em seguida, quando o "oggdemux" instancia suas *sometimes pads*, o *gst-launch* adiciona e conecta as componentes conexas restantes, a saber, a série "vorbisdec", "audioconvert" e "alsasink" e a série "queue", "theoradec", e "xvimagesink".

O comando *gst-launch* é particularmente útil para depurar ou testar a viabilidade de *pipelines* antes de implementá-los em C. Por exemplo, usando o *gst-launch* podemos facilmente construir uma versão genérica do *pipeline* da Listagem 8.4, isto é, construir um *pipeline* que demultiplexa e decodifica um fluxo de mídia arbitrário (não apenas Ogg), obtido de uma fonte arbitrária (não apenas do disco), e que reproduz os fluxos resultantes num ambiente arbitrário (não apenas GNU/Linux). O comando, nesse caso, é o seguinte:

```
$ gst-launch uridecodebin uri="file://$PWD/bunny.ogg" name=decbin\
              decbin. ! audioconvert ! autoaudiosink\
              decbin. ! autovideosink
```

Esse *pipeline* possui quatro elementos: "uridecodebin", "autovideosink", "audioconvert" e "autoaudiosink". O elemento "uridecodebin" é um *bin* que demultiplexa e decodifica uma URI, isto é, ele lê o conteúdo de uma URI, demultiplexa-o, decodifica seus subfluxos e escreve os subfluxos decodificados em *sometimes pads* correspondentes. O elemento "audioconvert" é um conversor de formato de áudio PCM. E os elementos "autoaudiosink" e "autovideosink" são *sinks* abstratos de áudio e vídeo que detectam e usam os *sinks* concretos mais adequados para um dado ambiente. Por exemplo, no nosso ambiente GNU/Linux os elementos "autoaudiosink" e "autovideosink" utilizam os *sinks* concretos "alsasink" e "xvimagesink", os mesmos que usamos nos programas das Listagens 8.3 e 8.4, mas em outro ambiente essa escolha pode ser diferente. A Figura 8.7 apresenta a estrutura do *pipeline* criado pelo comando *gst-launch* anterior.

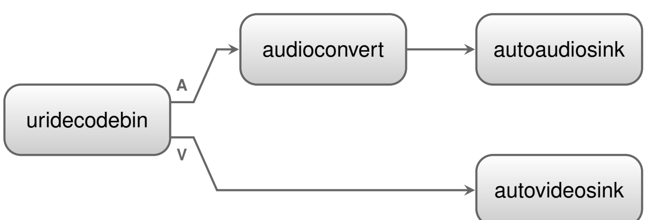


Figura 8.7. *Pipeline* genérico para reprodução de áudio e vídeo.

Além de *pipelines* para reprodução de áudio e vídeo, o comando *gst-launch* pode ser usado para criar *pipelines* que decodificam, transcodificam e transmitem fluxos de mídia na rede. Para mais informações sobre a sintaxe do comando (que é inspirada na sintaxe de *pipelines* da Bourne Shell), opções suportadas e exemplos de uso, veja a página de manual do *gst-launch* ("`man gst-launch`").

Outro programa instalado pelo pacote *gstreamer* é o *gst-inspect*, que permite inspecionar os *plugins* e elementos disponíveis no sistema.[4] Quando executado sem argumentos o *gst-inspect* imprime uma lista com os *plugins* e elementos disponíveis—no nosso ambiente, por exemplo, há 223 *plugins* e 1302 elementos disponíveis. Já quando executado com um nome de *plugin* ou nome de elemento como argumento o *gst-inspect* apresenta as informações do dado *plugin* ou elemento. A Listagem 8.5 apresenta a saída do comando "`gst-inspect equalizer`" que lista as informações do *plugin* "equalizer".

```
1  Plugin Details:
2    Name                     equalizer
3    Description              GStreamer audio equalizers
4    Filename                 /usr/lib/gstreamer-1.0/libgstequalizer.so
5    Version                  1.8.3
6    License                  LGPL
7    Source module            gst-plugins-good
8    Source release date      2016-08-19
9    Binary package           GStreamer Good Plugins (Arch Linux)
10   Origin URL               http://www.archlinux.org/
11
12   equalizer-nbands: N Band Equalizer
13   equalizer-3bands: 3 Band Equalizer
14   equalizer-10bands: 10 Band Equalizer
15
16   3 features:
17   +-- 3 elements
```

Listagem 8.5. Saída do comando "`gst-inspect equalizer`" no Arch Linux 4.7.4.

Na Listagem 8.5, as linhas 2–4 apresentam o nome do *plugin* ("equalizer"), a sua descrição (equalizadores de áudio) e o caminho da biblioteca dinâmica contendo o código do *plugin*. As linhas 5–10, apresentam informações adicionais (versão, licença, pacote, etc.) e as linhas 12–14 listam os elementos incluidos no *plugin*, no caso, "equalizer-nbands" (equalizador de *n* bandas), "equalizer-3bands" (equalizador de 3 bandas) e "equalizer-10bands" (equalizador de 10 bandas). Para listar as informações de um desses elementos, "equalizer-3bands" por exemplo, basta rodar o comando "`gst-inspect equalizer-3bands`". A saída nesse caso contém informações como a hierarquia de tipos do elemento, as interfaces que ele implementa, os *templates* de suas *pads*, as *pads* propriamente ditas (no caso, uma *source pad* com disponibilidade *always* do tipo "audio/x-raw", e uma *sink pad* também *always* e "audio/x-raw") e as propriedades do elemento. A Listagem 8.6 apresenta o trecho da saída do comando anterior em que as propriedades "band0", "band1" e "band2" são descritas. Pela descrição, essas propriedades controlam o ganho (dB) que o elemento aplica às faixas de frequência de 100Hz, 1100Hz e 11kHz; as três recebem valores no intervalo real [-24,12] e são inicializadas com zero.

```
1  band0: gain for the frequency band 100 Hz, ranging from -24.0 to +12.0
2         flags: readable, writable, controllable
3         Double. Range: -24 - 12 Default: 0
4  band1: gain for the frequency band 1100 Hz, ranging from -24.0 to +12.0
5         flags: readable, writable, controllable
6         Double. Range:  -24 - 12 Default: 0
7  band2: gain for the frequency band 11 kHz, ranging from -24.0 to +12.0
8         flags: readable, writable, controllable
9         Double. Range:  -24 -12 Default: 0
```

Listagem 8.6. Trecho da seção "Element Properties" da saída do comando "`gst-inspect equalizer-3bands`" no Arch Linux 4.7.4.

[4]Em algumas instalações o nome do comando é *gst-inspect-1.0*.

Os elementos equalizadores instalados pelo *plugin* "equalizer" anterior são processadores de áudio que permitem modificar o ganho dos graves, médios e agudos das amostras de um fluxo de áudio descomprimido. Os três elementos, "equalizer-nbands", "equalizer-3bands" e "equalizer-10bands", possuem a mesma estrutura: todos têm uma *source pad* que consome áudio descomprimido e uma *sink pad* que produz audio descomprimido, e todos eles processam uma amostra por vez, isto é, leem uma amostra da sua *source pad*, modificam-na acordo com os valores de ganho especificados em suas propriedades e escrevem a amostra resultante na sua *sink pad*. Elementos que realizam esse tipo de processamento são chamados de filtros.

8.4. Filtros

Um filtro é um elemento que aplica uma transformação sobre um fluxo de amostras descomprimidas. A transformação é normalmente ortogonal (depende apenas do conteúdo das amostras) e altera o fluxo original afim de produzir efeitos sonoros ou visuais no fluxo resultante. A Tabela 8.1 apresenta alguns filtros de áudio e vídeo disponíveis no GStreamer.

	Plugin	**Elemento**	**Descrição (controle)**
Áudio	audiofx	audiodynamic	Compressão ou expansão
	audiofx	audioecho	Eco
	audiofx	audiopanorama	Panorama estéreo
	equalizer	equalizer-nbands	Equalização (n bandas)
	freeverb	freeverb	Reverberação
	soundtouch	pitch	Tonalidade e tempo
	speed	speed	Velocidade
	volume	volume	Volume
Vídeo	coloreffects	coloreffects	Efeito de colorização
	effectv	dicetv	Efeito de fatiamento
	effectv	edgetv	Efeito de borda
	effectv	revtv	Efeito de relevo
	effectv	shagadelictv	Efeito de espiral caleidoscópica
	videocrop	videocrop	Recorte
	videofilter	videobalance	Brilho, cor e saturação
	videoscale	videoscale	Redimensionamento

Tabela 8.1. Alguns filtros de áudio e vídeo disponíveis no GStreamer.

Um filtro possui exatamente uma *source pad* e uma *sink pad*; ambas as *pads* têm disponibilidade *always* e escoam apenas fluxos descomprimidos ("audio/x-raw" ou "video/x-raw"). Consequentemente, num *pipeline* os elementos filtros normalmente aparecem entre os decodificadores e os *sinks*. Por exemplo, o comando a seguir adiciona o filtro "volume" ao *pipeline* da Figura 8.4, que reproduz um arquivo de áudio MP3.

```
$ gst-launch filesrc location=bunny.mp3\
             ! mad ! volume volume=.5 ! alsasink
```

Nesse caso, o áudio é reproduzido com a metade do volume original—a propriedade "volume" do elemento "volume", inicializada com o valor .5, indica ao elemento que o volume das amostras deve ser reduzido à metade.

Vejamos agora um exemplo mais complexo. O *shell script* da Listagem 8.7 usa o comando *gst-launch* para reproduzir um vídeo com um determinado efeito audiovisual. O *script* recebe dois argumentos, a URI do vídeo a ser reproduzido e o efeito a ser aplicado (número entre 0–4) e usa o comando *gst-launch* para montar o *pipeline* correspondente.

```
1  case "$2" in
2    0) af='volume volume=2';                       vf='coloreffects preset=1';;
3    1) af='pitch pitch=.5';                        vf='dicetv';;
4    2) af='audioecho delay=1000000000 intensity=1'; vf='shagadelictv';;
5    3) af='freeverb room-size=1 level=1';          vf='edgetv';;
6    4) af='equalizer-3bands band0=12 band1=-24';    vf='revtv';;
7  esac
8  if test -z "$1" || test -z "$vf" || test -z "$af" ; then
9    echo >&2 "usage: ${0##*/} URI [0-4]"; exit 1
10 fi
11 gst-launch uridecodebin uri="$1" name=decbin\
12   decbin. ! $af ! audioconvert ! autoaudiosink\
13   decbin. ! videoconvert ! $vf ! videoconvert ! autovideosink
```

Listagem 8.7. *Shell script* que reproduz um vídeo com efeito audiovisual.

A Figura 8.8 apresenta a estrutura final do *pipeline* criado pelo *script* da Listagem 8.7. Na figura, os elementos ⟨*audio filter*⟩ e ⟨*video filter*⟩ variam de acordo com o segundo argumento do *script*, e os elementos “audioconvert” e “videoconvert” garantem que os filtros selecionados gerem (e recebam, no caso do filtro de vídeo) amostras no formato esperado. Há cinco combinações de filtros possíveis:

- Se o argumento é 0, pela linha 2, o *script* utiliza o filtro de áudio “volume” (com “volume” 2) e o filtro de vídeo “coloreffects” (com “preset” 1). Nesse caso, as amostras de áudio são reproduzidas com o dobro do volume original e as amostras de vídeo são colorizadas para simular o efeito “câmera infravermelho” (Figura 8.9b).
- Se o argumento é 1, pela linha 3, o *script* utiliza o filtro de áudio “pitch” (com “pitch” .5) e o filtro de vídeo “dicetv”. Nesse caso, o tom (frequência) do áudio diminui para a metade do original e o vídeo ganha um efeito de fatiamento (Figura 8.9c).
- Se o argumento é 2, pela linha 4, o *script* utiliza o filtro de áudio “audioecho” (com “delay” 1s e “intensity” 1) e o filtro de vídeo “shagadelictv”. Nesse caso, o áudio ganha um efeito de eco e o vídeo ganha um efeito de espiral caleidoscópica (Figura 8.9d).
- Se o argumento é 3, pela linha 5, *script* utiliza o filtro áudio “freeverb” (com “room-size” 1 e “level” 1) e o filtro de vídeo “edgetv”. Nesse caso, o áudio ganha um efeito de reverberação numa sala ampla e o vídeo ganha um efeito de borda (Figura 8.9e).
- Finalmente, se o argumento é 4, pela linha 6, o *script* utiliza o filtro de áudio “equalizer-3bands” (com “band0” 12dB e “band1” -24dB) e o filtro de vídeo “revtv”. Nesse caso, o áudio têm os graves acentuados e médios suprimidos, e o video ganha um efeito de relevo (Figura 8.9f).

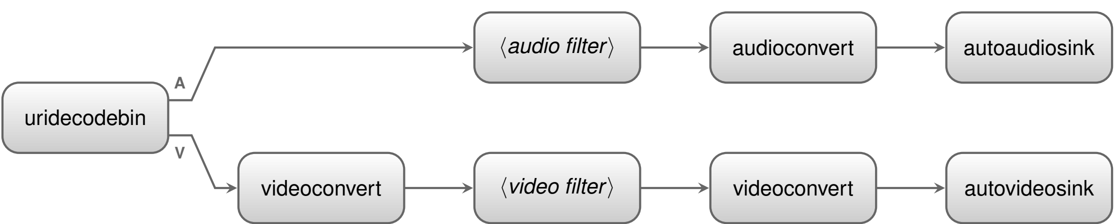
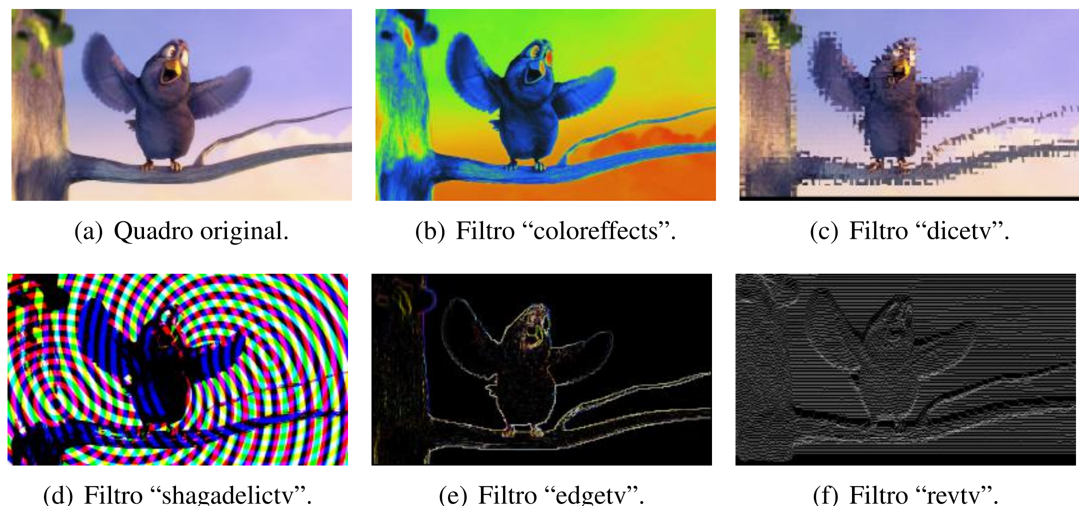


Figura 8.8. Estrutura final do *pipeline* da Listagem 8.7.



(a) Quadro original. (b) Filtro "coloreffects". (c) Filtro "dicetv".

(d) Filtro "shagadelictv". (e) Filtro "edgetv". (f) Filtro "revtv".

Figura 8.9. Efeitos de vídeo aplicados pelo *script* da Listagem 8.7.

A Listagem 8.8 apresenta um trecho do programa C equivalente ao *script* da Listagem 8.7. O trecho corresponde ao código da *callback* `pad_added_cb` disparada pelo sinal "pad-added" do elemento "uridecodebin". O código da função `main` apesar de omitido é trivial. Ele cria um elemento "uridecodebin", atribui à sua propriedade "uri" o primeiro argumento do programa, registra em seu sinal "pad-added" a *callback* `pad_added_cb` e passa como dado opcional da *callback* o segundo argumento do programa (número entre 0–4 que determina os filtros selecionados). Em seguida, a *thread* da aplicação transiciona o *pipeline* para *playing* e aguarda o fim da reprodução antes de liberar os dados alocados e terminar o programa. A operação da *callback* é similar àquela da Listagem 8.4. A diferença aqui é que os filtros são criados de acordo com o segundo argumento da aplicação (inteiro passado por referência à *callback* via o parâmetro `data`).

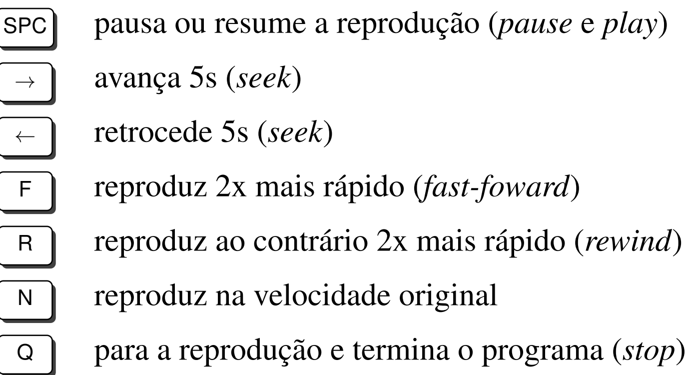
```
1  static void pad_added_cb (GstElement *src, GstPad *srcpad, gpointer data)
2  {
3    GstElement *pipeline;
4    GstCaps *caps;
5    const char *name;
6    GstPad *sinkpad;
7    GstPadLinkReturn ret;
8    gint filterno = *((gint *) data);
9    static gboolean has_audio_pad = FALSE, has_video_pad = FALSE;
10
11   pipeline = GST_ELEMENT (gst_element_get_parent (src));
12   caps = gst_pad_query_caps (srcpad, NULL);
13   name = gst_structure_get_name (gst_caps_get_structure (caps, 0));
14   if (g_str_equal (name, "audio/x-raw") && !has_audio_pad)
15     {
16       GstElement *filter = NULL, *conv, *sink;
17       switch (filterno)
18         {
19         case 0:
20           filter = gst_element_factory_make ("volume", NULL);
21           g_object_set (G_OBJECT (filter), "volume", 2.0, NULL);
22           break;
23         case 1:
24           filter = gst_element_factory_make ("pitch", NULL);
25           g_object_set (G_OBJECT (filter), "pitch", .5, NULL);
26           break;
27         case 2:
28           filter = gst_element_factory_make ("audioecho", NULL);
29           g_object_set (G_OBJECT (filter), "delay", GST_SECOND, "intensity", 1., NULL);
30           break;
```

```
31          case 3:
32            filter = gst_element_factory_make ("freeverb", NULL);
33            g_object_set (G_OBJECT (filter), "room-size", 1., "level", 1., NULL);
34            break;
35          case 4:
36            filter = gst_element_factory_make ("equalizer-3bands", NULL);
37            g_object_set (G_OBJECT (filter), "band0", 12., "band1", -24., NULL);
38            break;
39          }
40        conv = gst_element_factory_make ("audioconvert", NULL);
41        sink = gst_element_factory_make ("autoaudiosink", NULL);
42        g_assert (filter && conv && sink);
43        gst_bin_add_many (GST_BIN (pipeline), filter, conv, sink, NULL);
44        gst_element_sync_state_with_parent (filter);
45        gst_element_sync_state_with_parent (conv);
46        gst_element_sync_state_with_parent (sink);
47        gst_element_link_many (filter, conv, sink, NULL);
48        sinkpad = gst_element_get_static_pad (filter, "sink");
49        has_audio_pad = TRUE;
50      }
51    else if (g_str_equal (name, "video/x-raw") && !has_video_pad)
52      {
53        GstElement *pre, *filter = NULL, *pos, *sink;
54        switch (filterno)
55          {
56          case 0:
57            filter = gst_element_factory_make ("coloreffects", NULL);
58            g_object_set (G_OBJECT (filter), "preset", 1, NULL);
59            break;
60          case 1:
61            filter = gst_element_factory_make ("dicetv", NULL);
62            break;
63          case 2:
64            filter = gst_element_factory_make ("shagadelictv", NULL);
65            break;
66          case 3:
67            filter = gst_element_factory_make ("edgetv", NULL);
68            break;
69          case 4:
70            filter = gst_element_factory_make ("revtv", NULL);
71            break;
72          }
73        pre = gst_element_factory_make ("videoconvert", NULL);
74        pos = gst_element_factory_make ("videoconvert", NULL);
75        sink = gst_element_factory_make ("autovideosink", NULL);
76        g_assert (pre && filter && pos && sink);
77        gst_bin_add_many (GST_BIN (pipeline), pre, filter, pos, sink, NULL);
78        gst_element_sync_state_with_parent (pre);
79        gst_element_sync_state_with_parent (filter);
80        gst_element_sync_state_with_parent (pos);
81        gst_element_sync_state_with_parent (sink);
82        gst_element_link_many (pre, filter, pos, sink, NULL);
83        sinkpad = gst_element_get_static_pad (pre, "sink");
84        has_video_pad = TRUE;
85      }
86    else
87      {
88        goto done;
89      }
90    ret = gst_pad_link (srcpad, sinkpad);
91    g_assert (GST_PAD_LINK_SUCCESSFUL (ret));
92    gst_object_unref (sinkpad);
93   done:
94    gst_caps_unref (caps);
95    gst_object_unref (pipeline);
96  }
```

Listagem 8.8. Trecho do programa C equivalente ao *script* da Listagem 8.7.

8.5. Adicionando os controles *play*, *pause*, *stop*, *seek*, *fast-foward* e *rewind*

Os programas que vimos até agora não são interativos. Assim que inciados, eles constroem um *pipeline* que reproduz o conteúdo uma única vez do inicio ao fim. Nesta seção, vamos adicionar os controles *play*, *pause*, *stop*, *seek*, *fast-forward* e *rewind* ao programa "Olá mundo" da Listagem 8.1. Nosso objetivo é fazer com que o programa responda aos seguintes comandos de tecla:



Escolhemos o programa "Olá mundo" apenas para simplificar as listagens apresentadas. Os mecanismos para captura de teclas e controle do *pipeline* descritos sobre esse exemplo são gerais e se aplicam a qualquer aplicação GStreamer. Para obter teclas vamos capturar as mensagens de eventos de navegação (`GST_NAVIGATION_MESSAGE_EVENT`) postadas pelo *sink* de vídeo no *bus* do *pipeline*. Para implementar as operações de *play*, *pause* e *stop*, vamos transicionar o *pipeline* para os estados correspondentes. E para implementar as operações *seek*, *fast-foward* e *rewind*, vamos postar eventos de *seek* (`GST_EVENT_SEEK`) que requisitam mudanças na posição e na taxa de reprodução do conteúdo.

A Listagem 8.9 apresenta o trecho de código contendo a função `main` do novo programa "Olá mundo" (versão com controles). Até a linha 27, o código da função é idêntico ao código correspondente na Listagem 8.1. A diferença aqui está na forma como capturamos mensagens postadas no *bus*. Até agora, para obter mensagens do *bus*, usamos a função `gst_bus_timed_pop_filtered`, que bloqueia a *thread* da aplicação até que tipos específicos de mensagens sejam postados. Na Listagem 8.9, como a aplicação é orientada a eventos, usamos uma técnica diferente, baseada no *loop* de eventos da GLib.

O *loop* de eventos é criado pela chamada `g_main_loop_new` na linha 28 da Listagem 8.9. O primeiro argumento da chamada (`NULL`) indica que o *loop* irá receber eventos de qualquer fonte registrada via GLib, e o segundo argumento (`FALSE`) indica que o *loop* não deve ser iniciado imediatamente (iniciaremos o *loop* posteriormente).

As chamadas `g_object_set_data` seguintes (linhas 29–30) armazenam no *bus* ponteiros para o *pipeline* e para o *loop*. O *bus* (`GstBus`) também é um objeto (`GObject`) e, como todo objeto, possui uma tabela *hash* em que podem ser armazenados dados de usuário (*user data*). As chamadas das linhas 29–30 armazenam na *hash* do *bus* um ponteiro para o *pipeline*, indexado pela chave "pipeline", e um ponteiro para o*loop*, indexado pelo *loop*. Isso é necessário porque mais tarde, na *callback* do *bus*, não temos acesso direto à esses objetos.

A chamada `gst_bus_add_watch` (linha 31) registra a *callback* `bus_cb` como tratadora (*listener*) de mensagens *bus*. Ou seja, a *callback* será chamada no *loop* de eventos sempre que uma mensagem é postada no *bus*. E a chamada seguinte (linha 32) libera a referência obtida na linha 27.

A chamada `g_main_loop_run` (linha 34) inicia o *loop* de eventos da GLib. A partir desse ponto a aplicação torna-se orientada a eventos, isto é, a *thread* da aplicação passa a ser controlada pela GLib que apenas espera eventos e repassa-os às *callbacks* registradas (`bus_cb`, no caso). O controle só volta à `main` quando o *loop* é terminado explicitamente via uma chamada `g_main_loop_quit`—o que ocorre na *callback* do *bus*. Nesse caso, as chamadas restantes (linhas 36–38) liberam os dados alocados e terminam o programa.

```
1  #include <glib.h>
2  #include <gst/gst.h>
3  #include <gst/video/navigation.h>
4
5  static gboolean bus_cb (GstBus *bus, GstMessage *msg, gpointer data);
6
7  int main (int argc, char *argv[])
8  {
9    GstElement *playbin;
10   char *uri;
11   GstStateChangeReturn ret;
12   GstBus *bus;
13   GMainLoop *loop;
14
15   gst_init (&argc, &argv);
16   playbin = gst_element_factory_make ("playbin", "hello");
17   g_assert_nonnull (playbin);
18
19   uri = gst_filename_to_uri ("bunny.ogg", NULL);
20   g_assert_nonnull (uri);
21   g_object_set (G_OBJECT (playbin), "uri", uri, NULL);
22   g_free (uri);
23
24   ret = gst_element_set_state (playbin, GST_STATE_PLAYING);
25   g_assert (ret != GST_STATE_CHANGE_FAILURE);
26
27   bus = gst_element_get_bus (playbin);
28   loop = g_main_loop_new (NULL, FALSE);
29   g_object_set_data (G_OBJECT (bus), "pipeline", playbin);
30   g_object_set_data (G_OBJECT (bus), "loop", loop);
31   gst_bus_add_watch (bus, bus_cb, NULL);
32   gst_object_unref (bus);
33
34   g_main_loop_run (loop);       /* event loop */
35
36   g_main_loop_unref (loop);
37   gst_element_set_state (playbin, GST_STATE_NULL);
38   gst_object_unref (playbin);
39   return 0;
40 }
```

Listagem 8.9. Função `main` do programa “Olá mundo” com controles.

Vejamos agora a operação da *callback* `bus_cb`, cujo código é apresentado na Listagem 8.10. A cada mensagem postada no *bus*, o *loop* de eventos chama essa *callback* com três argumentos: o *bus*, a mensagem postada e o dado adicional passado à função `gst_bus_add_watch` (linha 31 da Listagem 8.9) no momento do registro da *callback* (no caso, `NULL`).

Na Listagem 8.10, as chamadas das linhas 3 e 4 obtém as referências para o *pipeline* e para o *loop* armazenadas na *hash* do *bus*. E o comando `switch` seguinte (linhas 5–37) trata a mensagem recebida de acordo com o seu tipo. Há três possibilidades:

1. Se a mensagem indica um erro ou o fim do fluxo (EOS), a *callback* termina o *loop* de eventos (linha 40) e remove a si mesmo da lista de tratadores (instrução "`return FALSE`" na linha 41).
2. Se a mensagem é proveniente de um elemento (linha 10), é uma mensagem de evento de navegação (linhas 14–15) e indica um evento de pressionamento de tecla (linhas 18–19), então a *callback* executa a operação associada à tecla pressionada. Ou seja, a *callback* chama a função auxiliar `toggle_pause` se a tecla é "space", chama a função auxiliar `seek` se a tecla é "right" ou "left", chama a função auxiliar `speed` se a tecla é "f", "r" ou "n", e termina o *loop* se a tecla é "q".
3. Caso contrário, a mensagem é ignorada (linha 38).

```
1  static gboolean bus_cb (GstBus *bus, GstMessage *msg, gpointer data)
2  {
3    GstElement *pipeline = g_object_get_data (G_OBJECT (bus), "pipeline");
4    GMainLoop *loop = g_object_get_data (G_OBJECT (bus), "loop");
5    switch (GST_MESSAGE_TYPE (msg))
6      {
7      case GST_MESSAGE_ERROR:
8      case GST_MESSAGE_EOS:
9        goto quit;
10     case GST_MESSAGE_ELEMENT:
11       {
12         GstEvent *evt;
13         const char *key;
14         if (gst_navigation_message_get_type (msg) != GST_NAVIGATION_MESSAGE_EVENT)
15           break;
16         if (!gst_navigation_message_parse_event (msg, &evt))
17           break;
18         if (gst_navigation_event_get_type (evt) != GST_NAVIGATION_EVENT_KEY_PRESS)
19           break;
20         gst_navigation_event_parse_key_event (evt, &key);
21         if (g_ascii_strcasecmp (key, "space") == 0)
22           toggle_pause (pipeline);
23         else if (g_ascii_strcasecmp (key, "right") == 0)
24           seek (pipeline, 5 * GST_SECOND);
25         else if (g_ascii_strcasecmp (key, "left") == 0)
26           seek (pipeline, -5 * GST_SECOND);
27         else if (g_ascii_strcasecmp (key, "f") == 0)
28           speed (pipeline, 2.);
29         else if (g_ascii_strcasecmp (key, "r") == 0)
30           speed (pipeline, -2.);
31         else if (g_ascii_strcasecmp (key, "n") == 0)
32           speed (pipeline, 1.);
33         else if (g_ascii_strcasecmp (key, "q") == 0)
34           goto quit;
35         break;
36       }
37     }
38   return TRUE;
39  quit:
40   g_main_loop_quit (loop);
41   return FALSE;
42 }
```

Listagem 8.10. Função `bus_cb` do programa "Olá mundo" com controles.

A Listagem 8.11 apresenta o código das funções auxiliares `toggle_pause`, `seek` e `resume` chamadas pela *callback* `bus_cb` da Listagem 8.10.

A função `toggle_pause` (linhas 1–12) alterna o estado do *pipeline* entre *paused* e *playing* dependendo do estado atual: se o *pipeline* está pausado ela o inicia, e se o *pipeline* está tocando ela o pausa.

A função `seek` (linhas 14–23) requisita um salto de `offset` nanosegundos (ns) na reprodução do conteúdo a partir do seu ponto atual. A chamada da linha 17 obtém o tempo absoluto do *pipeline* (ns) e armazena-o em `from`. A atribuição seguinte calcula o tempo absoluto (ns) em que o *pipeline* deve estar após o salto e armazena-o em `to`. E a chamada `gst_element_seek_simple` (linhas 19–22) posta um evento de *seek* no *pipeline* que requisita um salto na reprodução para o novo tempo absoluto `to`. A função `gst_element_seek_simple` recebe quatro argumentos: o elemento alvo da operação, o formato do deslocamento, *flags* opcionais e o valor do deslocamento. O formato (segundo argumento) indica o formato em que o descolocamento está especificado—no caso, `GST_FORMAT_TIME` indica um deslocamento no tempo. As *flags* (terceiro argumento) determinam as características da operação: a *flag* `GST_SEEK_FLAG_ACCURATE` indica que a operação deve ter uma precisão razoável, a *flag* `GST_SEEK_FLAG_FLUSH` indica que *caches* internos de elementos subsequentes devem ser descartados e a *flag* `GST_SEEK_FLAG_FLUSH` indica que (se possível) durante a operação apenas quadros chave (*key frames*) devem ser decodificados.

A última função, `speed`, (linhas 25–49) requisita uma mudança na taxa de reprodução do *pipeline* de acordo com o parâmetro *rate*. Esse tipo de mudança de taxa é feita por meio da função `gst_element_seek`, também utilizada para realizar operações de *seek* complexas. Essa função recebe oito argumentos: (1) o elemento alvo; (2) a nova taxa de reprodução; (3) o formato dos deslocamentos; (4) *flags* opcionais; (5) o tipo do início do desolamento; (6) o novo inicio; (7) o tipo do fim do deslocamento; e (8) o novo fim. Como aqui nosso objetivo é apenas alterar a taxa de reprodução o deslocamento é a própria posição atual do *fluxo* (variável `from` preenchida na linha 31). Se a taxa é positiva, o deslocamento é relativo ao início (linhas 34–37), caso contrário ele é relativo ao fim (linhas 41–44). Na prática, valores de taxa superiores a 1 aceleram a reprodução, valores entre 0 e 1 desaceleram a reprodução produzindo um efeito de “câmera lenta”, e valores negativos aplicam uma aceleração negativa, o que faz com que o conteúdo seja tocado ao contrário.

```
1  static void toggle_pause (GstElement *pipeline)
2  {
3    switch (GST_STATE (pipeline))
4      {
5      case GST_STATE_PAUSED:
6        gst_element_set_state (pipeline, GST_STATE_PLAYING);
7        break;
8      case GST_STATE_PLAYING:
9        gst_element_set_state (pipeline, GST_STATE_PAUSED);
10       break;
11     }
12 }
13
14 static void seek (GstElement *pipeline, GstClockTimeDiff offset)
15 {
16   GstClockTimeDiff from, to;
17   gst_element_query_position (pipeline, GST_FORMAT_TIME, &from);
```

```
18    to = MAX (from + offset, 0);
19    gst_element_seek_simple (pipeline, GST_FORMAT_TIME,
20                             GST_SEEK_FLAG_ACCURATE
21                             | GST_SEEK_FLAG_FLUSH
22                             | GST_SEEK_FLAG_TRICKMODE, to);
23  }
24
25  static void speed (GstElement *pipeline, double rate)
26  {
27    GstClockTimeDiff from;
28    GstSeekType start, stop;
29    GstClockTimeDiff start_time, stop_time;
30
31    gst_element_query_position (pipeline, GST_FORMAT_TIME, &from);
32    if (rate >= 0.)
33      {
34        start = GST_SEEK_TYPE_SET;
35        start_time = from;
36        stop = GST_SEEK_TYPE_NONE;
37        stop_time = GST_CLOCK_TIME_NONE;
38      }
39    else
40      {
41        start = GST_SEEK_TYPE_NONE;
42        start_time = GST_CLOCK_TIME_NONE;
43        stop = GST_SEEK_TYPE_SET;
44        stop_time = from;
45      }
46    gst_element_seek (pipeline, rate, GST_FORMAT_TIME,
47                      GST_SEEK_FLAG_FLUSH | GST_SEEK_FLAG_TRICKMODE,
48                      start, start_time, stop, stop_time);
49  }
```

Listagem 8.11. Funções `toggle_paused`, `seek` e `speed` do programa "Olá mundo" com controles.

Finalmente, observe que o programa anterior usa as declarações do cabeçalho "gstnavigation.h" (Listagem 8.9, linha 3) instalado pelo pacote *gst-plugins-base*. Ou seja, para compilar o programa é necessário informar ao pré-processador o caminho do diretório contendo esse cabeçalho e ao *linker* os nomes e os caminhos dos diretórios contendo as bibliotecas correspondentes. Para tal basta adicionar o módulo "gstreamer-video-1.0" à variável `MODULES` (linha 2) do *makefile* da Listagem 8.2.

8.6. Plugins

Nesta seção, vamos discutir como novos elementos podem ser criados, encapsulados em *plugins* (bibliotecas dinâmicas), instalados e usados por aplicações GStreamer. Mais especificamente, vamos escrever dois elementos: "myfilter" e "myvideofilter". O elemento "myfilter" é um filtro genérico que lê uma amostra da sua *sink pad*, imprime o seu tamanho na saída padrão e repassa-a à sua *source pad*. Já o elemento "myvideofilter" é um filtro de vídeo que aplica um efeito de ondulação aos quadros que o atravessam.

Elemento "myfilter"

A Listagem 8.12 apresenta o código do elemento "myfilter" e do *plugin* correspondente (também chamado "myfilter"). Na listagem, a estrutura `GstMyFilter` (linhas 4–7) contém os dados de uma instância de um elemento "myfilter", a saber, os dados do tipo pai (`GstElement`) e ponteiros para as *pads* do filtro. A diretiva da linha 9 define uma constante simbólica com o tipo do novo elemento, e as macros seguintes (linhas 10 e 11) declaram e definem o tipo propriamente dito (`GstMyFilter`).

```
1  #include <glib.h>
2  #include <gst/gst.h>
3
4  typedef struct _GstMyFilter {
5    GstElement parent_instance;
6    GstPad *sinkpad, *srcpad;
7  } GstMyFilter;
8
9  #define GST_TYPE_MY_FILTER (gst_my_filter_get_type ())
10 G_DECLARE_FINAL_TYPE (GstMyFilter, gst_my_filter, GST, MY_FILTER, GstElement)
11 G_DEFINE_TYPE (GstMyFilter, gst_my_filter, GST_TYPE_ELEMENT)
12
13 static GstStaticPadTemplate sink_template =
14   GST_STATIC_PAD_TEMPLATE ("sink", GST_PAD_SINK,
15                            GST_PAD_ALWAYS, GST_STATIC_CAPS ("ANY"));
16
17 static GstStaticPadTemplate src_template =
18   GST_STATIC_PAD_TEMPLATE ("src", GST_PAD_SRC,
19                            GST_PAD_ALWAYS, GST_STATIC_CAPS ("ANY"));
20
21 static GstFlowReturn gst_my_filter_chain (GstPad *pad, GstObject *obj, GstBuffer *buf)
22 {
23   GstMyFilter *filter = GST_MY_FILTER (obj);
24   g_print ("Processed %" G_GSIZE_FORMAT" bytes\n", gst_buffer_get_size (buf));
25   return gst_pad_push (filter->srcpad, buf);
26 }
27
28 static void gst_my_filter_class_init (GstMyFilterClass *klass)
29 {
30   GstElementClass *gstelement_class = GST_ELEMENT_CLASS (klass);
31   gst_element_class_set_static_metadata
32     (gstelement_class, "An example filter", "Example/myfilter",
33      "Example filter", "roberto@telemidia.puc-rio.br");
34   gst_element_class_add_pad_template
35     (gstelement_class, gst_static_pad_template_get (&src_template));
36   gst_element_class_add_pad_template
37     (gstelement_class, gst_static_pad_template_get (&sink_template));
38 }
39
40 static void gst_my_filter_init (GstMyFilter *filter)
41 {
42   GstPad *sink = gst_pad_new_from_static_template (&sink_template, "sink");
43   GstPad *src = gst_pad_new_from_static_template (&src_template, "src");
44   gst_pad_set_chain_function (sink, GST_DEBUG_FUNCPTR (gst_my_filter_chain));
45   GST_PAD_SET_PROXY_CAPS (sink);
46   gst_element_add_pad (GST_ELEMENT (filter), sink);
47   filter->sinkpad = sink;
48   GST_PAD_SET_PROXY_CAPS (src);
49   gst_element_add_pad (GST_ELEMENT (filter), src);
50   filter->srcpad = src;
51 }
52
53 static gboolean my_filter_plugin_init (GstPlugin *plugin)
54 {
55   return gst_element_register (plugin, "myfilter", GST_RANK_NONE, GST_TYPE_MY_FILTER);
56 }
57
58 #define PACKAGE "myfilter"
59 GST_PLUGIN_DEFINE (
60   GST_VERSION_MAJOR, GST_VERSION_MINOR,
61   myfilter, "Contains the myfilter element", my_filter_plugin_init,
62   "1.0", "LGPL", "TeleMidia/PUC-Rio", "http://www.telemidia.puc-rio.br"
63 )
```

Listagem 8.12. Código do elemento “myfilter” e *plugin* correspondente.

A macro `GST_DECLARE_FINAL_TYPE` (linha 10) recebe cinco argumentos—o nome do novo tipo (`GstMyFilter`), o prefixo das funções do tipo (`gst_my_filter`), o prefixo das macros do tipo (`GST`), o sufixo das macros do tipo (`MY_FILTER`) e o tipo pai `GstElement`—e usa esses argumentos para emitir as declarações necessárias. Já a macro `GST_DEFINE_TYPE` (linha 11) recebe três argumentos—o nome do novo tipo, o prefixo das funções do tipo e o código (`GType`) do tipo pai—e define a função `gst_my_filter_get_type` que retorna o código `GType` do novo tipo. Ambas as macros fazem parte do *framework* de objetos da GLib.

Continuando, as instruções das linhas 13–18 declaram e inicializam duas variáveis estáticas, `src_template` e `sink_template`, com os *templates* para as *pads* do filtro—no caso, ambas as *pads* terão disponibilidade *always* e *caps* "any" (aceitarão qualquer formato de *buffers*).

A função `gst_my_filter_chain` (linhas 21–25) é a função que processa os *buffers* recebidos pela *sink pad* do filtro e os repassa à *source pad*. A função recebe três argumentos: um ponteiro para a *sink pad*, um ponteiro para o objeto filtro e um ponteiro para o *buffer*; e retorna um código de *status* correspondente. Nesse exemplo, a função apenas imprime o tamanho do *buffer* na saída padrão (linha 24) e repassa-o à *source pad* do elemento via a chamada `gst_pad_push`.

A função `gst_my_filter_class_init` (linhas 28–38) é utilizada pela GLib para inicializar o tipo (classe) `GstMyFilter`. Ele é chamada uma única vez (antes de o primeiro objeto do tipo ser instanciado) e, no caso, registra os metadados do novo elemento "myfilter" (linhas 31–33) e adiciona à classe do elemento os *templates* das suas futuras *pads* (linhas 34–37).

A função `gst_my_filter_init` (linhas 40–51) é utilizada pela GLib para inicializar cada instância (objeto) do tipo `GstMyFilter`. A função, nesse caso, cria as *pads* do novo filtro (linhas 42–43), inicializa-as (linhas 45 e 48), adiciona-as ao elemento (linhas 46–47 e 49–50), e registra a função de encadeamento na *sink pad* (linha 44). A função de encadeamento (`gst_my_filter_chain`, linhas 21–25) é chamada sempre que um *buffer* é recebido pela *sink pad* do elemento.

O restante do código (linhas 53–63) contém o ponto de entrada do *plugin* (função `my_filter_plugin_init`, linhas 53–56) e as declarações associadas. O ponto de entrada, função `my_filter_plugin_init`, é executada assim que o *plugin* (biblioteca dinâmica) "myfilter" é processado pelo GStreamer (o que ocorre durante a chamada `gst_init`) e, nesse caso, simplesmente adiciona os dados do novo *plugin* ao registro do GStreamer. Já a macro `GST_PLUGIN_DEFINE` (linhas 58–63) emite as declarações do *plugin* a partir dos argumentos fornecidos, a saber, versão mínima do GStreamer requirida, nome simbólico do *plugin*, descrição, ponto de entrada, versão, licença, pacote e origem).

Compilando e usando o elemento "myfilter"

Um *plugin* GStreamer é uma biblioteca dinâmica carregada em tempo de execução pelo *framework*. O código da Listagem 8.12 define um *plugin* chamado "myfilter" contendo apenas um elemento, também chamado "myfilter". Para gerar uma biblioteca dinâmica a partir desse código é necessário informar ao compilador os diretórios contendo os

cabeçalhos utilizados (diretivas *include*) e ao *linker* os nomes e diretórios das bibliotecas utilizadas. Além disso, é preciso instruir o *linker* a produzir uma biblioteca dinâmica, ao invés de um executável. No GNU/Linux, isso pode ser feito através do comando:

```
$ cc myfilter.c -o myfilter.so -shared -fPIC\
     `pkg-config --cflags --libs gstreamer-1.0 gstreamer-base-1.0`
```

Esse comando compila e *link*-edita uma biblioteca dinâmica (arquivo "myfilter.so") contendo o *plugin* da Listagem 8.12.

Finalmente, para construir um *pipeline* usando o *plugin* "myfilter" basta instalá-lo num dos diretórios de *plugins* do GStreamer ou indicar ao GStreamer que esse *plugin* deve ser carregado. Por exemplo, assumindo que o arquivo "myfilter.so" gerado pelo comando anterior está no diretório atual, o seguinte comando monta um *pipeline* contendo o elemento "myfilter":

```
$ gst-launch --gst-plugin-path=.\
     videotestsrc ! myfilter ! ximagesink
```

O valor "." para a opção "- -gst-plugin-path" informa ao *gst-launch* que, além dos diretórios padrão, *plugins* também devem ser buscados no diretório corrente ("."). Por conta disso, assim que é iniciado o *gst-launch* carrega o *plugin* "myfilter" que adiciona elemento "myfilter" ao registro do GStreamer, tornando-o "visível" à aplicação. Como o elemento "myfilter" consome e produz dados de qualquer tipo (isto é, as *caps* de ambas as suas *pads* são "any") podemos utilizá-lo entre quaisquer elementos. O comando anterior utiliza-o entre os elementos "videotestsrc", que produz *buffers* de teste de vídeo, e "autovideosink", que mostra esses *buffers* na tela. Se tudo der certo, quando o comando é executado os *buffers* de teste são exibidos na tela e o tamanho de cada *buffer* é impresso pelo "myfilter" na saída padrão.

Elemento "myvideofilter"

O próximo exemplo, elemento "myvideofilter", é um filtro de vídeo. A diferença desse elemento em relação ao anterior ("myfilter") é que aqui estamos interessados em filtrar apenas quadros de vídeo. Dessa forma, ao invés de herdar diretamente do tipo `GstElement`, vamos herdar do tipo especializado `GstVideoFilter`, que já trata as tarefas comuns envolvidas no processamento de quadros de vídeo. Além desse tipo, o GStreamer possui tipos especializados para construção de *sources* (`GstBaseSrc`), *sinks* (`GstBaseSink`), filtros em geral (`GstBaseTransform`), filtros de áudio (`GstAudioFilter`), etc.

O elemento "myvideofilter" é um filtro de vídeo que aplica um efeito de ondulação aos amostras que o atravessam. Mais precisamente, o elemento transforma cada pixel (u,v) do quadro recebido num pixel (x,y) no quadro resultante, de acordo com a fórmula:

$$x(u,v) = u + 20 \times \sin(\mathit{factor} \times v)$$
$$y(u,v) = v,$$

em que *factor* é o valor da propriedade "factor" (número real) do elemento.

A Listagem 8.12 apresenta o código do elemento "myvideofilter" e do *plugin* correspondente. Na listagem, a estrutura `GstMyVideoFilter` (linhas 8–11) contém os

dados de uma instância de um elemento "myvideofilter", a saber, os dados do tipo pai `GstVideoFilter` e o valor corrente da propriedade "factor" do elemento. As diretivas das linhas 15–18 declaram e definem o novo tipo (`GstMyVideoFilter`), que agora herda de `GstVideoFilter`, e as instruções seguintes (linhas 20–26) declaram e inicializam variáveis contendo os *templates* das *pads* do elemento—ambas as *pads* esperam *buffers* de vídeo nos formatos RGBx ou RGB.

```
1  #include <math.h>
2  #include <string.h>
3  #include <glib.h>
4  #include <gst/gst.h>
5  #include <gst/video/video.h>
6  #include <gst/video/gstvideofilter.h>
7
8  typedef struct _GstMyVideoFilter {
9    GstVideoFilter parent_instance;
10   gdouble factor;
11 } GstMyVideoFilter;
12
13 enum { PROP_0, PROP_FACTOR };
14
15 #define GST_TYPE_MY_VIDEO_FILTER (gst_my_video_filter_get_type ())
16 G_DECLARE_FINAL_TYPE (GstMyVideoFilter, gst_my_video_filter,
17                       GST, MY_VIDEO_FILTER, GstVideoFilter)
18 G_DEFINE_TYPE (GstMyVideoFilter, gst_my_video_filter, GST_TYPE_VIDEO_FILTER)
19
20 static GstStaticPadTemplate sink_template =
21   GST_STATIC_PAD_TEMPLATE ("sink", GST_PAD_SINK, GST_PAD_ALWAYS,
22                            GST_STATIC_CAPS (GST_VIDEO_CAPS_MAKE ("{BGRx, RGB}")));
23
24 static GstStaticPadTemplate src_template =
25   GST_STATIC_PAD_TEMPLATE ("src", GST_PAD_SRC, GST_PAD_ALWAYS,
26                            GST_STATIC_CAPS (GST_VIDEO_CAPS_MAKE ("{BGRx, RGB}")));
27
28 static void gst_my_video_filter_get_property (GObject *obj, guint id,
29                                               GValue *val, GParamSpec *pspec)
30 {
31   GstMyVideoFilter *filter = GST_MY_VIDEO_FILTER (obj);
32   switch (id)
33     {
34     case PROP_FACTOR:
35       g_value_set_double (val, filter->factor);
36       break;
37     default:
38       G_OBJECT_WARN_INVALID_PROPERTY_ID (obj, id, pspec);
39     }
40 }
41
42 static void gst_my_video_filter_set_property (GObject *obj, guint id,
43                                               const GValue *val, GParamSpec *pspec)
44 {
45   GstMyVideoFilter *filter = GST_MY_VIDEO_FILTER (obj);
46   switch (id)
47     {
48     case PROP_FACTOR:
49       filter->factor = g_value_get_double (val);
50       break;
51     default:
52       G_OBJECT_WARN_INVALID_PROPERTY_ID (obj, id, pspec);
53     }
54 }
55
56 static GstFlowReturn gst_my_video_transform (GstVideoFilter *vf, GstVideoFrame *in,
57                                              GstVideoFrame *out)
58 {
59   GstMyVideoFilter *filter = GST_MY_VIDEO_FILTER (vf);
60   gint i, j, w, h;
```

```
61    gint stride, pixel_stride;
62    guint8 *data, *indata;
63    guint outoffset, inoffset, u;
64
65    indata = GST_VIDEO_FRAME_PLANE_DATA (in, 0);
66    data = GST_VIDEO_FRAME_PLANE_DATA (out, 0);
67    stride = GST_VIDEO_FRAME_PLANE_STRIDE (out, 0);
68    w = GST_VIDEO_FRAME_COMP_WIDTH (out, 0);
69    h = GST_VIDEO_FRAME_COMP_HEIGHT (out, 0);
70    pixel_stride = GST_VIDEO_FRAME_COMP_PSTRIDE (out, 0);
71    for (i = 0; i < h; i++)
72      {
73        for(j = 0; j < w; j++)
74          {
75            outoffset = (i * stride) + (j * pixel_stride);
76            u = i + 20 * sin (filter->factor * j);
77            inoffset = (u * stride) + (j * pixel_stride);
78            if (u < h)
79              memcpy (data + outoffset, indata + inoffset, pixel_stride);
80          }
81      }
82    return GST_FLOW_OK;
83  }
84
85  static void gst_my_video_filter_class_init (GstMyVideoFilterClass *klass)
86  {
87    GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
88    GstElementClass *gstelement_class = GST_ELEMENT_CLASS (klass);
89    GstVideoFilterClass *gstvideofilter_class = GST_VIDEO_FILTER_CLASS (klass);
90
91    gobject_class->set_property = gst_my_video_filter_set_property;
92    gobject_class->get_property = gst_my_video_filter_get_property;
93    g_object_class_install_property
94      (gobject_class, PROP_FACTOR, g_param_spec_double
95       ("factor", "factor", "distortion factor",
96        0.0, 320.0, 2*G_PI/130, G_PARAM_READWRITE));
97
98    gst_element_class_set_static_metadata
99      (gstelement_class, "An example video filter", "Example/myvideofilter",
100      "Example filter", "roberto@telemidia.puc-rio.br");
101   gst_element_class_add_pad_template
102     (gstelement_class, gst_static_pad_template_get (&src_template));
103   gst_element_class_add_pad_template
104     (gstelement_class, gst_static_pad_template_get (&sink_template));
105
106   gstvideofilter_class->transform_frame = GST_DEBUG_FUNCPTR (gst_my_video_transform);
107 }
108
109 static void gst_my_video_filter_init (GstMyVideoFilter *filter)
110 {
111   filter->factor = 2*G_PI/130;
112 }
113
114 static gboolean my_video_filter_plugin_init (GstPlugin *plugin)
115 {
116   return gst_element_register (plugin, "myvideofilter",
117                               GST_RANK_NONE, GST_TYPE_MY_VIDEO_FILTER);
118 }
119
120 #define PACKAGE "myvideofilter"
121 GST_PLUGIN_DEFINE (
122   GST_VERSION_MAJOR, GST_VERSION_MINOR,
123   myvideofilter, "Contains the myvideofilter element", my_video_filter_plugin_init,
124   "1.0", "LGPL", "TeleMidia/PUC-Rio", "http://www.telemidia.puc-rio.br"
125 )
```

Listagem 8.13. Código do elemento “myvideofilter” e *plugin* correspondente.

Na Listagem 8.13, a declaração *enum* da linha 13 cria uma constante simbólica para representar a propriedade "factor" (`PROP_FACTOR`). A lógica de obtenção e atribuição de valores de propriedades está contida nas funções `gst_my_video_filter_get_property` (linhas 28–40) e `gst_my_video_filter_set_property` (linhas 42–54). Nesse caso, o elemento reconhece apenas uma propriedade, "factor", cujo valor é armazenado ou obtido a partir do campo correspondente da estrutura `GstMyVideoFilter`.

A função `gst_my_video_filter_class_init` (linhas 85–107) inicializa o tipo (classe) `GstMyVideoFilter`. Aqui, além de registrar os metadados do elemento (linhas 98–100) e os *templates* das futuras *caps*, a função sobrescreve as funções para manipulação de propriedades (herdadas de `GObject`, linhas 91–92), "instala" na classe os metadados da propriedade "factor" (linhas 93–96) e sobrescreve a função que processa os *buffers* de vídeo recebidos (herdada de `GstVideoFilter`, linha 106). Já a função `gst_my_video_filter_init` (linhas 109–112), nesse caso, apenas inicializa a propriedade "factor" com o seu valor *default*.

A função `gst_my_video_transform` (linhas 56–83) é a função principal do exemplo. Ela manipula os *pixels* do quadro de entrada transformando-os de acordo com a fórmula anterior, e produz um efeito de ondulação no quadro de saída. Observe que diferentemente da função `gst_my_filter_chain` da Listagem 8.12 que recebia um *buffer* (`GstBuffer`), a função `gst_my_video_transform` recebe e devolve (via parâmetros) quadros de vídeo (`GstVideoFrame`).

Finalmente, as linhas 114–125 são análogas às linhas correspondentes da Listagem 8.12. Ou seja, contém o ponto de entrada do *plugin* (linhas 114–118) e as declarações associadas (linhas 120–125).

Compilando e usando o elemento "myvideofilter"

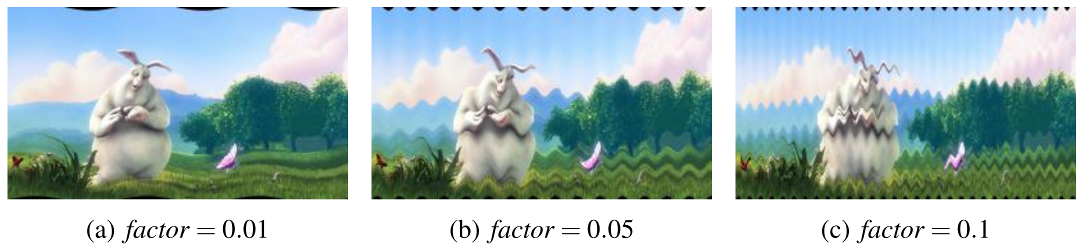
O comando seguinte compila e *link*-edita o *plugin* "myvideofilter" no GNU/Linux:

```
$ cc myvideofilter.c -o myvideofilter.so -shared -fPIC\
     `pkg-config --cflags --libs\
         gstreamer-1.0 gstreamer-base-1.0 gstreamer-video-1.0`
```

Observe que além dos módulos "gstreamer-1.0" e "gstreamer-base-1.0" o *plugin* "myvideofilter" depende do módulo "gstreamer-video-1.0", que contém cabeçalhos e bibliotecas para manipulação de *buffers* de vídeo.

O comando seguinte construí um *pipeline* que aplica o novo elemento ao vídeo *Big Buck Bunny*. O resultado do comando com diferentes valores para a propriedade "factor" é apresentado na Figura 8.10.

```
$ gst-launch --gst-plugin-path=.\
               uridecodebin uri="file://$PWD/bunny.ogg"\
               ! videoconvert\
               ! myvideofilter factor=.10\
               ! videoconvert\
               ! autovideosink
```



(a) *factor* = 0.01 (b) *factor* = 0.05 (c) *factor* = 0.1

Figura 8.10. Efeito de ondulação produzido pelo filtro "myvideofilter".

8.7. Conclusão

Neste minicurso apresentamos uma introdução ao *framework* multimídia GStreamer. Discutimos o modelo de computação *dataflow* e, em particular, a forma como esse modelo é instanciado no *framework*. Além disso, através de exemplos incrementais, apresentamos os principais conceitos da API C básica do GStreamer. Mais especificamente, mostramos como aplicações simples para reprodução de conteúdo multimídia podem ser programadas, e discutimos brevemente a criação de novos *plugins* contendo elementos processadores.

Apesar dos exemplos discutidos serem relativamente simples, com o que vimos já é possível que criar aplicações multimídia avançadas. Por exemplo, usando as mesmas APIs podemos usar elementos *mixers* (elementos "audiomixer" e "compositor") para compor diversos fluxos de áudio e vídeo num mesmo *pipeline*. Outra possibilidade é usar o *framework* para transcodificar fluxos de mídia, ou seja, montar *pipelines* que decodificam e re-codificam (em outro formato) um ou mais fluxos de mídia. Nesse caso, o *sink* "filesink" pode ser usado para escrever o fluxo resultante no disco.

Alguns tópicos avançados, porém, foram deliberadamente omitidos. No restante desta seção discutimos brevemente alguns desses tópicos com o objetivo de direcionar os leitores a aprofundamentos futuros.

Um dos temas desafiadores em sistemas multimídia é a transmissão e recepção de dados multimídia em tempo real. Embora não tratados aqui, o GStreamer possui diversos elementos que permitem transmitir e receber fluxos da rede. Por exemplo, o pacote *gst-plugins-good* contém elementos que podem ser usados para criar servidores e clientes RTP, RTCP e RTSP. No Gstreamer, um servidor RTP pode ser implementado como um *pipeline* que fragmenta fluxos codificados, encapsula esses fragmentos em pacotes RTP e transmite-os na rede via UDP, conforme ilustrado no *pipeline* da Figura 8.11a. Na figura, o elemento "rtptheorapay" recebe um fluxo de vídeo Theora e o encapsula em pacotes RTP que são transmitidos na rede via UDP pelo elemento seguinte, "udpsink". De forma análoga, podemos implementar um cliente RTP usando os elementos *udpsrc*, que recebe um fluxo de dados via UDP, e *rtptheorapay*, que desempacota um fluxo RTP e gera um fluxo codificado resultante, como ilustrado no *pipeline* da Figura 8.11b. Além de elementos que encapsulam e desencapsulam formatos específicos de fluxos em pacotes RTP, o GStreamer possui diversos outros elementos que facilitam a implementação desse tipo de aplicação, por exemplo, *rtpmanager*, *rtpbin*, *rtpjitterbuffer*, etc.

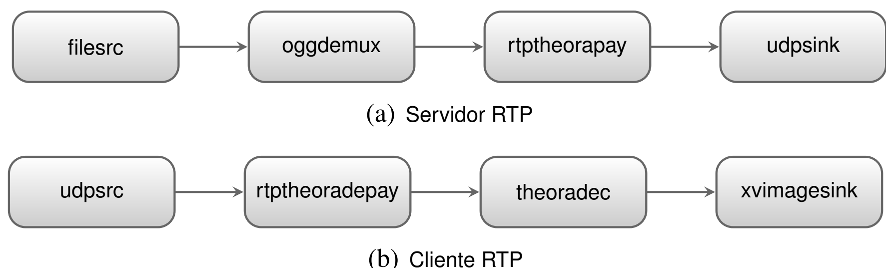


Figura 8.11. Servidor e cliente RTP para vídeos Ogg/Theora.

Outro tema importante que não discutimos é QoS (qualidade de serviço). De fato, mencionamos rapidamente eventos de QoS na Seção 8.1. Além desses eventos, o GStreamer possui outras funcionalidades que visam manter a QoS de aplicações multimídia. Por exemplo, internamente todo *pipeline* mantém um relógio (`GstClock`) que é usado para sincronizar a exibição dos *buffers* que o percorrem. Elementos *sink* usam esse relógio tanto para controlar a apresentação das amostras quanto para descartar amostras atrasadas. Dependendo da aplicação, se necessário, podemos sincronizar o relógio de *pipelines* diferentes (mesmo em máquinas separadas) via objetos `GstNetTimeProvider`, que expõe o tempo de um `GstClock` para a rede, ou `GstNetClientClock`, que sincroniza o relógio de um *pipeline* a um objeto provedor de tempo (`GstNetTimeProvider`), e podemos ainda criar relógios sincronizados com um servidor NTP (`GstNtpClock`) ou PTP (`GstPtpClock`). Dessa forma, é possível usar o GStreamer para implementar aplicações que requerem a sincronização de mídias em múltiplos destinos ou em múltiplas origens—cenários comuns em ambientes híbridos de TV digital [van Deventer et al., 2016].

Há certamente mais temas importantes que não abordamos. Todavia, acreditamos que a partir da base apresentada aqui os leitores estejam aptos a atacar esses temas. No site oficial do GStreamer [GStreamer, 2016] há diversos materiais detalhando o funcionamento interno do *framework*. Além disso, apesar de não ser essencial, àqueles que querem trabalhar diretamente no código do GStreamer, ou mesmo estendê-lo por meio de *plugins*, recomendamos a documentação do *framework* de objetos GObject [GLib, 2016a]. Finalmente, caso C não seja a sua linguagem favorita, existem *bindings* do GStreamer para outras linguagens e ambientes, por exemplo, Lua, Python, Java, C++, Qt, Android, Vala, Ruby, Haskell, etc.

Referências

[ALSA, 2016] ALSA (2016). Advanced Linux Sound Architecture (ALSA) Project Homepage. `http://www.alsa-project.org`. Acessado em 4 de outubro de 2016.

[Amatriain et al., 2008] Amatriain, X., Arumi, P., e Garcia, D. (2008). A framework for efficient and rapid development of cross-platform audio applications. *Multimedia Systems*, 14(1):15–32.

[Blender, 2008] Blender (2008). Big Buck Bunny. `http://peach.blender.org`. Acessado em 4 de outubro de 2016.

[Chatterjee e Maltz, 1997] Chatterjee, A. e Maltz, A. (1997). Microsoft DirectShow: A new media architecture. *SMPTE Motion Imaging Journal*, 106(12):865–871.

[GLib, 2016a] GLib (2016a). GLib reference manual. `https://developer.gnome.org/GLib`. Acessado em 4 de outubro de 2016.

[GLib, 2016b] GLib (2016b). GObject reference manual. `https://developer.gnome.org/GObject`. Acessado em 4 de outubro de 2016.

[GNOME, 2016] GNOME (2016). GNOME. `https://www.gnome.org`. Acessado em 4 de outubro de 2016.

[Gough, 2005] Gough, B. J. (2005). *An Introduction to GCC*. Network Theory Ltd, United Kingdom, rev. edition.

[GStreamer, 2016] GStreamer (2016). GStreamer: Open source multimedia framework. `http://gstreamer.freedesktop.org`. Acessado em 4 de outubro de 2016.

[GStreamer Developers, 2016] GStreamer Developers (2016). GStreamer apps. `https://gstreamer.freedesktop.org/apps/`. Acessado em 4 de outubro de 2016.

[ISO/IEC, 1993] ISO/IEC (1993). *ISO/IEC 11172-3:1993: Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1,5Mbit/s — Part 3: Audio*. International Organization for Standardization, Geneva, Switzerland.

[Kahn e MacQueen, 1977] Kahn, G. e MacQueen, D. B. (1977). Coroutines and networks of parallel processes. In Gilchrist, B., editor, *Proceedings of IFIP Congress 77, Toronto, Canada, 8–12 August, 1977*, pages 993–998, Amsterdam, The Netherlands. North-Holland Publishing Company.

[Kernighan e Ritchie, 1988] Kernighan, B. W. e Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition.

[Lee e Parks, 1995] Lee, E. A. e Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.

[Mecklenburg, 2005] Mecklenburg, R. (2005). *Managing Projects with GNU Make*. O'Reilly, Sebastopol, CA, USA, 3rd edition.

[Orlarey et al., 2009] Orlarey, Y., Fober, D., e Letz, S. (2009). FAUST: An efficient functional approach to DSP programming. In Assayag, G. e Gerzso, A., editors, *New Computational Paradigms for Computer Music*. Éditions Delatour Paris, Sampzon, France.

[Pfeiffer, 2003] Pfeiffer, S. (2003). The Ogg encapsulation format version 0. RFC 3533, RFC Editor.

[Puckette, 2007] Puckette, M. S. (2007). *The Theory and Technique of Electronic Music*. World Scientific Publishing Company, Singapore.

[Underbit, 2016] Underbit (2016). MAD: MPEG audio decoder. http://www.underbit.com/products/mad/. Acessado em 4 de outubro de 2016.

[van Deventer et al., 2016] van Deventer, M. O., Stokking, H., Hammond, M., Feuvre, J. L., e Cesar, P. (2016). Standards for multi-stream and multi-device media synchronization. *IEEE Communications Magazine*, 54(3):16–21.

[Wang e Cook, 2003] Wang, G. e Cook, P. R. (2003). ChucK: A concurrent, on-the-fly, audio programming language. In *Proceedings of the 2003 International Computer Music Conference, Singapore, 29 September–4 October, 2003*, pages 219–226, San Francisco, CA, USA. International Computer Music Association.

[Xiph.Org, 2011] Xiph.Org (2011). Theora specification. Xiph.org specification.

[Xiph.Org, 2015] Xiph.Org (2015). Vorbs I specification. Xiph.org specification.

[X.Org, 2016] X.Org (2016). X.Org. http://www.x.org. Acessado em 4 de outubro de 2016.

[Yviquel et al., 2015] Yviquel, H., Sanchez, A., Jääskeläinen, P., Takala, J., Raulet, M., e Casseau, E. (2015). Embedded multi-core systems dedicated to dynamic dataflow programs. *Journal of Signal Processing Systems*, 80(1):121–136.

Biografia Resumida dos Autores

Guilherme F. Lima é pesquisador associado do Laboratório TeleMídia da PUC-Rio. Seus interesses de pesquisa incluem linguagens de programação e modelos para sincronismo multimídia. Obteve o Doutorado em Informática pela PUC-Rio em 2015. Também possui Mestrado em Informática (2011) e Bacharelado em Sistemas de Informação (2009), ambos pela PUC-Rio.

Rodrigo C. M. Santos é doutorando em Informática na PUC-Rio. Mestre em Ciência da Computação pela Universidade Federal do Maranhão (2013). Atualmente é pesquisador do laboratório TeleMídia da PUC-Rio e colaborador do laboratório LAWS/UFMA, tendo como principais interesse de pesquisa: sistemas multimídia distribuídos, sincronização de mídia em diferentes dispositivos e linguagens reativas síncronas.

Roberto G. de A. Azevedo é pesquisador associado do Laboratório TeleMídia da PUC-Rio. Possui doutorado (2015) e mestrado (2010) em Informática pela PUC-Rio e é Bacharel em Ciência da Computação pela Universidade Federal do Maranhão (2008). Seus interesses de pesquisa incluem: representação e autoria de cenas multimídia interativas; e representação, codificação, transmissão e renderização de vídeos 3D.

Índice de Autores

A
Amorim, Marcello N. de 66
Antonelli, Humberto L. 37
Azevedo, Roberto G. de A. 215

B
Brandão, Rafael 181

C
Cerqueira, Renato 181
Cunha, Bruna C. R. da 119

F
Fortes, Renata P. M. 37

G
Gonçalves, João Carlos 155

K
Kuniwake, Júlio T. 155

L
Lima, Guilherme F. 215

M
Machado Neto, Olibário J. 119
Magagnatto, Yuri N. Z. G. 119
Moreno, Marcio F. 181
Muchaluat-Saade, Debora 1

O
Oliveira, Cintia C. 155
Oliveira, Daniele C. 155
Orlando, Alex F. 119

P
Pimentel, Maria da G. C. 119

R
Rocha, André C. 119
Rodrigues, Kamila R. H. 119

S
Salgado, André de L. 37
Santos, Celso A. S. 66
Santos, Joel dos 1
Santos, Rodrigo C. M. 215
Santos, Rodrigo 93
Segundo, Ricardo M. C. 66

V
Viana, Davi 93
Viel, Caio C. 119

Z
Zaine, Isabela 119