

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

PINTEREST, INC.

Petitioners,

v.

OPENTV, INC.

Patent Owner.

Case No. IPR2026-00365

U.S. Patent No. 11,825,139

**PETITION FOR *INTER PARTES* REVIEW
OF U.S. PATENT NO. 11,825,139**

TABLE OF CONTENTS

	<u>Page</u>
I. INTRODUCTION	1
II. MANDATORY NOTICES.....	1
A. Real Party-in-Interest (37 C.F.R. § 42.8(b)(1))	1
B. Related Matters (37 C.F.R. § 42.8(b)(2)).....	1
1. Related Patent Office Proceedings.....	1
2. Related Litigation.....	1
C. Lead and Backup Counsel and Service Information (37 C.F.R. §§ 42.8(b)(3)-(4)).....	1
D. Payment of Fees (37 C.F.R. § 42.15(a))	2
E. Requirements For <i>Inter Partes</i> Review (37 C.F.R. §§ 42.101(A)-(C), 42.104(A), and 42.018).....	2
III. GROUNDS	3
IV. BACKGROUND	3
A. '139 Patent	3
1. Prosecution History.....	4
B. Summary of References.....	4
1. Orr.....	4
2. Wu.....	5
3. Hsu	5
4. Lim.....	5
5. Hervey (EX1009).....	6
V. PERSON OF ORDINARY SKILL.....	12
VI. CLAIM CONSTRUCTION.....	12
VII. GROUND 1: ORR/WU	13
A. Motivation to Combine	13
1. Claim 1	18
2. Claim 2.....	34
3. Claim 3.....	36
4. Claim 4.....	37

Petition for *Inter Partes* Review
U.S. Patent No. 11,825,139

5.	Claim 8.....	37
6.	Claim 9.....	39
7.	Claim 10.....	40
8.	Claim 11.....	41
9.	Claim 12.....	42
10.	Claim 14.....	43
11.	Claim 15.....	45
12.	Claim 16.....	45
13.	Claim 17.....	45
14.	Claim 18.....	45
15.	Claim 19.....	45
VIII.	GROUND 2: ORR/WU/HSU	45
A.	Motivation to Combine	45
1.	Claim 6.....	48
2.	Claim 7.....	50
3.	Claim 13.....	51
IX.	GROUND 3: ORR/WU/LIM.....	52
A.	Motivation to Combine	52
1.	Claim 5.....	55
X.	GROUND 4: HERVEY	57
1.	Claim 1.....	57
2.	Claim 2.....	70
3.	Claim 3.....	71
4.	Claim 4.....	72
5.	Claim 8.....	74
6.	Claim 9.....	78
7.	Claim 10.....	79
8.	Claim 11.....	80
9.	Claim 12.....	81
10.	Claim 14.....	81
11.	Claim 15.....	84
12.	Claim 16.....	84

13.	Claim 17.....	84
14.	Claim 18.....	84
15.	Claim 19.....	84
XI.	GROUND 5: HERVEY/HSU	85
A.	Motivation to Combine	85
1.	Claim 6.....	87
2.	Claim 7.....	87
3.	Claim 13.....	87
XII.	GROUND 6: HERVEY/LIM.....	88
A.	Motivation to Combine	88
1.	Claim 5.....	90

EXHIBIT LIST

Exhibit	Description
EX1001	U.S. Patent 11,825,139 (“139 Patent”)
EX1002	Declaration of Immanuel Freedman, Ph.D.
EX1003	Declaration of Edward Hervey
EX1004	File History of U.S. Patent 11,825,139
EX1005	U.S. Patent Application Publication 2009/0132725 (“Orr”)
EX1006	Chinese Patent Application No. CN 106060628 A and certified translation
EX1007	U.S. Patent 8,516,144 (“Hsu”)
EX1008	U.S. Patent 9,628,543 (“Lim”)
EX1009	“ <i>decodebin3, Modern playback use-cases</i> ” presentation by Edward Hervey at the GStreamer Conference 2015, October 8 th , 2015 (“Hervey”) available at https://web.archive.org/web/20170226222859/https://gstreamer.freedesktop.org/data/events/gstreamer-conference/2015/Edward%20Hervey%20-%20decodebin3.pdf
EX1010	Declaration of Mina Ching
EX1011	“GStreamer Conference 2015,” available at https://web.archive.org/web/20160106210441/http://gstreamer.freedesktop.org/data/events/gstreamer-conference/2015/
EX1012	“Index of data/events/gstreamer-conference/2015,” available at https://web.archive.org/web/20160109011525/http://gstreamer.freedesktop.org/conference/2015/index.html
EX1013	“GStreamer Conference 2015, Schedule,” available at https://web.archive.org/web/20161018073419/https://gstreamer.freedesktop.org/conference/2015/schedule.html
EX1014	“GStreamer Conference 2015, Talks and Speakers,” https://web.archive.org/web/20160125231357/http://gstreamer.freedesktop.org/conference/2015/talks-and-speakers.html

Exhibit	Description
EX1015	LWN.net article “Upcoming work in and around GStreamer,” https://web.archive.org/web/20160330004434/https://lwn.net/Articles/661403/
EX1016	U.S. Pat. No. 8,289,370
EX1017	U.S. Pat. App. Pub. No. 2015/0104158
EX1018	“GstCaps” available at https://web.archive.org/web/20160111144202/http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/GstCaps.html
EX1019	“GstTypeFind” available at https://web.archive.org/web/20151026053357/http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/gstreamer-GstTypeFind.html
EX1020	Curriculum Vitae of Immanuel Freedman, Ph.D.

LIST OF CHALLENGED CLAIMS

Claim	
1[pre]	A method comprising:
1[a]	causing playback of a first piece of content using a playback pipeline;
1[b]	receiving, by a digital receiver, a selection of a new piece of content for playback during playback of a first piece of content;
1[c]	in response to the receiving of the selection, transitioning from playback of the first piece of content to the new piece of content;
1[d]	determining whether the new piece of content is of a same content type as the first piece of content; and
1[e]	based on the new piece of content being the same content type as the first piece of content, determining whether an audio decoder or a video decoder of the playback pipeline used to play back the first piece of content should be preserved; and
1[f]	based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer of the playback pipeline while replacing the audio decoder and the video decoder.
2[pre]	The method of claim 1, wherein the determining whether the audio decoder and the video decoder should be preserve comprises:
2[a]	detecting codec information for the new piece of content, the codec information indicating types of decoders needed for the new piece of content; and
2[b]	determining whether the audio decoder and the video decoder in the playback pipeline are of the types indicated by the codec information.
3	The method of claim 2, further comprising preserving an entirety of the playback pipeline in response to the audio decoder and the video decoder in the playback pipeline being the types indicated by the codec information.
4[pre]	The method of claim 2, further comprising:
4[a]	in response to the audio decoder and the video decoder in the playback pipeline being different than the types indicated by the codec information,
4[b]	obtaining a new audio decoder and a new video decoder based on the codec information.

Claim	
5	The method of claim 2, wherein the detecting the codec information for the new piece of content comprises examining prefetched metadata for the new piece of content.
6	The method of any one of claim 1, wherein: the transitioning from playback of the first piece of content to the new piece of content comprises initiating playback of the new piece of content using a bitrate used for the playback of the first piece of content; and the determining whether to preserve occurs during playback of the new piece of content using the bitrate.
7[pre]	The method of claim 6, further comprising:
7[a]	during playback of the new piece of content using the bitrate, detecting whether the new piece of content can maintain playback at the bitrate; and
7[b]	in response to the detecting that the new piece of content cannot continue playback at the bitrate, adjusting the bitrate.
8[pre]	The method of claim 1, further comprising:
8[a]	based on the first piece of content and the new piece of content not being the same content type, deconstructing the playback pipeline by releasing the source element, the demultiplexer, the audio decoder, and the video decoder; and
8[b]	creating a new playback pipeline by acquiring other resources that are applicable to a content type of the new piece of content.
9	The method of claim 1, wherein content types include HTTP Live Streaming (HLS) format and Dynamic Adaptive Streaming over HTTP (DASH) format.
10	The method of claim 1, wherein the receiving the selection, transitioning to the new piece of content, determining whether the new piece of content is of the same content type as the first piece of content, and determining whether to preserve is performed by a player.
11[pre]	The method of claim 1, further comprising:
11[a]	based on the first piece of content and the new piece of content not being the same content type, releasing the source element; and
11[b]	creating a new playback pipeline by acquiring a different source element that is applicable to a content type of the new piece of content.

Claim	
12[pre]	The method of claim 1, further comprising:
12[a]	based on the first piece of content and the new piece of content not being the same content type, releasing the demultiplexer; and
12[b]	creating a new playback pipeline by acquiring a different demultiplexer that is applicable to a content type of the new piece of content.
13	The method of claim 1, wherein the source element initiates the playback of the new piece of content using a same bitrate as the first piece of content and adjusts the bitrate in response to detecting that the new piece of content cannot continue playback at the bitrate.
14[pre]	A system comprising:
14[a]	one or more hardware processors; and
14[b]	a storage device storing instructions that configure the one or more hardware processors to perform operations comprising:
14[c]	causing playback of a first piece of content using a playback pipeline;
14[d]	receiving a selection of a new piece of content for playback during playback of the first piece of content;
14[e]	in response to the receiving of the selection, transitioning from playback of the first piece of content to the new piece of content;
14[f]	determining whether the new piece of content is of a same content type as the first piece of content;
14[g]	based on the new piece of content being the same content type as the first piece of content, determining whether an audio decoder and a video decoder of the playback pipeline used to play back the first piece of content should be preserved; and
14[h]	based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer of the playback pipeline while replacing the audio decoder and the video decoder.
15[pre]	The system of claim 14, wherein the determining whether the audio decoder and the video decoder should be preserved comprises:
15[a]	detecting codec information for the new piece of content, the codec information indicating types of decoders needed for the new piece of content; and

Claim	
15[b]	determining whether the audio decoder and the video decoder in the playback pipeline are of the types indicated by the codec information.
16	The system of claim 15, wherein the operations further comprise: preserving an entirety of the playback pipeline in response to the audio decoder and the video decoder in the playback pipeline being the types indicated by the codec information.
17[pre]	The system of claim 15, wherein the operations further comprise:
17[a]	in response to the audio decoder and the video decoder in the playback pipeline being different than the types indicated by the codec information,
17[b]	obtaining a new audio decoder and a new video decoder based on the codec information.
18[pre]	The system of claim 14, wherein the operations further comprise:
18[a]	based on the first piece of content and the new piece of content not being the same content type, deconstructing the playback pipeline by releasing the source element, the demultiplexer, the audio decoder, and the video decoder; and
18[b]	creating a new playback pipeline by acquiring other resources that are applicable to a content type of the new piece of content.
19[pre]	A machine-storage medium storing instructions that when executed by one or more hardware processors of a machine, cause the machine to perform operations comprising:
19[a]	causing playback of a first piece of content using a playback pipeline;
19[b]	receiving a selection of a new piece of content for playback during playback of the first piece of content;
19[c]	in response to the receiving of the selection, transitioning from playback of the first piece of content to the new piece of content;
19[d]	determining whether the new piece of content is of a same content type as the first piece of content;
19[e]	based on the new piece of content being the same content type as the first piece of content, determining whether an audio decoder of the playback pipeline used to play back the first piece of content should be preserved; and
19[f]	based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and

Claim	
	a demultiplexer of the playback pipeline while replacing the audio decoder and the video decoder.

I. INTRODUCTION

Petitioner Pinterest (“Petitioner”) requests *Inter Partes* Review of claims 1-19 of U.S. Patent No. 11,825,139 (“’139 patent”) assigned to OpenTv, Inc. (“Patent Owner”). The ’139 patent is directed to “[s]ystems and methods for optimizing a content change process” in which during playback of first content, new content is received, and its content type is compared to determine if the original audio and video decoders should be preserved. (EX1001, Abstract, Claim 1.) As demonstrated below, playback pipeline preservation was ubiquitous in the prior art, and the ’139 patent claims no non-obvious features or implementation.

II. MANDATORY NOTICES

A. Real Party-in-Interest (37 C.F.R. § 42.8(b)(1))

The real parties-in-interest for Petitioner are Pinterest, Inc.

B. Related Matters (37 C.F.R. § 42.8(b)(2))

1. Related Patent Office Proceedings

The ’139 patent is in the same family as U.S. Patent Nos. 11,184,652 and 12,231,703.

2. Related Litigation

There is presently no related litigation.

C. Lead and Backup Counsel and Service Information (37 C.F.R. §§ 42.8(b)(3)-(4))

Electronic service may be made on the email addresses identified below and

in the accompanying Power of Attorney.

LEAD COUNSEL	BACK-UP COUNSEL
James M. Glass (Reg. No. 46,729) jimglass@quinnemanuel.com QUINN EMANUEL URQUHART & SULLIVAN, LLP 51 Madison Avenue, 22 nd Floor New York, NY 10010 Tel: (212) 849-7000	Quincy Lu (Reg. No. 76,954) quincylu@quinnemanuel.com QUINN EMANUEL URQUHART & SULLIVAN, LLP 1109 First Avenue, Suite 210 Seattle, WA 98101 Tel: (206) 905-7000 Ron Hagiz (<i>pro hac vice</i> forthcoming) ronhagiz@quinnemanuel.com QUINN EMANUEL URQUHART & SULLIVAN, LLP 295 5 th Avenue, 9th Floor New York, NY 10016 Tel: (212) 849-7000

D. Payment of Fees (37 C.F.R. § 42.15(a))

The Office is authorized to charge the fee required for this Petition (and any additional fees) to Deposit Account No. 50-5708.

E. Requirements For *Inter Partes* Review (37 C.F.R. §§ 42.101(A)-(C), 42.104(A), and 42.018)

Petitioner certifies that the '139 patent is available for *inter partes* review, that Petitioner is not barred or estopped from challenging the '139 claims on the grounds identified herein, and that the prohibitions of 35 U.S.C. §§ 315(a)-(b) are

inapplicable.

III. GROUNDS

Ground	Basis	References	Claims
1	§103	Orr, Wu	1-4, 9-12, 15-17, 19
2	§103	Orr, Wu, Hsu	6-8, 13-14
3	§103	Orr, Wu, Lim	5
4	§103	Hervey	1-4, 9-12, 15-17, 19
5	§103	Hervey, Hsu	6-8, 13-14
6	§103	Hervey, Lim	5

IV. BACKGROUND

A. '139 Patent

The '139 patent is states that content switch “typically causes a previous content delivery pipeline to be deconstructed (e.g., resources previously acquired are released),” requiring an entirely new pipeline. (EX1001, 2:55-58.) (EX1002, ¶¶23-26.) If both formatting type and decoding type are the same, “the entire pipeline 204 is preserved.” (EX1001, 8:17–20.) (EX1002, ¶¶27-29.)

If formatting types match but different decoders are required, new decoders are reacquired while “the source element 212 and the demux 214[] are preserved.” EX1001, 8:27-33.) (EX1002, ¶30.)

1. Prosecution History

The original claims of the '139 patent only required preservation of a portion of the pipeline if the content type matched. (EX1004, Nov. 9, 2021 Claims.) (EX1002, ¶¶32-36.)

In response to a rejection, claims were amended to add that when the audio/video decoders are not preserved, the source element and demultiplexer are nevertheless preserved. (EX1004, Mar. 22, 2023 Amendment and Remarks.)

Following the amendment, the claims were allowed.

B. Summary of References

1. Orr

Orr is directed to a channel switching system that preserves a pipeline when content types match. (EX1005, Abstract; ¶[0003–05].) Orr states that, conventionally, when a channel change is detected, the player tears down the existing rendering pathway and constructs a new one, a process that “may take more than a few seconds” and produces delays, blank screens, and distorted images. (EX1005, ¶[0004].)

Orr’s solution is to determine whether the new channel’s content is compatible with the existing rendering pathway and, if so, to feed the new content directly into the existing pathway without reinitialization. (EX1005, ¶[0015]; ¶[0021-22].) (EX1002, ¶¶44-45.)

2. Wu

Wu is directed to seamless decoding of audio and video when the codec of the audio/video change. Wu teaches that “if encoding parameters change ..., it is clearly not feasible to decode and play using the original decoder.” (EX1006, ¶[0005].) Upon detecting the decoding format has changed, “the current audio and video decoders” are “[d]isconnect[ed],” and new audio/video decoders are “create[d] ... based on the aforementioned audio and video decoder parameters.” (*Id.*, ¶¶[0016], [0028], [0079]-[0080].) Wu teaches that its demultiplexer does not change. (EX1002, ¶46.)

3. Hsu

Hsu is directed to an adaptive bitrate streaming media player that tracks can default to the current bitrate when playing new content. (EX1007; Abstract, 5:28-40, 6:1-30.) Hsu states that conventional systems “default to the lowest bitrate in the dynamic profile set,” resulting in users “playing a lower quality video than what the current bandwidth to that device allows,” with the time to reach optimal quality taking “as long as fifteen (15) seconds.” (EX1007; 5:28-40.) (EX1002, ¶47.)

4. Lim

Lim is directed to prefetching content metadata for content the user has not yet selected, based on navigational context during playback of current content. (EX1008; Abstract, claim 1, [0026–36.].) (EX1002, ¶48.)

5. Hervey (EX1009)

Hervey is a PowerPoint document that was publicly presented by Edward Hervey at the GStreamer Conference 2015 on October 8th, 2015 in Dublin, Ireland. (EX1002, ¶¶49-52.)

a) Hervey is Authentic

Hervey (EX1009) was presented on October 8th, 2015 at the GStreamer Conference 2015 in Dublin, Ireland. According to Mr. Hervey, the author and presenter, Exhibit 2009 is a true and authentic copy of the document presented. EX1003.

Ms. Ching of the Internet Archive confirms that Exhibit 1009 is an authentic copy of the presentation made publicly available on the Internet by February 26, 2017. EX1010, 1.

b) Hervey Qualifies as a Printed Publication

As explained below, Hervey is a printed publication because, *first*, it was publicly available on the Internet by February 26, 2017, and, *second*, it was presented in 2015 at the GStreamer Conference.

(1) *Hervey Was Publicly Available By February 26, 2017*

Hervey was publicly accessible and available by February 26, 2017.

According to Ms. Ching, the Wayback Machine archives webpages “by use of web archiving software that automatically stores copies of files available via the

Internet.” (EX1010, ¶4.) Each Internet Archive URL contains the date and time when the webpage was archived. (*Id.*, ¶5.) Accordingly, per Ms. Ching, Hervey was publicly accessible and available by February 26, 2017, when the Wayback Machine archived the link.

Dr. Freedman, who has over 30 years’ experience with video imaging and compression and authored the “Video Compression” chapter for the *Internet Encyclopedia*, is familiar with the Wayback Machine and agrees with Ms. Ching that the Internet Archive exhibits are archives of publicly accessible and available webpages at the time they were archived, and that those exhibits are true and correct copies. EX1002, ¶¶53-54. Moreover, the Federal Circuit has found that the Wayback Machine is inherently reliable and “highly persuasive” on the issue of public accessibility. *Valve Corp. v. Ironburg Inventions, Ltd.*, 8 F.4th 1364, 1374-75 (Fed. Cir. 2021).

(2) *Hervey Was Presented to the Public On October 8, 2015*

According to the author, Mr. Hervey, it was presented to the public on October 8, 2015 at the 2015 GStreamer Conference, with no obligation of confidentiality. Numerous websites that were contemporaneously available confirm that the conference took place in 2015 and was open to the public. According to Ms. Ching, these websites were archived by the Wayback Machine at around the same time the

conference took place. EX1011-EX1015; EX1010.

The conference is well-known and attended by practitioners in the multimedia fields, and its proceedings are routinely consulted by ordinary artisans (EX1011).

The conference invited “[a]nyone interested in open source cross-platform multimedia.” In addition, registration was open to the public. (EX1011.)

Who should attend?

Anyone interested in open source cross-platform multimedia is encouraged to join us at the GStreamer Conference 2015. Whether you are a hobbyist or a student interested in the field of multimedia, or working for a company using or looking to use GStreamer in their products, this is the perfect event for you to network at and to learn more about the latest technological breakthroughs in and around GStreamer. The talks will target both developers and managers.

Registration

Registration is now open and will stay open until the conference.

[Sign up for the GStreamer Conference here](#)

There will be separate registration options for both professionals and for students and hobbyists:

Professional	275 USD incl. lunch (until 31 July)
Professional	385 USD incl. lunch (from 1 August)
Student/hobbyist	100 USD incl. lunch

The registration fee includes continental breakfast and lunch at the venue on both conference days.

For those uncertain about whether to register as a professional or hobbyist, the general rule is that if your company pays for you or sponsors you, you are a professional, and if you personally pay you are a student or hobbyist. If in doubt drop us a mail or just register as hobbyist. The important thing is that you'll make it to the conference!

Hervey's presentation was scheduled for October 8, 2015 at 11:20am. The below snapshot was archived on October 18, 2016. (EX1012.)

Petition for *Inter Partes* Review
U.S. Patent No. 11,825,139

GStreamer Conference 2015 Schedule			
Dublin, Ireland, 8-9 October 2015			
Back to conference main page			
Schedule for Thursday 8 October 2015			
Schedule for Friday 9 October 2015			
Thursday 8th of October 2015			
	Liffey Meeting Room 2a	Liffey Meeting Room 2b	
08.30	Registration and Breakfast Buffet		08.30
09.00-09.10	Opening and welcome to the 2015 GStreamer Conference Liffey Meeting Room 2		09.00-09.10
09.10-09.40	Keynote GStreamer State of the Union Tim-Philipp Müller, Centricular Liffey Meeting Room 2		09.10-09.40
09.40-10.00	mini break		09.40-10.00
10.00-10.30	Tiled Streaming of UHD video in real-time Arjen Veenhuizen, TNO	Kurento Media Server: experiences bringing GStreamer capabilities to WWW developers José Antonio Santos, Kurento	10.00-10.30
10.35-10.55	GStreamer-VAAPI: Hardware-accelerated encoding and decoding on Intel hardware Victor M. Jaquez L., Igalia	ogv.js: bringing open codecs to Safari and IE with emscripten Brion Vibber, Wikimedia	10.35-10.55
10.55-11.20	Coffee Break		10.55-11.20
11.20-12.05	Decodebin3, or dealing with modern playback use-cases Edward Hervey (bilboed), Centricular	Chromium: a new media Backend based on GStreamer Julien Iserce, Samsung	11.20-12.05
12.10-12.30	GstPlayer - A simple cross-platform API for all your media playback needs Sebastian Dröge (slomo), Centricular	GStreamer and WebKit Philippe Normand (philn), Igalia	12.10-12.30
12.30-13.30	Lunch Break: Buffet Lunch for GStreamer Conference attendees		12.30-13.30
13.30-14.00	Going Live with ToolsOnAir's GStreamer-based broadcast mixing architecture Heinrich Fink, ToolsOnAir	The HeliosTv Distributed DVB stack Romain Picard, Soft@Home	13.30-14.00
14.05-14.25	How to Contribute to GStreamer Luis de Bethencourt (luisbg), Samsung	Stereoscopic 3D Video in GStreamer Redux Jan Schmidt (thaytan), Centricular	14.05-14.25
14.25-14.45	mini break		14.25-14.45
14.45-15.15	Faster transcoding with GStreamer in the cloud Thiago Sousa Santos, Samsung	Colorspaces and HDMI Hans Verkuil, Cisco	14.45-15.15
15.15-15.40	Coffee Break		15.15-15.40
15.40-16.10	Demystifying the allocation query Nicolas Dufresne (stormer), Collabora	Video Filters and their Applications Sanjay Narasimha Murthy, Samsung	15.40-16.10
16.15-16.45	Camera Sharing and Sandboxing with Pinos Wim Taymans (wtay), RedHat	Hardware-accelerated multimedia on TI's Jacinto 6 SoC Pooja Prajod, Texas Instruments	16.15-16.45
16.45-17.05	mini break		16.45-17.05
17.05-18.15	Lightning Talks - part 1 Liffey Meeting Room 2		17.05-18.15
18.15	End of Day		18.15
	Social Event: Drinks and Food at the Toners Pub , 139 Lower Baggot Street, Dublin 2, D02 N231		

The Talks and Speakers page (EX1013) for the 2015 GStreamer Conference lists Edward Hervey's presentation.

Decodebin3, or dealing with modern playback use-cases, Edward Hervey (bilboed), Centricular

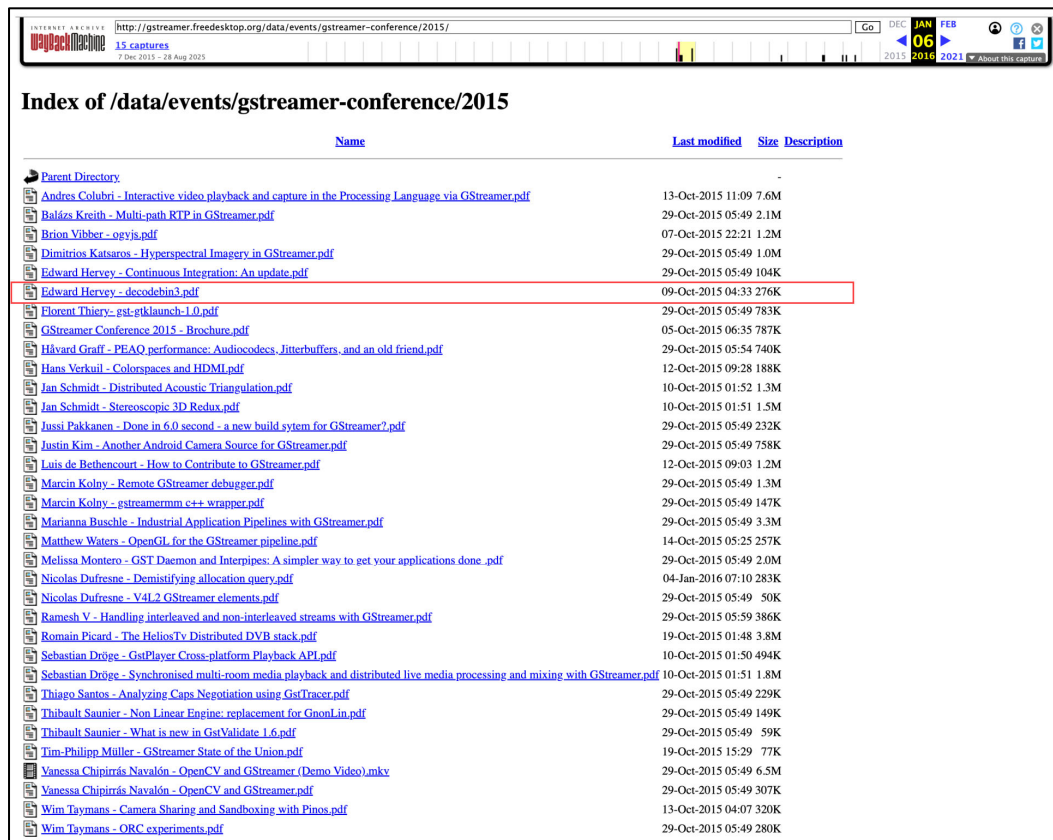
decodebin2 (decodebin in 1.x) is 10 years old. While it did serve its purpose, there are a number of limitations in regards to handling modern use-cases, features that are in playbin that should be present in decodebin, non-optimal memory usage, and so forth. This talk might include apologies from the original decodebin2 author *cough*.

After looking at the current design and limitation of stream handling in general, and decodebin2 in particular, we will look at a proposal for a new way of handling stream listing, stream selection, how this help with having predictable behaviour and drop all 0.10-ism that still remain in decodebin2.

The new design will allow reducing cpu/memory usage by only using the decoders actually needed, re-use decoders when switching streams (where possible), and better deal with adaptive streaming and dynamic MPEG-TS use-cases in general, amongst other things.

Edward Hervey has been contributing for over 12 years to GStreamer, ending up there after starting the PiTiVi video editor and then maintaining various components over the years. After having started Collabora Multimedia in 2007, attempting to go on sabbatical, and doing various freelancing, Edward Hervey is currently a part-time consultant for Centricular.

The conference page made Hervey's presentation available on its website.
(EX1011.) As shown below, the presentation was last modified on October 9, 2015
(the day after his presentation). (EX1014.)



Name	Last modified	Size	Description
Parent Directory	-	-	-
Andres Colubri - Interactive video playback and capture in the Processing Language via GStreamer.pdf	13-Oct-2015 11:09	7.6M	
Balázs Kreith - Multi-path RTP in GStreamer.pdf	29-Oct-2015 05:49	2.1M	
Brion Vibber - ogvjs.pdf	07-Oct-2015 22:21	1.2M	
Dimitrios Katsaros - Hyperspectral Imagery in GStreamer.pdf	29-Oct-2015 05:49	1.0M	
Edward Hervey - Continuous Integration: An update.pdf	29-Oct-2015 05:49	104K	
Edward Hervey - decodebin3.pdf	09-Oct-2015 04:33	276K	
Florent Thiery - gst-rtklaunch-1.0.pdf	29-Oct-2015 05:49	783K	
GStreamer Conference 2015 - Brochure.pdf	05-Oct-2015 06:35	787K	
Håvard Graff - PEAQ performance: Audio codecs, Jitter buffers, and an old friend.pdf	29-Oct-2015 05:54	740K	
Hans Verkuil - Colorspaces and HDMI.pdf	12-Oct-2015 09:28	188K	
Jan Schmidt - Distributed Acoustic Triangulation.pdf	10-Oct-2015 01:52	1.3M	
Jan Schmidt - Stereoscopic 3D Redux.pdf	10-Oct-2015 01:51	1.5M	
Jussi Pakkanen - Done in 6.0 second - a new build system for GStreamer?.pdf	29-Oct-2015 05:49	232K	
Justin Kim - Another Android Camera Source for GStreamer.pdf	29-Oct-2015 05:49	758K	
Luis de Bethencourt - How to Contribute to GStreamer.pdf	12-Oct-2015 09:03	1.2M	
Marcin Kolny - Remote GStreamer debugger.pdf	29-Oct-2015 05:49	1.3M	
Marcin Kolny - gstreamermm-c++ wrapper.pdf	29-Oct-2015 05:49	147K	
Marianna Buschle - Industrial Application Pipelines with GStreamer.pdf	29-Oct-2015 05:49	3.3M	
Matthew Waters - OpenGL for the GStreamer pipeline.pdf	14-Oct-2015 05:25	257K	
Melissa Montero - GST Daemon and Interpipes: A simpler way to get your applications done .pdf	29-Oct-2015 05:49	2.0M	
Nicolas Dufresne - Demystifying allocation query.pdf	04-Jan-2016 07:10	283K	
Nicolas Dufresne - V4L2 GStreamer elements.pdf	29-Oct-2015 05:49	50K	
Ramesh V - Handling interleaved and non-interleaved streams with GStreamer.pdf	29-Oct-2015 05:59	386K	
Romain Picard - The HeliosTV Distributed DVB stack.pdf	19-Oct-2015 01:48	3.8M	
Sebastian Dröge - GstPlayer Cross-platform Playback API.pdf	10-Oct-2015 01:50	494K	
Sebastian Dröge - Synchronised multi-room media playback and distributed live media processing and mixing with GStreamer.pdf	10-Oct-2015 01:51	1.8M	
Thiago Santos - Analyzing Caps Negotiation using GstTracer.pdf	29-Oct-2015 05:49	229K	
Thibault Saunier - Non-Linear Engine: replacement for GnonLin.pdf	29-Oct-2015 05:49	149K	
Thibault Saunier - What is new in GstValidate 1.6.pdf	29-Oct-2015 05:49	59K	
Tim-Philipp Müller - GStreamer State of the Union.pdf	19-Oct-2015 15:29	77K	
Vanessa Chipirrás Navalón - OpenCV and GStreamer (Demo Video).mkv	29-Oct-2015 05:49	6.5M	
Vanessa Chipirrás Navalón - OpenCV and GStreamer.pdf	29-Oct-2015 05:49	307K	
Wim Taymans - Camera Sharing and Sandboxing with Pinos.pdf	13-Oct-2015 04:07	320K	
Wim Taymans - ORC experiments.pdf	29-Oct-2015 05:49	280K	

Thus, Hervey is a printed publication because it was presented publicly at the
GStreamer Conference 2015.

c) Hervey Declaration

Mr. Hervey is the author of Exhibit 1009 and has direct, personal knowledge regarding his presentation. He declares that he presented Hervey at the 2015 GStreamer Conference. (EX1003, ¶4)). His presentationpresentation “was part of an open session of the conference and many industry participants attended.” (*Id.*, ¶5). Copies of Hervey were “freely distributed without confidentiality restrictions to attendees at the conference in October of 2015.” (*Id.*, ¶6). And, “[t]he presentation was also made available on the internet on GStreamer’s website shortly after the conference in 2015.” (*Id.*, ¶6). The Hervey presentation was reported on contemporaneously and video recordings of the presentation were publicly available on the GStreamer website shortly after the conference. ((*Id.*, ¶7).

d) Contemporaneous Reporting

Shortly after the GStreamer 2015 Conference, the Hervey presentation was reported on by Linux Weekly News (LWN). (EX1015.)



User: Password:

Upcoming work in and around Gstreamer

Benefits for LWN subscribers

The primary benefit from [subscribing to LWN](#) is helping to keep us publishing, but, beyond that, subscribers get immediate access to all site content and access to a number of extra site features. Please sign up today!

By **Nathan Willis**
October 21, 2015

[GStreamer Conference](#)

The 2015 GStreamer Conference included several talks that introduced new in-development features or ideas that are experimental in nature. They include a complete redesign of the automatic-decoding element, support for distributing broadcast television, and support for the WebRTC streaming protocol. Some of the features are being developed by members of the core GStreamer development team, while others originate in outside projects. As it frequently does, though, the conference allowed for these distinct teams to put their heads together.

decodebin3

Edward Hervey of Centricular presented his plans for a significant rewrite of the [decodebin](#) element, to get feedback from the GStreamer community before proceeding further. In GStreamer, the *-bin* suffix is used to designate an "auto-plugging" element that tries to recursively construct a processing pipeline to do the right thing for a common task. The decodebin element takes a media stream as input, figures out the stream type, and sets up the correct demultiplexing, parsing, and decoding elements for the content within the stream.

The trouble is that the current implementation, decodebin2, was written in 2006—when, Hervey said, "I didn't know the MPEG spec backward." So there are inefficiencies, such as the fact that decodebin2 decodes every media stream in the container file, even if only a subset of the streams are actually being played (say, one out of four available audio tracks). And switching between streams was convoluted. To continue the multiple-language-track example from above, switching from the English to the French track required setting up a new set of connections for the French audio *and* the video, emitting a signal to stop the current video and English audio streams, then restarting playback with the new setup. Worse still, if the user tried to do a seek event while this swap-over was being performed, they would jump to a position with no media available at all.

But these problems are magnified by more recent streaming protocols like HLS/DASH. A server might offer a selection of dozens of audio streams at a single URL; decodebin2 provides no way to inspect what each stream contains (even simple attributes like language tags), and decoding every stream would likely mean excessive overhead.

Hervey's plan is to write a new decodebin centered around a new, generic "stream" API. First, there will be a high-level GstStream object that makes important metadata (like video resolution or language tags) queryable. Next, multiple streams will be collected in an immutable GstStreamCollection

V. PERSON OF ORDINARY SKILL

A person of ordinary skill in the art as of the asserted priority date (“POSITA”) had at least a Bachelor of Science in electrical engineering or a similar field, such as Physics, and at least two years of practical experience in the field of video streaming applications. More education can supplement for less practical experience, and vice versa. (EX1002, ¶¶37-39.)

Dr. Freedman met this level by the priority date. (EX1002, ¶40.)

VI. CLAIM CONSTRUCTION

Claim terms should be given their plain and ordinary meaning in view of the

intrinsic evidence. (EX1002, ¶¶41-43.) The challenged claims have not been construed in other proceedings.

VII. GROUND 1: Orr/Wu

A. Motivation to Combine

A POSITA would have been motivated to modify Orr to incorporate Wu’s teachings of preserving a demultiplexer while using a new audio and video decoder and would have had a reasonable expectation of success in doing so. (EX1002, ¶¶55-63.) A POSITA would have been motivated by the known benefits of reducing latency and increasing processing efficiency when reusing a demultiplexer, as otherwise creating a new demultiplexer requires time and resources. (*Id.*)

The integration of Wu’s teachings into Orr is technically straightforward and would have been well-within the skill of an ordinary artisan. Orr describes a playback pipeline that includes a “source filter 214” which takes media received from the media manager and “extracts media data for presentation to the transform filter 216.” (EX1005, ¶[0024], Fig. 2.) The transform filter is a “pluggable filter” that uses decoders, like an “H264 decoder” and decompresses the media data to be rendered by the renderer 218. (*Id.*) These components align with Wu’s system “for audio and video decoding and playback” where it uses “separation unit 311” which splits the data into separate audio and video data, and “video decoder 302” and “audio decoder 303,” which decompress the individual audio and video data.

(EX1006, ¶[0076], Fig. 3.) (EX1002, ¶56.)

Orr teaches that its playback pipeline determines if “[c]ontent type of new channel [is] same as current channel,” and if so, then the system will “[i]nput content of new channel into current rendering pathway,” which as discussed above contains both the source filter (*i.e.* demultiplexer) and transform filter (*i.e.* decoders). (EX1005, Fig. 4, blocks 410 and 414.) In other words, Orr teaches checking both the container format and the compression format to decide whether to reuse those elements in the rendering pathway. (EX1002, ¶57.)

The proposed combination would involve modifying Orr’s playback pipeline to instead first check if the container format is the same to determine whether to reuse the source filter (*i.e.* demultiplexer), and if so, then check if the compression format is the same to determine whether to reuse the transform filter (*i.e.* decoders). When receiving new content that has the same container format as old content, but audio and video content that was compressed using different encoding formats, Wu teaches that if the “encoding parameters change (such as the audio format changing from mp3 to aac ...), it is clearly not feasible to decode and play using the original decoder.” (EX1006, ¶[0005].) Wu teaches that its system therefore “obtain[s] the audio and video decoder parameters,” and determines “changes in encoding formats” or whether they “have not changed.” (*Id.*, ¶¶[0016], [0028].) If the format has changed, “the current audio and video decoders” are “[d]isconnect[ed],” and new

audio/video decoders are “create[d] ... based on the aforementioned audio and video decoder parameters.” (*Id.*, ¶¶[0079]-[0080].) However, the demultiplexer remains the same. Thus, the combination would modify Orr in view of Wu to check the encoding format of the audio and video stream, determine if they have changed from the old content, and disconnect and create new audio and video decoders if the format has changed. (EX1002, ¶58.)

A POSITA would have been motivated to modify Orr to incorporate Wu’s teachings because of the known benefits of reusing a demultiplexer when the container format is the same. Orr itself recognizes the benefits of reusing elements of the playback pipeline, stating that re-initialization of the playback pipeline “may take more than a few seconds” because the media player will “tear[] down its existing rendering pathway and construct[] a new rendering pathway.” (EX1005, ¶[0004].) As a result, the “user will experience a delay in viewing the selected channel, which may result in an unsatisfactory user experience.” (*Id.*) Orr therefore teaches the well-known concept that reusing components will mitigate delay. In a situation where new content uses the same container format as old content, but different encoding formats, Wu recognizes that it is “not feasible to decode and play using the original decoder.” (EX1006, ¶[0005].) But a POSITA would recognize that the demultiplexer, which is responsible for separating the video and audio content from the container, could be reused if the container format is the same. (EX1002,

¶59.) By reusing the demultiplexer, this reduces the latency for rendering the new content.

A POSITA would also understand that creating a new demultiplexer requires re-establishing bitstream syncing and renegotiating the downstream media type (for example, RGB32 at a fixed resolution or PCM at a fixed sample rate). This requires more time and resources than simply replacing the decoders, which merely involves instantiating a new codec instance and feeding it the audio or video stream. Thus, reusing the demultiplexer saves time and increases processing efficiency. A POSITA would understand that whereas reinitializing the demultiplexer can cause noticeable discontinuities in the playback, maintaining the demultiplexer even while reinitializing newer codecs allows for seamless playback. (EX1002, ¶60.)

In addition, to the extent Patent Owner argues that Orr does not expressly disclose a demultiplexer, it also would have been obvious to incorporate the functionality of Wu's separator (*i.e.* a demultiplexer) into Orr's source filter. Demultiplexers for extracting digital audio and video signals have been known since the 1950s, and are well-attested in the prior art. (*See, e.g.*, EX1017, ¶52 ("The stream demultiplexer 634 separates the AV bit stream AVS into a video bit stream VBS and an audio bit stream ABS")). Orr discloses that its content signals contain both a video and an audio signal (EX1005, ¶[0018]) as well as the use of software decoders (such as H264), which decode audio or video signals, but not both (EX1005, ¶[0024]).

Thus, a POSITA would be motivated to demultiplex the audio and video signal in order that they can be decoded and rendered to the user. Without such demultiplexing, the decoders would be unable to decode the signal. A POSITA would have had a reasonable expectation of success because demultiplexers for digital signals have been in use for decades and well-within the skill of an ordinary artisan. (EX1002, ¶61.)

A POSITA would have had a reasonable expectation that the combination would succeed. (EX1002, ¶62.) Orr already describes performing a content type check to determine whether to reuse the rendering pathway—the proposed modification would simply check whether the container format is the same before determining whether to reuse Orr’s source filter (which performs the function of demultiplexing), then check whether the encoding formats are the same before determining whether to reuse Orr’s transform filter (which performs the function of the decoders). Since Orr already describes reusing the demultiplexer and the decoders, the proposed modification would simply require a slight modification in the programming logic. This would have been well-within the skill of an ordinary artisan.

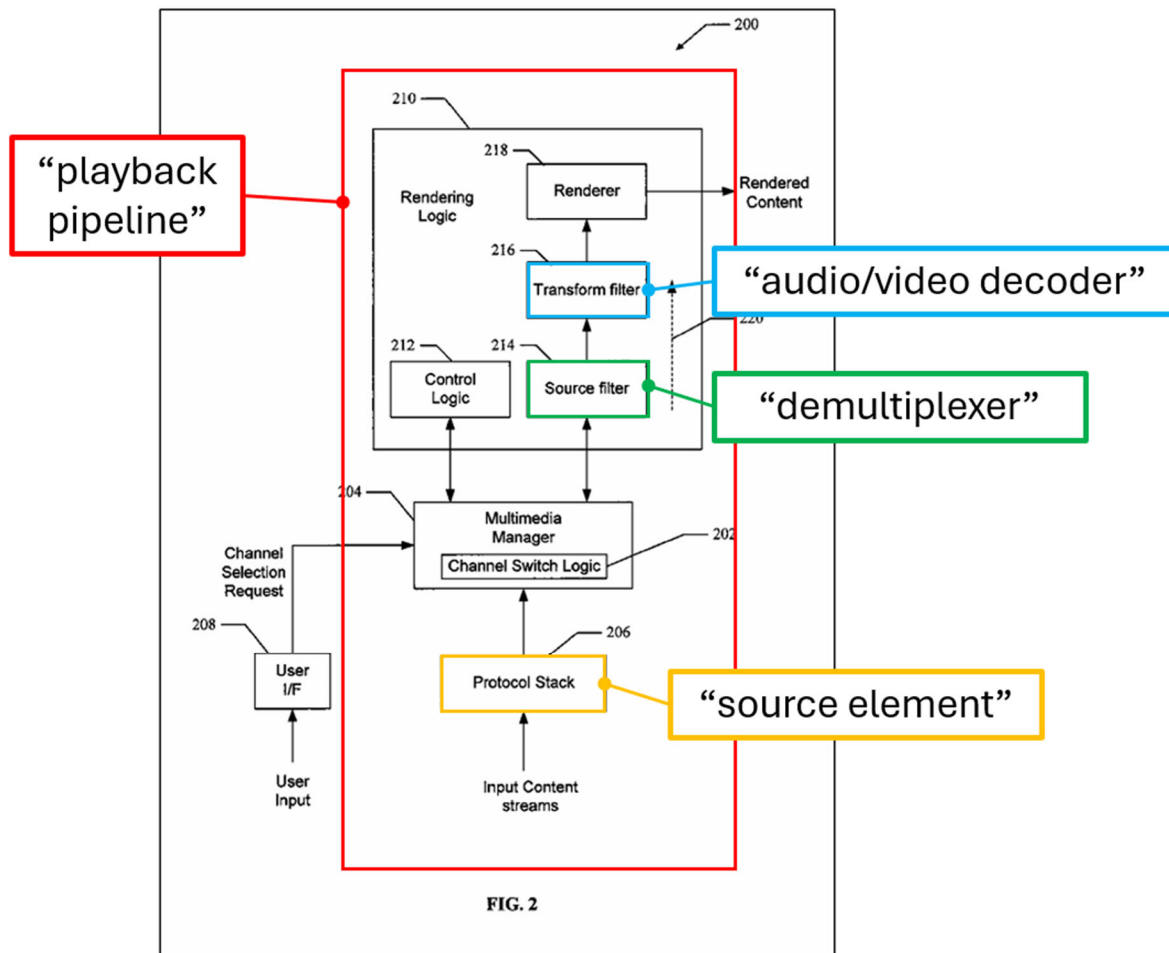
Orr and Wu are analogous art, because both are directed to rendering content. *See, e.g.*, EX1005, Abstract (“Methods and apparatus for fast channel switching between real time content on a device.”); EX1006, Abstract (“This invention

discloses a method supporting variable encoding based on DirectShow”). (EX1002, ¶63.)

1. Claim 1

- a) 1[pre]: A method comprising:**
- b) 1[a]: causing playback of a first piece of content using a playback pipeline;**

Orr discloses that the device includes several components that are arranged in a series and used to play back content, *i.e.* a ***playback pipeline***. For example, in Figure 2, Orr discloses that content is played back using a “protocol stack 206,” a “multimedia manager 204,” and “rendering logic 210.” (EX1005, ¶¶[0023-28]; FIG. 2.) These components arranged in a series constitute a ***playback pipeline*** and comprise, for example, a source element, demultiplexer, audio decoder, and video decoder. (EX1002, ¶¶63-67.)



(EX1005, Fig. 2) (Annotated)

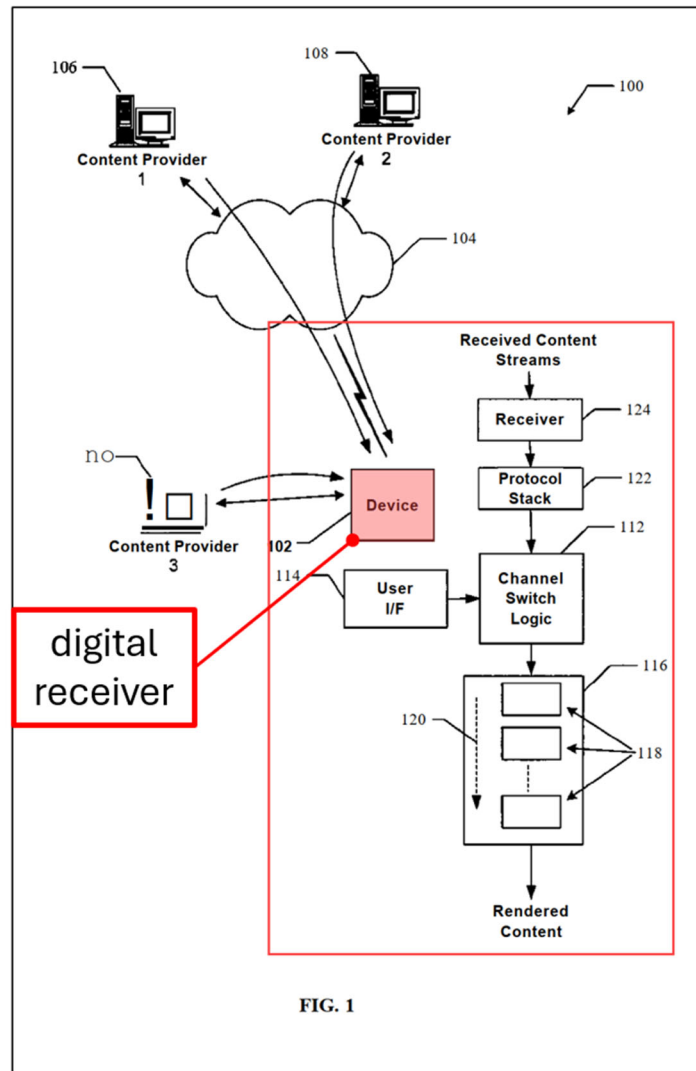
Orr describes that this playback pipeline is used for *playback of a first piece of content*. (EX1002, ¶68.) When a user selects a channel, the “protocol stack 206” of the playback pipeline “receives content streams ... from a network and/or from another communication channel.” (EX1005, ¶¶[0023], [0036] (“[T]he processing logic 302 controls the protocol stack interface 310 to obtain the content for the selected channel.”)) The multimedia manager then “provides control signals to

control logic 212 of rendering logic 210,” which “construct[s] a rendering pathway 220 that includes functional elements that operate to process content data associated with a selected content channel.” (EX1005, ¶[0024].) Within the rendering pathway, “[t]he source filter 214 understands the format of the received transport stream and extracts media data for presentation to the transform filter 216,” which “is a pluggable filter, such as an H264 decoder that takes in data from the source filter 214 and decompresses it.” (*Id.*) The decoded media “is then input to a renderer 218 that operates to render the content on the device,” *i.e. playback of a first piece of content.* (*Id.*)

- c) **1[b]: receiving, by a digital receiver, a selection of a new piece of content for playback during playback of the first piece of content;**

“digital receiver”: Content is delivered to Orr’s device via a data network 104 as a **digital** signal. (EX1005, ¶[0016]; ¶[0017] (“The content providers 106, 108 and 110 operate to provide content to the device 102.”); FIG. 1.) Orr describes that the “circuits described in [Orr] may be implemented or performed with ... a digital signal processor (DSP). (*Id.*, ¶[0053].)

Orr describes its device 102 as including a receiver 124, channel switch logic 112, a user interface 114, a protocol stack 122, and rendering logic 116. (*Id.*, ¶[0018]; FIG. 1.) Orr’s “device 120” is an express disclosure of claim 1’s “**digital receiver.**” (EX1002, ¶¶70-71.)

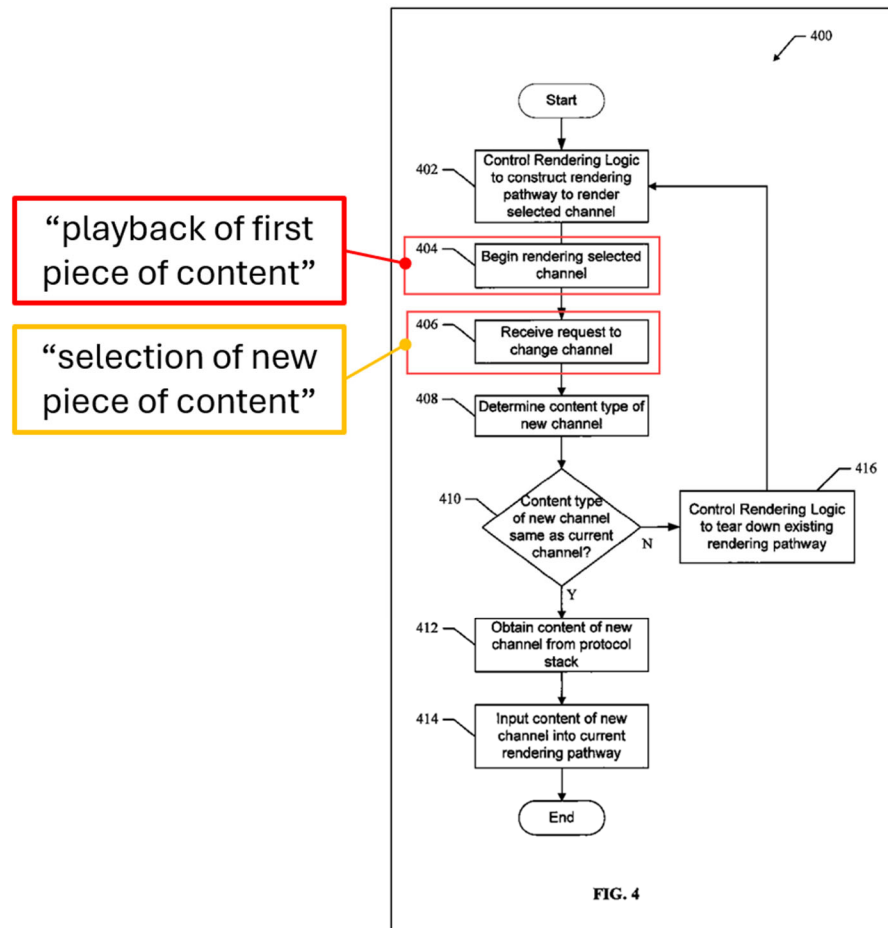


“receiving a selection of new piece of content”: Orr describes that “channel switch logic 112 [which is part of device 102] operates to select a particular content channel and inputs data from this content data to the rendering logic 116.” (EX1005, ¶[0019].) “The channel switch logic 112 then operates to control rendering logic 116 to construct the functional elements 118 needed to decode and render the content [including] content filters [118], transform filters [118], decoders [118], and/or any

other type of functional element needed to process the content flow of the selected channel for rendering at the device 102.” (EX1005, ¶[0019].) (EX1002, ¶72.)

“during playback of the first piece of content”: Orr discloses that “[d]uring operation, the device user may request a channel *change* to render content from a *different* channel.” (EX1005, ¶[0020]; ¶[0042] (“[A] channel request is received to render content on a *different* channel.”).) The user interface receives a channel selection indicator from the user and passes it to the channel switch logic during active playback. (EX1005, ¶[0042]; claim 1.) (EX1002, ¶74.)

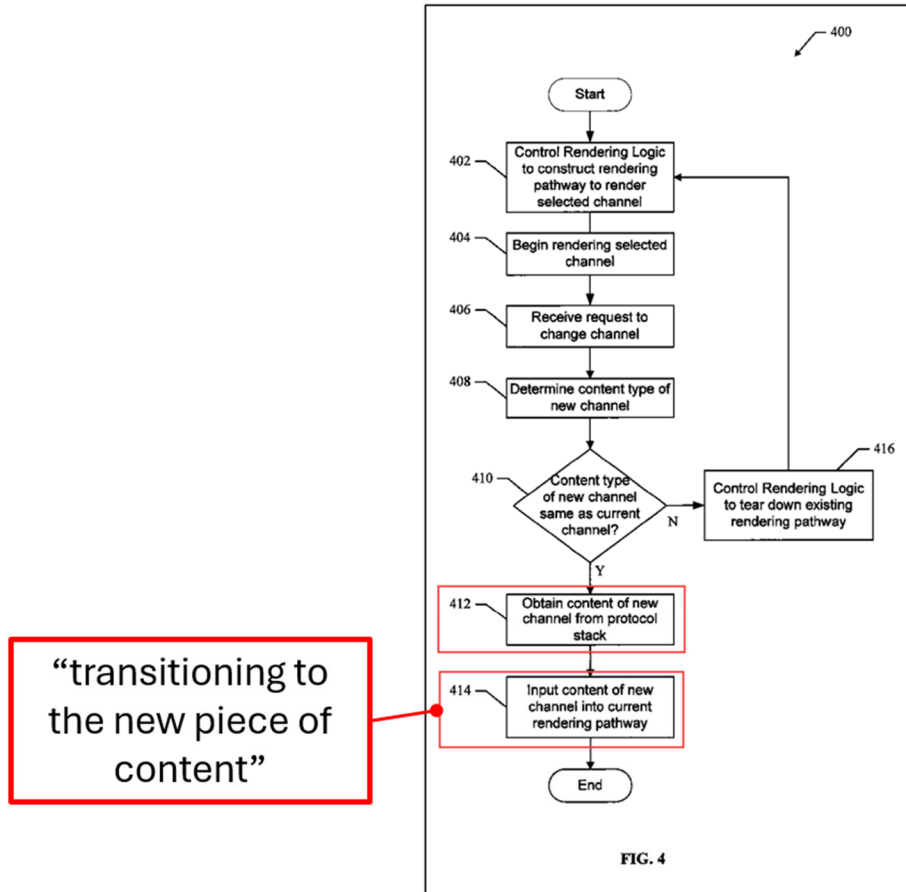
In the following annotated figure, FIG. 4 depicts that block 406 (“Receive request to change channel”) appears in the flow after block 404 (“Begin rendering selected channel”) confirming that the selection for a new piece of content is received while the first channel is actively being rendered. (EX1005, FIG. 4, blocks 404, 406.) (EX1002, ¶75.)



- d) 1[c]: in response to the receiving of the selection, transitioning from playback of the first piece of content to the new piece of content;

Orr describes *transitioning to the new channel's content in response to the channel selection*. In response to receiving a new channel selection indicator (the same selection in element 1[b]), the channel switch logic performs a series of operations that eventually results in obtaining the new content from the protocol stack and either inputs it into the existing rendering pathway or constructs a new pathway, both of which are the device *transitioning to the new content*. (EX1005,

¶[0022]; ¶[0046-47]; FIG. 4, blocks 412–414.) The *transition to the new content* requires the selection of new content, and is therefore *in response to* the selection. Orr’s Claim 1 also recites “inputting the second content into the first rendering pathway if it is determined that the second content can be rendered using the first rendering pathway.” As shown in the annotated figure below, FIG. 4, after in response to receiving a “request to change channel” at step 406, Orr’s device performs several steps (described *infra* at limitations 1[d], 1[e], and 1[f]) that results in “[o]btaining content of new channel” at step 412 and playing the content at step 414. (EX1005, ¶[0046-47].) (EX1002, ¶¶76-77.)

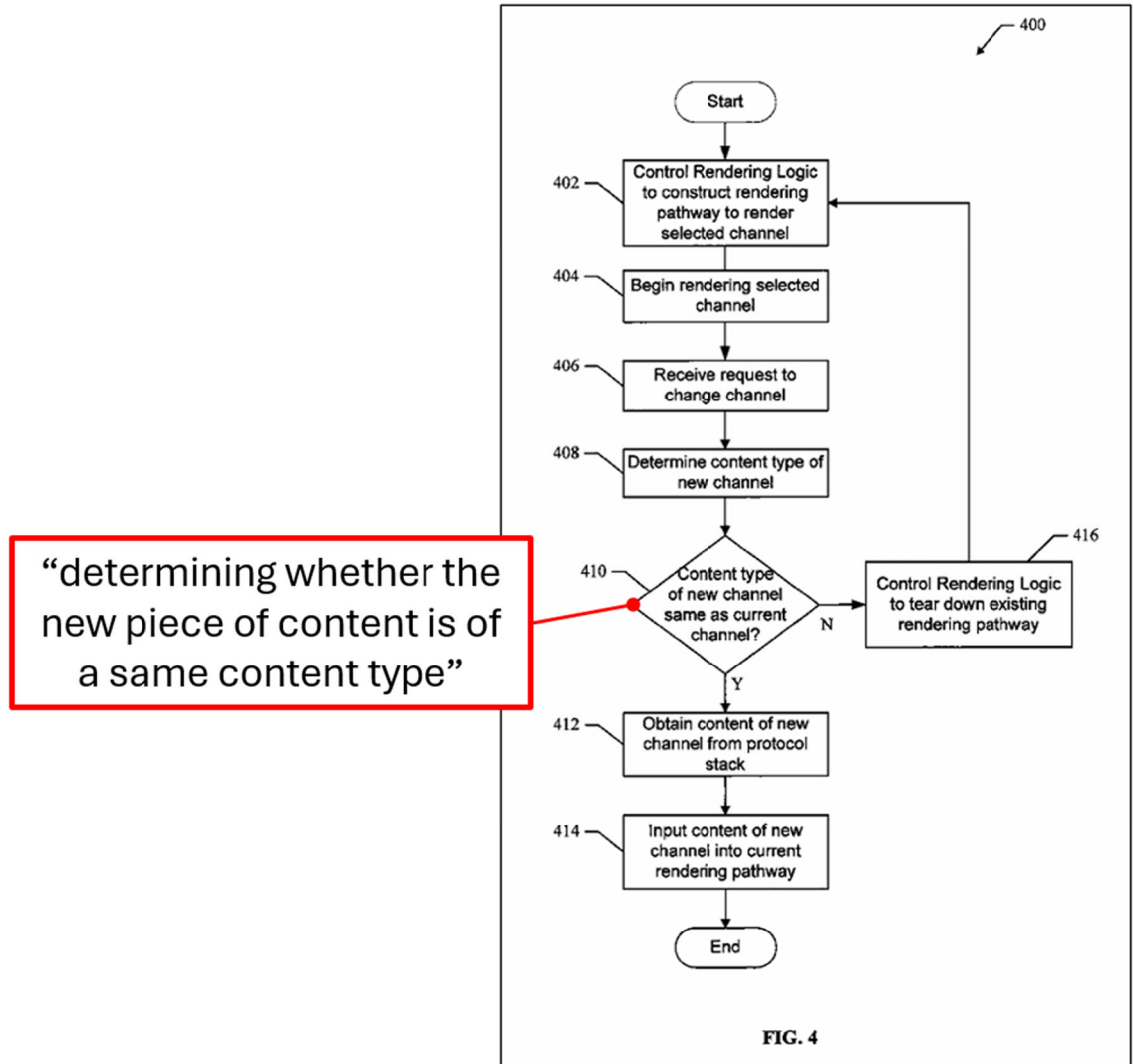


(EX1005, FIG. 4.) (Annotated)

- e) **1[d]: determining whether the new piece of content is of a same content type as the first piece of content; and**

Orr’s device *determines whether the new piece of content is the same content type as the first piece of content*. Specifically, Orr’s channel switch logic “determines the type of content on the new channel to be rendered” and then assesses “if a current rendering pathway can be used to render the content on the new channel.” (EX1005, ¶[0021]; *see also* ¶[0043]–¶[0044].) Orr describes that the

determination of the content-type involves “determin[ing] if the content on the new channel *has the same format* as the content currently being rendered.” (EX1005, ¶[0026].) FIG. 4 depicts this determination explicitly at block 410: “Content type of new channel *same as current channel?*” (EX1005, FIG. 4, block 410.) As discussed above, in the proposed modification in view of Wu, this determination first checks if the new content has the same container format as the old content, *i.e. determine if they have the same content format*. (EX1002, ¶¶78-80.) As discussed in the next limitation, the proposed combination then checks if the new content has the same encoding formats as the old content.



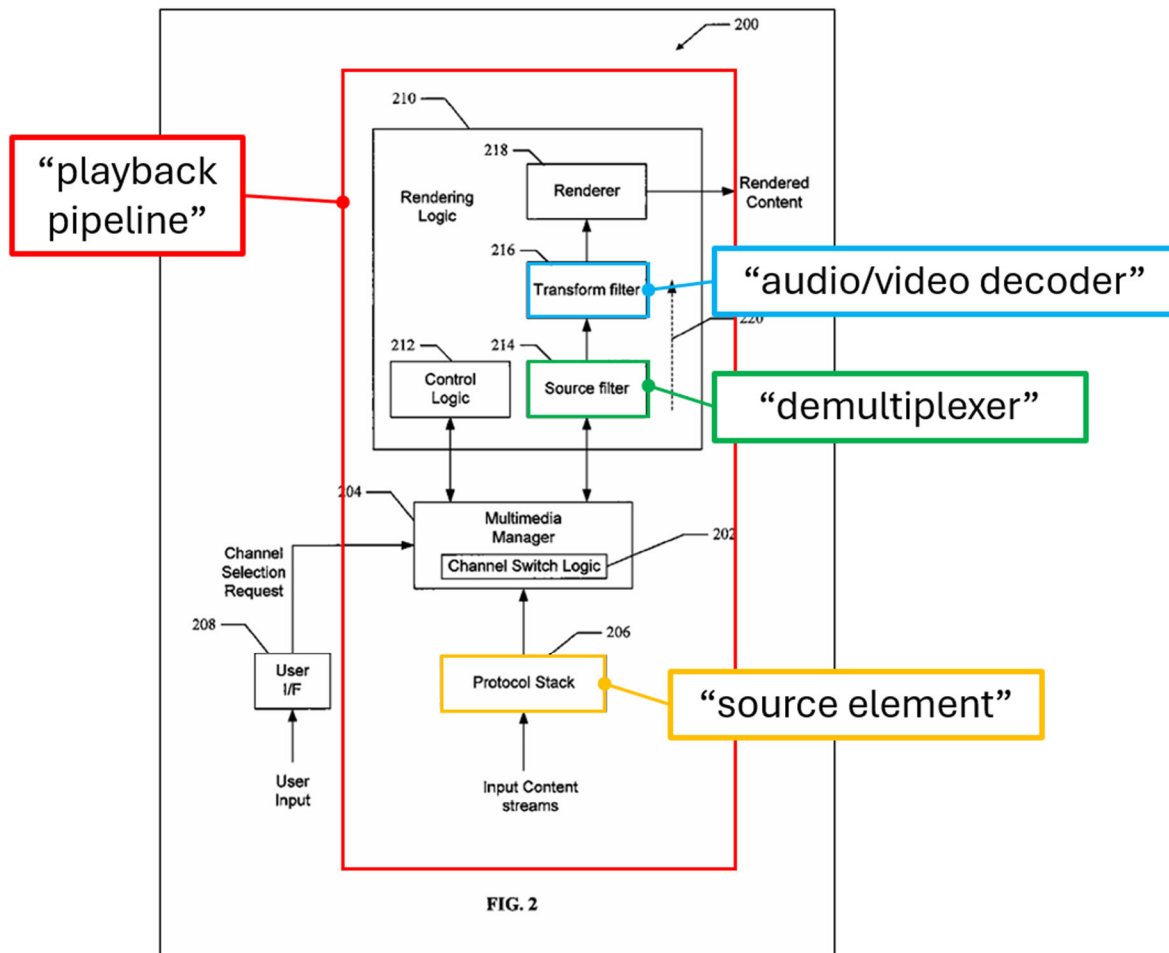
(EX1005, FIG. 4.) (Annotated)

- f) 1[e]: based on the new piece of content being the same content type as the first piece of content, determining whether an audio decoder or a video decoder of the playback pipeline used to play back the first piece of content should be preserved; and
- g) 1[f]: based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer of

the playback pipeline while replacing the audio decoder and the video decoder.

Orr in view of Wu renders obvious these limitations. These limitations require (1) a “playback pipeline” for the first piece of content that has a source element, a demultiplexer, an audio decoder, and a video decoder, and (2) based on determining the new content has the same content type, determining whether to preserve an audio decoder or a video decoder of the “playback pipeline,” and (3) based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer while replacing the audio and video decoder. (EX1002, ¶¶81-82.)

“playback pipeline”: Orr discloses that the device includes several components that are arranged in a series and used to play back content, *i.e.* a **playback pipeline**. For example, in Figure 2, Orr discloses that content is played back using a “protocol stack 206,” a “multimedia manager 204,” and “rendering logic 210.” (EX1005, ¶¶[0023-28]; FIG. 2.) These components arranged in a series constitute a **playback pipeline** and comprise the claimed source element, demultiplexer, audio decoder, and video decoder. (EX1002, ¶83.)



(EX1005, Fig. 2) (Annotated)

"Source element." Orr discloses that its playback pipeline includes a *source element*. (EX1002, ¶84.) Specifically, Orr discloses a "protocol stack 206" and a "protocol stack interface 310," which together constitute a *source element*. (EX1005, ¶¶[0025], [0033].) The protocol stack "receive[s] channel content from a variety of sources and operates to make this channel content available to the protocol stack interface 310," including "content data associated with a selected content

channel,” *i.e.* the ***new piece of content***. (EX1005, ¶[0033]; *see also* EX1005, ¶[0023] (“[T]he protocol stack 206 may receive the content streams from a network and/or from another communication channel.”)).

“A demultiplexer.” Orr discloses that its playback pipeline includes a ***demultiplexer***, *i.e.* a component that separates the video and audio signal. (EX1001, 5:36-38 (“The demux demultiplexes the content data ... to obtain an audio signal and a video signal.”).) Orr teaches that content “compris[es] video, audio, captioning, and overlays etc.” (EX1005, ¶[0018].) Orr further discloses ***demultiplexing*** of the audio and video performed by a “source filter 214” that is part of the “rendering logic 210,” *i.e.* a portion of the ***playback pipeline***. (EX1005, ¶[0024].) Specifically, the source filter “extracts media data for presentation to the transform filter 216.” (EX1005, ¶[0024].) Orr explains that the “transform filter 216 is a pluggable filter” and uses, for example, “an H264 decoder that takes in data from the source filter 214 and decompresses it.” (EX1005, ¶[0024].) H264 is a known ***video*** compression/decompression algorithm—and decoders generally decompress audio or video but not both. (EX1016 [US8289370], 2:44-47 (“H.264 standard video codecs”).) Thus, because the decoders can only operate on either a video or audio signal, but not both, a POSITA would understand that the “source filter” “extracts media data” in the form of a separate video and audio signal for decoding by the transform filter using respective video and audio decoders. (EX1002, ¶85.) This is

further confirmed by Orr’s disclosure that the transform filter sends the decompressed signal to “the renderer 218 [which] operates to render *video, audio*, and overlays that are included in the content data.” (EX1005, ¶[0024].)

In addition, as discussed above, it would have been obvious to incorporate Wu’s separator function into Orr’s source filter. Wu’s separator separates the audio and video signal, *i.e. a demultiplexer*. (EX1002, ¶86.)

“An audio decoder.” Orr discloses that its playback pipeline includes *an audio decoder*. Orr teaches that content data includes “audio” and that the “rendering logic” includes elements “needed to decode and render the content.” (EX1005, ¶¶[0018]-[0019].) In addition, Orr discloses that the rendering logic includes a “transform filter 216,” which “is a pluggable filter” that “decompresses” (*i.e. decodes*) media content using a “decoder.” (EX1005, ¶[0024].) The pluggable filter sends the decoded signals to the “renderer” which “render[s] ... audio.” (EX1005, ¶[0024].) A POSITA would therefore understand that the transform filter of the rendering logic includes an *audio decoder*. (EX1002, ¶87.)

In addition, it would have been obvious to include an audio decoder in Orr’s playback pipeline. Audio decoders to decode digital audio signals for media playback were well-known, including those often used with the H.264 video decoder like AAC. (EX1017, ¶[0055] (“The stream demultiplexer ... outputs ... the audio bit stream ABS to the audio decoder 612”); ¶¶[0091]-[0092] (“H.264 ... may be

similarly used ... MPEG-2 AAC is used as an encoding scheme for speech signals.”)). Orr already teaches that the rendering logic includes a decoder for rendering content, which includes audio. (EX1005, ¶¶[0018]-[0019].) A POSITA would be motivated to give effect to Orr’s teachings by using an audio decoder, because that would allow the audio to be decoded and rendered for playback to the user. (EX1002, ¶88.) A POSITA would have had a reasonable expectation of success because audio decoders were ubiquitous at the time of the ’139 patent and it was well-within the skill of an ordinary artisan to implement an audio decoder in a playback pipeline.

“A video decoder.” Orr discloses that its playback pipeline includes a ***video decoder***. (EX1002, ¶89.) Specifically, Orr discloses that its “transform filter 216 is a pluggable filter, such as an H264 decoder,” a well-known video decoder. (EX1005, ¶[0024].)

“based on the new piece of content being the same content type as the first piece of content, determining whether an audio decoder or a video decoder of the playback pipeline used to play back the first piece of content should be preserved”:

Orr teaches that if the “[c]ontent type of new channel [is] same as current channel,” then the system will “[i]nput content of new channel into current rendering pathway.” (EX1005, Fig. 4, blocks 410 and 414.) Because the rendering pathway includes both the source filter (*i.e.* ***demultiplexer***) and transform filter (*i.e.*

decoders), a POSITA would understand that this determination checks both the container format and the encoder format. (EX1002, ¶90.) As discussed above, in the proposed combination, this teaching in Orr is modified to first check the container format, and if they have the same container format (*i.e. new content being the same content type as the first content*), then checking if the encoder formats are the same (*i.e. determine whether the audio/video decoders should be preserved*).

Wu teaches how to check the encoder formats: by “[o]btaining the audio and video decoder parameters” and determining whether there were “changes in encoding formats,” or whether they “have not changed.” (EX1006, ¶¶[0016], [0028].) If the encoding formats have changed, it is “not feasible to decode and play using the original decoder.” (EX1006, ¶[0005].) Thus, checking whether the encoder formats are the same is a *determination whether an audio decoder or a video decoder of the playback pipeline should be preserved*. (EX1002, ¶91.)

“based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer of the playback pipeline while replacing the audio decoder and the video decoder”:

Wu teaches that if it determines that the encoder formats are not the same (*i.e. determine that audio and video decoders should not be preserved*), then “the current audio and video decoders” are “[d]isconnect[ed],” and new audio/video decoders are “create[d] ... based on the aforementioned audio and video decoder parameters,” *i.e.*

replace the audio and video decoders. (*Id.*, ¶¶[0079]-[0080].) In Wu’s system, the separator (*i.e.* *demultiplexer*) is preserved. (*Id.*, Fig. 3.) And in Orr’s system, the protocol stack (*i.e.* *source element*) is preserved when the decoders are replaced. (EX1005, ¶[0044].) As discussed above, it would have been obvious to incorporate these teachings into Orr. (EX1002, ¶92.)

2. Claim 2

- a) **2[pre]: The method of claim 1, wherein the determining whether to preserve comprises:**
- b) **2[a]: detecting codec information for the new piece of content, the codec information indicating types of decoders needed for the new piece of content; and**

Orr teaches detecting the format and codec requirements of the new content as part of the content-type determination. Orr describes its channel switch logic as determining “whether or not the functional elements 118 that currently exist on the rendering pathway 120 are able to process and render the content on the new channel,” (EX1005, ¶[0021]), which a POSITA would understand requires ascertaining the codec or format type of the new channel’s content. (EX1002, ¶¶93-94.) Orr further states: “there is no guarantee that the source 214 and transform 216 filters to decode the new channel will be the same as those being used for the current channel” (EX1005, ¶[0026]), confirming that the system evaluates what decoder types the new content requires. Orr’s claim 4 expressly recites “determining that the second content can be rendered using the first rendering pathway if the second

content is of the same format as the first content.” This necessarily involves detecting the format (codec) information of the new content. (EX1005, claim 4.) (EX1002, ¶95.)

Wu also teaches that its system “obtain[s] the audio and video decoder parameters,” and determines “changes in encoding formats” or whether they “have not changed.” (EX1006, ¶¶[0016], [0028].) (EX1002, ¶96.)

c) 2[b]: determining whether the audio decoder and the video decoder in the playback pipeline are of the types indicated by the codec information.

Orr teaches this determination as part of its format compatibility check. Having determined the format and codec requirements of the new content, Orr's processing logic determines whether the existing functional elements (including the transform filter and decoders) are capable of processing content in that format: “channel switch logic 112 determines whether or not the functional elements 118 that currently exist on the rendering pathway 120 are able to process and render the content on the new channel.” (EX1005, ¶[0021].) This is a direct comparison between the existing decoder types and the codec requirements of the new content (including audio and video decoders). (*See, e.g.*, EX1005, ¶[0024] (Orr describes the transform filter as “a pluggable filter, such as an H264 decoder that takes in data from the source filter 214 and decompresses it.”) (EX1002, ¶¶97-98.)

Wu also teaches that its system “obtain[s] the audio and video decoder

parameters,” and determines “changes in encoding formats” or whether they “have not changed.” (EX1006, ¶¶[0016], [0028].) (EX1002, ¶99.)

3. Claim 3

- a) **The method of claim 2, further comprising preserving an entirety of the playback pipeline in response to the audio decoder and the video decoder in the playback pipeline being the types indicated by the codec information.**

Orr teaches preserving the entirety of the existing rendering pathway when the content types match (*i.e.* both container format and encoder format match). As discussed above in limitation 1[e], Orr discloses a ***playback pipeline*** in the form of a protocol stack (*i.e.* ***source element***), a multimedia manager, and rendering logic (*i.e.* ***demultiplexer and audio/video decoders***). Orr teaches that when the processing logic determines that the existing pathway can render the new content, “the channel switch logic 112 does not re-initialize the rendering logic 116 but instead, feeds the content from the new channel into the current rendering pathway.” (EX1005, ¶[0022].) FIG. 4 reflects this at block 414: the entire rendering pathway (source filter, transform filter, and renderer) is preserved and used for the new content without teardown. (EX1005, ¶[0028]; ¶[0047]; FIG. 4, blocks 412–414.) Orr's claim 1 recites “inputting the second content into the first rendering pathway if it is determined that the second content can be rendered using the first rendering pathway,” confirming that the entire existing pathway is preserved and reused.

(EX1005, claim 1.) (EX1002, ¶¶100-01.)

4. Claim 4

- a) **4[pre]: The method of claim 2, further comprising:**
- b) **4[a]: in response to the audio decoder and the video decoder in the playback pipeline being different than the types indicated by the codec information,**
- c) **4[b]: obtaining a new audio decoder and a new video decoder based on the codec information.**

See limitations 1[e] and 1[f]. (EX1002, ¶102.)

5. Claim 8

- a) **8[pre]: The method of claim 1, further comprising:**
- b) **8[a]: based on the first piece of content and the new piece of content not being the same content type, deconstructing the playback pipeline by releasing the source element, the demultiplexer, the audio decoder, and the video decoder; and**

Orr teaches full deconstruction of the rendering logic when content types differ. At FIG. 4, block 416, when the content types are determined not to match (block 410, “N[o]”), the processing logic controls the rendering logic interface to signal the rendering logic to tear down the existing rendering pathway. (EX1005, ¶¶0044]–¶¶0045]; FIG. 4, block 416.) Orr’s claim 6 recites “tearing down the first rendering pathway” in this scenario. (EX1005, claim 6.) The teardown releases all elements of the rendering pathway including the source filter (corresponding to the source element), the transform filter (corresponding to the demultiplexer, audio

decoder, and video decoder), and the renderer. (EX1005, ¶[0045]; FIG. 2.) (EX1002, ¶¶103-04.)

Orr also discloses that the protocol stack is released. As discussed above for limitation 1[d], Orr describes that the protocol stack can use RTP to manage downloading of the content. However, Orr also describes that content can be provided “as files in (MP4) format,” and a POSITA would further understand that other formats and streaming protocols (*e.g.* container formats) could be used. (EX1005, ¶[0017].) Orr also states that the “multimedia manager 204 communicates with the protocol stack 206 to receive real time content and *also* functions to multiplex in *streams from other transports if needed.*” (*Id.*, ¶[0025].) Thus, Orr teaches that “if needed,” *i.e.* if the streaming format or protocol is different from that supported by the current protocol stack, a new “transport” or protocol stack would be created as it would be needed. Accordingly, in the event the new piece of content is not the same content type because it also uses a completely different protocol or format, it would be necessary to release and create a new protocol stack that is compatible with the new protocol or format. (EX1002, ¶105.)

A POSITA would also have considered it obvious to release and create a new protocol stack in the event the new content uses a different protocol. A POSITA would understand that different container formats are supported using different APIs. Therefore, if a different protocol or format is encountered, the multimedia

manager would need to use a different API than the current protocol stack. In creating a new protocol stack by calling the new API, a POSITA would be motivated to release the current protocol stack, as it would save computing resources. A POSITA would also have a reasonable expectation of success, as releasing the protocol stack is easily implemented using standard programming techniques and well-within the skill of an ordinary artisan. (EX1002, ¶106.)

c) 8[b]: creating a new playback pipeline by acquiring other resources that are applicable to a content type of the new piece of content.

Following the pathway teardown at block 416, FIG. 4 of Orr returns to block 402 (“Control Rendering Logic to construct rendering pathway to render selected channel”), at which point new resources appropriate to the new content’s format are acquired to construct a fresh rendering pathway. (EX1005, ¶[0040]; ¶[0045-46]; FIG. 4.) In addition, Orr’s claim 6 recites “constructing a second rendering pathway to render the second content” which inherently includes acquiring new elements applicable to the new content type. (EX1005, claim 6.) (EX1002, ¶¶107-08.)

In addition, as discussed above for limitation 8[a], a POSITA would be motivated to release and create a new protocol stack. (EX1002, ¶109.)

6. Claim 9

a) The method of claim 1, wherein content types include

HTTP Live Streaming (HLS) format and Dynamic Adaptive Streaming over HTTP (DASH) format.

Orr teaches a channel switching system for devices that receive and render content streams over data networks, including wireless networks and the Internet. (EX1005, ¶¶[0014-17].) Orr’s protocol stack 310 is designed to receive content “from a variety of sources” delivered over any such network. (EX1005, ¶[0033].) Orr further teaches that its system handles content delivered in various formats, including RTP and MP4 formats explicitly referenced as examples. (EX1005, ¶[0017].) The ’139 patent identifies HLS and DASH as examples of adaptive streaming content types. (EX1001, 5:1-10.) HLS and DASH are adaptive HTTP streaming protocols that were well known to POSITAs by the priority date as the predominant formats for streaming content delivery over IP networks, and were a natural and obvious choice of content types to implement in a channel-switching system like Orr’s that targets IP-networked consumer devices. (*See, e.g.*, EX1009, 15.) A POSITA would have been motivated to implement Orr’s channel-switching system using HLS and DASH formats with a reasonable expectation of success, as both formats involve the same type of format-compatibility determination and pipeline reuse that Orr teaches. (EX1002, ¶¶110-11.)

7. Claim 10

- a) The method of claim 1, wherein the receiving the selection, transitioning to the new piece of content,**

determining whether the new piece of content is of the same content type as the first piece of content, and determining whether to preserve is performed by a player.

The '139 patent describes the “player” as the component that constructs the pipeline, manages it, and decides whether to preserve or reconstruct it upon a content switch. (EX1001, 4:66-5:10.) (EX1002, ¶¶112-13.)

Orr’s channel switch logic 202 (which may be embedded within the multimedia manager 204) (EX1005, ¶[0023]; FIG. 2.) performs each of the operations recited in this limitation. It receives channel selection requests from the user interface (EX1005, ¶[0025]; ¶[0042]); determines the content type of the new channel and whether it is compatible with the existing rendering pathway (EX1005, ¶[0021]; ¶[0026]; ¶[0043-44]); and controls the rendering logic to either preserve the existing pathway or construct a new one. (EX1005, ¶[0022]; ¶[0037]; ¶[0044-47].) Orr's claim 7 further confirms this mapping, reciting an “apparatus for channel switching” comprising “processing logic configured to determine whether the real time content can be rendered using an existing rendering pathway” and “a rendering interface configured to input the real time content into the existing rendering pathway.” (EX1005, claim 7.) (EX1002, ¶114.)

8. Claim 11

- a) 11[pre]: The method of claim 1, further comprising:**
- b) 11[a]: based on the first piece of content and the new**

piece of content not being the same content type, releasing the source element; and

- c) 11[b]: creating a new playback pipeline by acquiring a different source element that is applicable to a content type of the new piece of content.**

See Claim 8. (EX1002, ¶115.)

9. Claim 12

- a) 12[pre]: The method of claim 1, further comprising:**
- b) 12[a]: based on the first piece of content and the new piece of content not being the same content type, releasing the demultiplexer; and**

As discussed above in limitation 8[a], when content types are determined not to match at block 410 (“N[o]”), Orr’s channel switch logic signals the rendering logic to tear down the existing rendering pathway in its entirety at block 416. (EX1005, ¶¶[0044-45]; FIG. 4, block 416.) Orr’s claim 6 recites “tearing down the first rendering pathway” in this scenario. (EX1005, claim 6.) The teardown releases all elements of the rendering pathway including the demultiplexer. (EX1005, ¶[0045]; FIG. 2.) (EX1002, ¶116.)

- c) 12[b]: creating a new playback pipeline by acquiring a different demultiplexer that is applicable to a content type of the new piece of content.**

Following the teardown at block 416, Orr returns to block 402 to construct a new rendering pathway with functional elements suited to the new content type. (EX1005, ¶¶[0040], [0045-46]; FIG. 4.) Orr’s source filter 214 performs

demultiplexing functions by extracting media data from the received transport stream for presentation to the transform filter. (EX1005, ¶[0024].) Orr teaches that the source filter “understands the format of the received transport stream” in performing this function. (EX1005, ¶[0024].) Because the source filter’s demultiplexing function is tied to understanding the format of the content stream, a POSITA would understand that when a new rendering pathway is constructed for content of a different type, the new source filter acquired for that pathway must be one capable of understanding and extracting media data from the new content type, *i.e.*, a different element performing demultiplexing that is applicable to the new content type. This is further confirmed by Orr’s teaching that different content types require different rendering pathways constructed with appropriate functional elements: “the channel switch logic 202 notifies the control logic 212 to construct a new rendering pathway 220 to decode the content on the new channel.” (EX1005, ¶[0027].) Orr’s claim 6 recites “constructing a second rendering pathway to render the second content” following teardown, which inherently includes acquiring a new source filter suited to the new content’s format. (EX1005, claim 6.) (EX1002, ¶¶117-18.)

10. Claim 14

- a) 14[pre]: A system comprising:**
- b) 14[a]: one or more hardware processors; and**

Orr discloses a device comprising hardware processors. Orr's processing logic is described as comprising “at least one of a CPU, processor, gate array, hardware logic, memory elements, virtual machine, and/or hardware executing software.” (EX1005, ¶[0031].) Orr’s apparatus claims similarly recite “processing logic configured to determine whether the real time content can be rendered using an existing rendering pathway.” (EX1005, claim 7.) (EX1002, ¶¶119-20.)

c) 14[b]: a storage device storing instructions that configure the one or more hardware processors to perform operations comprising:

Orr discloses a storage device storing executable instructions. Orr teaches that its channel switching system “comprises a computer program product having one or more program instructions (‘instructions’) or sets of ‘codes’ stored or embodied on a machine-readable medium,” and that “[w]hen the codes are executed by at least one processor ... their execution causes the processor to provide the functions of the channel switching system described herein.” (EX1005, ¶[0038].) Orr further describes specific example storage media including “a floppy disk, CDROM, memory card, FLASH memory device, RAM, ROM, or any other type of memory device or machine-readable medium.” (*Id.*) Orr’s claim 19 recites a corresponding computer program product comprising “a machine-readable medium encoded with codes executable to: receive a channel selection indicator ...; determine whether the real time content can be rendered using an existing rendering pathway; and input the

real time content into the existing rendering pathway.” (EX1005, claim 19.)
(EX1002, ¶121.)

d) 14[c]-14[h]

See limitations 1[a]-1[f]. (EX1002, ¶¶122-27.)

11. Claim 15

See claim 2. (EX1002, ¶128.)

12. Claim 16

See claim 3. (EX1002, ¶129.)

13. Claim 17

See claim 4. (EX1002, ¶130.)

14. Claim 18

See claim 8. (EX1002, ¶131.)

15. Claim 19

a) 19[pre]

See limitations 14[a] and 14[b]. (EX1002, ¶132.)

b) 19[a]-19[f]

See limitations 1[a]-1[f]. (EX1002, ¶¶133-138.)

VIII. Ground 2: Orr/Wu/Hsu

A. Motivation to Combine

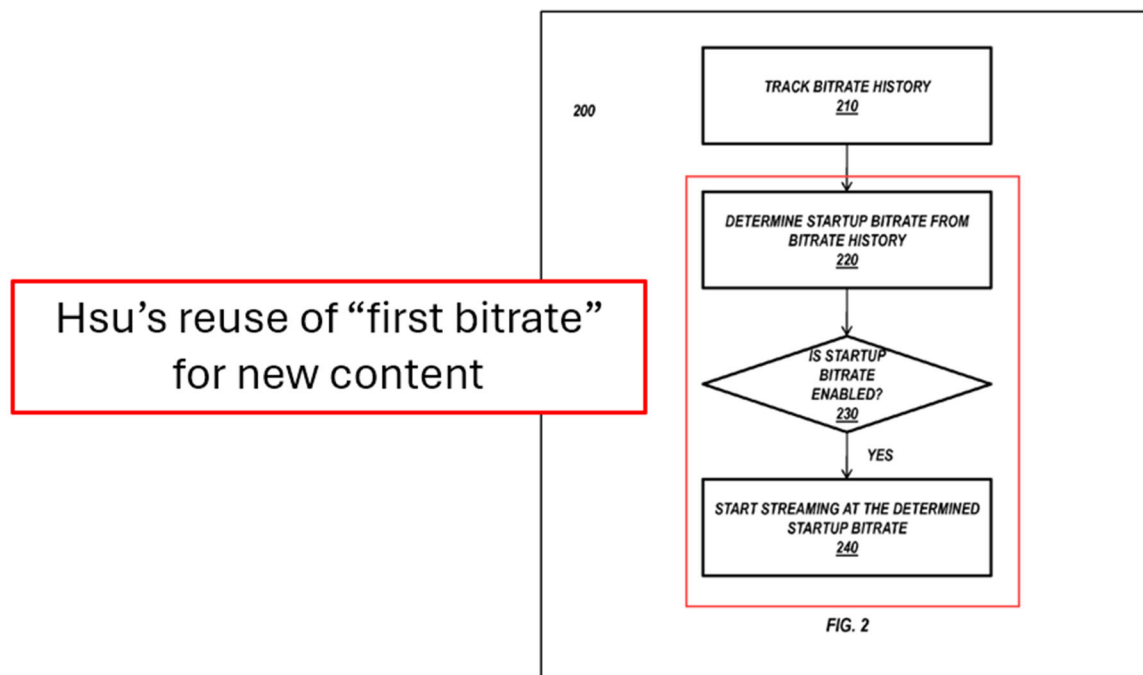
A POSITA would have been motivated to modify Orr (in view of Wu) to incorporate Hsu’s teachings of bitrate initialization and would have had a reasonable

expectation of success in doing so. (EX1002, ¶¶139-44.) A POSITA would have been motivated by the known benefits of reducing latency and degradation of the stream quality by reusing the current bitrate for new content, as changing to the default bitrate for new content requires time and resources, and also causes the new content to initially stream at a lower quality. (*Id.*)

The integration of Hsu's bitrate initialization component into Orr is technically straightforward. Orr's protocol stack manages the retrieval of content data from the content source. (EX1005, ¶[0024].) Hsu teaches initializing a new content download session at a historically derived bitrate rather than the system default. (EX1007, 6:1-5.) A POSITA would implement Hsu's bitrate initialization teaching in the source filter of Orr's pipeline, configuring it to begin downloading new content at the previously achieved bitrate. This requires no architectural modification to Orr and produces the entirely predictable result of faster, higher-quality content transitions. Neither reference teaches away from the combination. In addition, the modification of Orr in view of Wu affects downstream processing and would also not teach away the inclusion of Hsu. (EX1002, ¶140.)

A POSITA would have been motivated to address the same fundamental user experience problem that Orr and Hsu identify: the latency and quality degradation experienced by a user when switching between pieces of content on a streaming device. Orr addresses the pipeline reconstruction component of this problem:

eliminating the delay caused by tearing down and rebuilding the rendering pathway. (EX1005, ¶¶[0004]–¶[0005]; ¶[0015].) As shown in FIG. 2 below, Hsu addresses the bitrate initialization component: eliminating the quality degradation and latency caused by changing to the default lowest bitrate at the start of each new session. (EX1007, 5:33-39; FIG. 2.) (EX1002, ¶141.) Thus, a POSITA would be motivated to use Hsu in order to improve the user experience by increase streaming quality and reduce latency, consistent with Orr’s goals.



(EX1007, FIG. 2.)

These are complementary solutions: Orr operates at the pipeline and rendering layer, while Hsu operates at the download and bitrate layer. A POSITA designing a comprehensive content-switch optimization system would naturally consider and

implement both. (EX1002, ¶152.)

A POSITA combining Orr and Hsu would implement Hsu's bitrate initialization teaching in the source element of Orr's pipeline. For example, by configuring the source element to initiate the new content download at the bitrate derived from Hsu's historical tracking mechanism, rather than at the system default lowest bitrate. This integration is architecturally natural and technically straightforward, as the source element is the component that manages download sessions and bitrate selection in Orr's pipeline. (EX1002, ¶143.)

Orr and Hsu are analogous art. Orr is directed to devices receiving and rendering real-time content streams on wireless networks and Internet-connected consumer devices. (EX1005, ¶[0014-16].) Hsu is directed to adaptive bitrate streaming media players on client devices explicitly including "set-top box, Internet-enabled television" and other consumer electronics. (EX1007, 4:50-58.) A POSITA working on content-switch optimization for any such device would routinely consult both references. (EX1002, ¶144.)

1. Claim 6

- a) **6[pre]: The method of any one of claim 1, wherein:**
- b) **6[a]: the transitioning from playback of the first piece of content to the new piece of content comprises initiating playback of the new piece of content using a bitrate used for the playback of the first piece of content; and**

Hsu addresses bitrate maintenance across content transitions. Hsu discloses that, upon initiating playback of new content, the player uses a bitrate derived from the history of prior sessions, specifically the bitrate used during prior playback, as the startup bitrate for the new content: “The technology can use the most recent bitrate as the startup bitrate.” (EX1007, 6:1-5.) Hsu’s claim 1 recites “determining a startup bitrate based on the bitrate history, the startup bitrate comprising an initial bitrate for streaming the media content ... wherein the streaming begins at the determined startup bitrate.” The bitrate “used for playback of the first piece of content” in claim 6 of the ’139 patent corresponds directly to the “most recent bitrate” tracked by Hsu’s player, which is used to initialize the new streaming session. (EX1002, ¶¶145-46.)

c) 6[b]: the determining whether to preserve occurs during playback of the new piece of content using the bitrate.

Hsu teaches continuous monitoring of whether the current bitrate can be maintained during playback. Hsu’s adaptive bitrate player “determin[es] bitrate values used at a plurality of different times for providing the streaming media content” during playback, tracking whether the current bitrate is sustainable given available network bandwidth. (EX1007, claim 1). This continuous monitoring is the mechanism by which the adaptive bitrate system detects that the current bitrate is or is not achievable for the new content. When combined with Orr, the combination

renders obvious this limitation. (EX1002, ¶¶147-48.)

2. Claim 7

- a) **7[pre]: The method of claim 6, further comprising:**
- b) **7[a]: during playback of the new piece of content using the bitrate, detecting whether the new piece of content can maintain playback at the bitrate; and**

Hsu teaches continuous monitoring of whether the current bitrate can be maintained during playback. Hsu's adaptive bitrate player "determin[es] bitrate values used at a plurality of different times for providing the streaming media content" during playback, tracking whether the current bitrate is sustainable given available network bandwidth. (EX1007, claim 1). This continuous monitoring is the mechanism by which the adaptive bitrate system detects that the maintained bitrate is or is not achievable for the new content. (EX1002, ¶¶149-50.)

- c) **7[b]: in response to the detecting that the new piece of content cannot continue playback at the bitrate, adjusting the bitrate.**

Hsu teaches downward bitrate adjustment when the current bitrate cannot be sustained. Hsu describes a capping mechanism that accounts for users moving between higher and lower bandwidth environments, and the standard adaptive bitrate mechanism taught by Hsu lowers the bitrate when download throughput is insufficient to sustain the current level: the player "adjusts the quality of a video stream accordingly" when bandwidth weakens (EX1007, 3:62-4:5) and describes the

system's response to lower-bandwidth environments as selecting a lower bitrate. (EX1007, 6:16-26.) When combined with Orr, the combination renders obvious this limitation. (EX1002, ¶¶151-52.)

3. Claim 13

- a) **The method of claim 1, wherein the source element initiates the playback of the new piece of content using a same bitrate as the first piece of content and adjusts the bitrate in response to detecting that the new piece of content cannot continue playback at the bitrate.**

“the source element initiating playback of the new piece of content using the same bitrate as the first piece of content” Hsu teaches that the player initiates new content streaming at the historically derived startup bitrate: “The technology can use the most recent bitrate as the startup bitrate.” (EX1007, 6:1-5.) Hsu's claim 1 further recites “determining a startup bitrate based on the bitrate history, the startup bitrate comprising an initial bitrate for streaming the media content... wherein the streaming begins at the determined startup bitrate.” (EX1007, claim 1.) (EX1002, ¶¶153-54.)

“the source element adjusting the bitrate when the new content cannot continue playback at that bitrate”: Hsu expressly teaches that the adaptive bitrate system adjusts the quality of the stream downward when bandwidth conditions cannot support the current bitrate: the system “detect[s] a playback device bandwidth

and other factors in real time and adjust[s] the quality of a video stream accordingly.” (EX1007, 3:62-4:5; *see also*, 6:16-26.) (EX1002, ¶155.)

A POSITA integrating Hsu into Orr would be motivated to implement both of these teachings in the source filter of Orr’s pipeline, the component responsible for obtaining content data from the content source and presenting it for downstream processing. A POSITA would have readily incorporate these teaching into Orr to ensure that a new piece of content played smoothly without quality degradation. The source filter is a natural and logical integration point for Hsu’s bitrate teachings within Orr’s architecture. It is the upstream component that controls how content data is retrieved and the component at which an initial bitrate parameter would be set and at which bitrate adjustments would be made in response to network conditions. A POSITA applying Orr’s pipeline reuse architecture to an adaptive HTTP streaming context would understand that the source filter in such an implementation manages the adaptive bitrate download session, and would apply Hsu’s startup bitrate and adjustment teachings to that component as a matter of ordinary skill. (EX1002, ¶156.)

IX. Ground 3: Orr/Wu/Lim

A. Motivation to Combine

A POSITA would have been motivated to integrate Lim’s prefetching of metadata into Orr and Wu, specifically Orr’s multimedia manager and would have

had a reasonable expectation of success in doing so. (EX1002, ¶157.)

A POSITA would be motivated by the fact that Lim's prefetched metadata reduces latency by supplying in advance the content-type information that Orr's format compatibility determination requires. Rather than obtaining this information at the time of the channel switch, a POSITA would have integrated Lim's prefetched metadata to perform Orr's compatibility check instantaneously before receiving the user's selection, thereby reducing the latency typically needed when switching to new content and allowing for a seamless transition to video. This is the straightforward application of prefetching, a well-known technique for reducing access latency, to the specific bottleneck of on-demand content-type retrieval in Orr's channel-switch system. (EX1002, ¶158.)

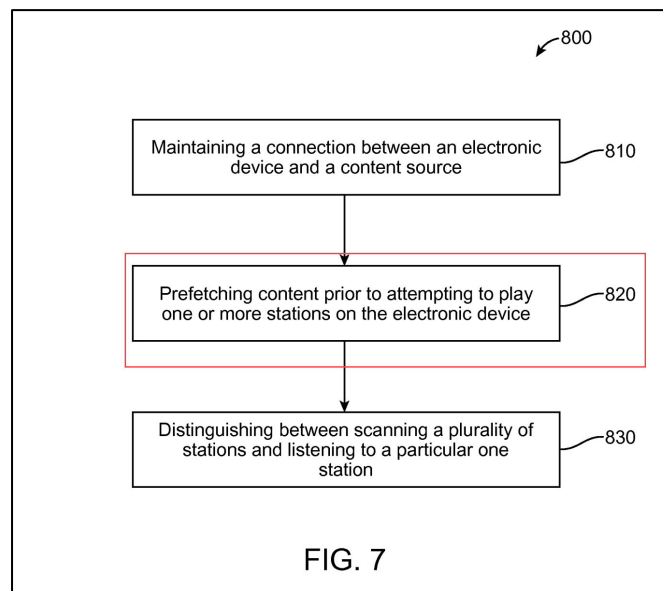
Orr and Lim address sequential and complementary steps in the same content-switch optimization workflow. Orr's channel switch logic performs a format compatibility determination at the time of the channel switch (after the user has made a selection) to decide whether the existing rendering pathway can be reused to reduce latency. (EX1005, ¶¶[0021]; ¶¶[0043-44].) To make this determination, Orr's processing logic must have access to information about the new content's format. (EX1005, ¶¶[0026]; EX1002, ¶159.)

Lim teaches prefetching this kind of content information (metadata including type, format, and codec-relevant data) during playback of current content, before the

user makes a selection. (EX1008, *e.g.*, 3:51-67; 5:23-39; 15:37-46; FIG. 2; claim 1.)

This reduces latency between a user's content selection and the commencement of playback: "when the user changes the content source (*e.g.*, radio stations, video stations, etc.), the mobile device starts playing without reduced buffering time."

(EX1008, 4:1-7.; EX1002, ¶160.)



(EX1008, FIG. 7.)

Instead of waiting for Orr's channel switch logic to perform a format compatibility determination at the time of the channel switch (after the user has made a selection), a POSITA would have been motivated to integrate Lim's prefetching of metadata *before* the user makes a selection to produce faster, higher-quality content transitions between content. (EX1002, ¶161.)

Orr and Lim are analogous art, as they are both directed to reducing the

latency experienced when a user switches between content channels on a playback device, and both are premised on the same insight: that preparing the system in advance of a user's selection reduces the delay after that selection is made. Orr prepares the rendering pathway by evaluating whether existing components can be reused; Lim prepares the content metadata so it is available the moment a selection occurs. A POSITA working on content-switch optimization would recognize these as naturally complementary techniques operating at different points in the same workflow. (EX1002, ¶162.)

A POSITA would have reasonably expected the combination to succeed. Lim provides detailed disclosures regarding implementation of prefetching, and it would have been well-within the skill of an ordinary artisan to implement such functionality into the software or hardware of Orr's multimedia manager. Such a modification would only require ordinary programming knowledge. (EX1002, ¶163.)

1. Claim 5

- a) The method of claim 2, wherein the detecting the codec information for the new piece of content comprises examining prefetched metadata for the new piece of content.**

Claim 5 adds to the codec detection requirement of claim 2 the specific limitation that the codec information is obtained by examining metadata that was prefetched, *i.e.* retrieved and stored locally before the user selects the content.

(EX1002, ¶164.)

Orr in view of Wu teaches the codec and format detection step itself, as set forth above with respect to claim 2: Orr’s channel switch logic determines the content format of the new channel and assesses whether the existing pipeline elements can handle it. (EX1002, ¶165.)

Lim teaches prefetching metadata for candidate next-play content during playback of current content. Lim’s claim 1 recites “prefetching, by the electronic device, content prior to attempting to play at least one station on the electronic device,” and claim 2 expressly specifies that “prefetching content comprises prefetching preview content clips, cover art, and metadata for all on-deck content for the plurality of stations.” (EX1008, claims 1-2.) Lim’s specification confirms this teaching: Lim describes “the client 305 efficiently prefetch[ing] the preview clips, cover art, and metadata for all the on-deck songs of all the stations” using a cold start function while current content plays. (EX1008, 9:6-10.) Lim further describes the prefetch manager module 450 implementing a “Prefetch Song” process that “downloads a song’s preview clip and cover art and stores them in the prefetch inventory 455 and image cache 470,” and a “CheckOnDeckPrefetch” process that triggers a prefetch of the next on-deck song’s data while the current song still has sufficient buffer remaining — *i.e.*, while current content is still playing. (EX1008, 12:44-55.) Lim further describes the metadata prefetched for each song as including

the “song URL and cover art URL,” and notes that the aggregate station binary downloaded at cold start contains “all of the stations’ first song metadata, preview clip, and cover-art.” (EX1008, 10:27-41; 12:37-42.) (EX1002, ¶166.)

A POSITA combining Orr and Lim would implement Orr’s format compatibility check using metadata prefetched per Lim’s method. Lim’s prefetch manager module 450 fetches and stores in the prefetch inventory 455 the metadata for candidate next-play content during playback of current content. When the user makes a channel selection, Orr’s processing logic reads from the locally cached prefetch inventory rather than waiting to obtain format information from the incoming stream, enabling the compatibility determination to be made instantaneously upon selection. (EX1002, ¶167.)

X. Ground 4: Hervey

1. Claim 1

a) 1[pre]

The presentation describes GStreamer as a software framework for implementing playback pipelines on electronic devices that receive and process digital content streams. Hervey describes playbin as a “convenience pipeline” that manages uridecodebin, decodebin, and playsink to receive a URI, decode the content, and present it to a sink, respectively. (EX1009, 4.) (EX1002, ¶¶168-69.)

Playback in GStreamer

- playbin (convenience pipeline)
 - uridecodebin
 - decodebin
 - playsink
 - And some non-reusable code

A POSITA familiar with GStreamer would understand that the framework described in Hervey is a software framework that is deployed on devices in order to receive and render digital content, and would understand the methods described in Hervey as performed by such a device. (EX1002, ¶170.)

b) 1[a]

Hervey describes a GStreamer-based playback framework in which content is played back through a pipeline of functional elements. Hervey describes playbin as a “convenience pipeline” that incorporates uridecodebin, decodebin, and playsink to receive a URI, decode the content, and present it to a sink for rendering. (EX1009, 4.) Hervey further describes uridecodebin as obtaining “the proper source element to use for the URI” and performing “buffering/queueing” of network streams, which constitutes managing the downloading of content data for playback. (EX1009, 13.) The output of uridecodebin feeds into decodebin3, which comprises GstParseBin –

a component that recursively identifies and instantiates the demuxer and parser elements needed to parse the content stream (EX1009, 25), followed by a multiqueue and decoder elements including a video decoder (VDec) and audio decoders (ADec). (EX1009, 24, 26.) These components arranged in series from content acquisition through parsing and decoding to rendering constitute a ***playback pipeline*** within the meaning of this limitation. A POSITA familiar with GStreamer would understand the playbin/decodebin3 stack described in Hervey as a software framework deployed on devices to receive and render digital content through such a pipeline. (EX1002, ¶¶171-72.)

The causing of playback of a first piece of content using this pipeline is reflected throughout Hervey’s description of active playback as the baseline state from which stream switching occurs. For example, Hervey describes the scenario of switching streams during active playback (“we might switch streams later on ... we don’t want the other stream to be drained completely”) confirming that the pipeline is actively playing back a first piece of content prior to any switch. (EX1009, 7.) (EX1002, ¶173.)

c) 1[b]

Hervey describes receiving a new stream selection during active playback of a first stream. For example, Hervey describes receiving a GST_EVENT_SELECT_STREAMS event during ongoing playback which

“[c]ompare[s] available, active, pending and requested streams.” The reference to “active” streams confirms that the selection is received while first content is playing. (EX1009, 28.) (EX1002, ¶174.)

Switching Streams

- GST_EVENT_SELECT_STREAMS
 - Ex : “video1”, “audio2”
 - Compare available, active, pending and requested streams.
 - Need to switch audio1 to audio2
 - Put audio2 in pending selection
 - Set idle probe on multiqueue output of audio1
 - Unlinks from decoder
 - Send GST_EVENT_RECONFIGURE to audio2 multiqueue source pad
 - Audio2 multiqueue source pad will use same logic as before

Hervey further describes the system receiving a new content input while current content plays: “When EOS reaches all multiqueue slot input, emit ‘about-to-finish,’ [u]sers have a chance to change the input of decodebin3.” (EX1009, 32.) A POSITA would understand that the “about-to-finish” flag indicates that the current playing content is about to finish. (EX1002, ¶175.)

Gapless playback in decodebin3

- Can we re-use more elements ?
- Just re-use elements for gapless playback
- When EOS reaches all multiqueue slot input, emit 'about-to-finish'
 - Users have a chance to change the input of decodebin3
 - Earlier than with legacy playbin (EOS on output of decodebin)
- GstParseBin reconfigurable
 - Handles the reconfiguration
 - And we just get new streams... as if it was a demuxer update

d) 1[c]

Hervey teaches transitioning to the newly selected stream in direct response to the stream selection event. Upon receiving `GST_EVENT_SELECT_STREAMS`, `decodebin3` performs the stream comparison and pipeline reconfiguration steps and then begins delivering the new content through the pipeline: “Send `GST_EVENT_RECONFIGURE` to audio2 multiqueue source pad,” and “Audio2 multiqueue source pad will use same logic as before.” (EX1009, 28.) (EX1002, ¶176.) This *transition to the new content* requires selection of the new content, and is therefore *in response to* that selection. (*Id.*)

Switching Streams

- GST_EVENT_SELECT_STREAMS
 - Ex : “video1”, “audio2”
 - Compare available, active, pending and requested streams.
 - Need to switch audio1 to audio2
 - Put audio2 in pending selection
 - Set idle probe on multiqueue output of audio1
 - Unlinks from decoder
 - Send GST_EVENT_RECONFIGURE to audio2 multiqueue source pad
 - Audio2 multiqueue source pad will use same logic as before

The gapless playback slide similarly describes switching to new content upon the “about-to-finish” signal. (EX1009, 32.) (EX1002, ¶177.)

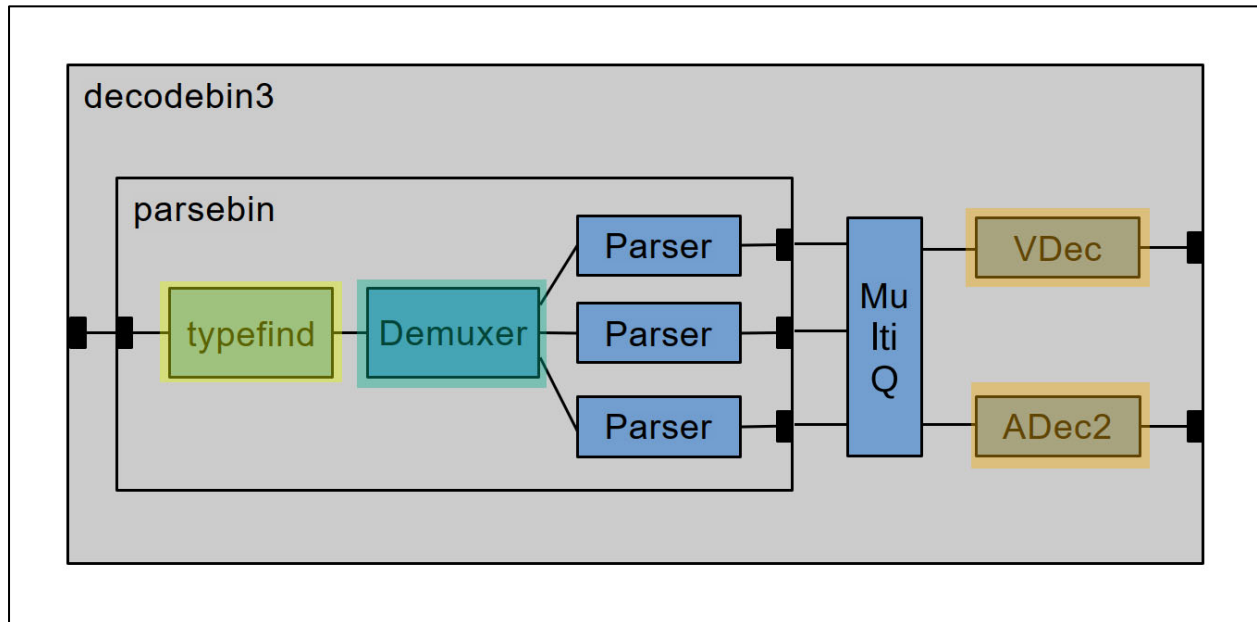
Gapless playback in decodebin3

- Can we re-use more elements ?
- Just re-use elements for gapless playback
- When EOS reaches all multiqueue slot input, emit 'about-to-finish'
 - Users have a chance to change the input of decodebin3
 - Earlier than with legacy playbin (EOS on output of decodebin)
- GstParseBin reconfigurable
 - Handles the reconfiguration
 - And we just get new streams... as if it was a demuxer update

e) 1[d]

Hervey teaches that its pipeline first detects the content type of a new,

incoming stream. Specifically, Hervey teaches that its pipeline contains an element called “parsebin,” which has a pipeline that uses a “single input sink pad” which inputs into a “typefind” element. (EX1009, 24-25.) The “typefind” element connects to a “Demuxer.” (*Id.*) (EX1002, ¶178.)



A POSITA would understand that the “typefind” element detects the content type of the new incoming stream. Contemporaneous GStreamer documentation discloses that the “typefind” element’s purpose is to “detect the media type of an unknown stream.” (EX1019 (GstTypeFind documentation).) (EX1002, ¶179.)

Hervey teaches that once the pipeline’s “parsebin” detects the new content’s content type, it will “[r]ecursively figure[] out ‘decodable’ elements needed,”

including the “[d]emuxer[.]” (EX1009, 25.)¹ Hervey specifically teaches that for “input changes,” the “parsebin” element “re-use[s] elements if possible.” (*Id.*) Thus, Hervey teaches that its “parsebin” element will ***determine whether the new content has the same container type as the old content*** and, if it does, reuse the demuxer. (EX1002, ¶180.)

Hervey teaches determining whether a new stream is compatible with the existing pipeline, which a POSITA would understand as determining whether the new content is of the same content type as the current content. The “Switching Streams” slide describes comparing “available, active, pending and requested streams” to determine what action to take. (EX1009, 28.) (EX1002, ¶181.)

¹ Of note, Hervey explains that its “parsebin” does not determine which decoder to use. (EX1009, 25.) As discussed below in limitations 1[e] and 1[f], a different portion of the pipeline performs that function. (EX1002, ¶180.)

Switching Streams

- GST_EVENT_SELECT_STREAMS
 - Ex : “video1”, “audio2”
 - Compare available, active, pending and requested streams.
 - Need to switch audio1 to audio2
 - Put audio2 in pending selection
 - Set idle probe on multiqueue output of audio1
 - Unlinks from decoder
 - Send GST_EVENT_RECONFIGURE to audio2 multiqueue source pad
 - Audio2 multiqueue source pad will use same logic as before

The “Re-using decoders” slide describes examining the CAPS of the new stream: “STREAM_START / CAPS on multiqueue source pad.” For example, this “[c]hecks if in requested/pending selection,” “[c]heck[s] if there is an output available to be re-used,” and “[c]heck[s] if decoder can accept CAPS.” (EX1009, 29.) (EX1002, ¶182.)

Re-using decoders

- STREAM_START / CAPS on multiqueue source pad
 - Checks if in requested / pending selection
 - Check if there is an output available to be re-used
 - Check if decoder can accept CAPS
 - If so, just link
 - If not, unlink, insert new decoder, link

CAPS in GStreamer is the descriptor of stream format and content type. (EX1018, 2 (“Caps (capabilities) are lightweight refcounted objects describing media types.”) A POSITA would understand that checking whether a decoder can accept a stream’s CAPS determines whether the new content is of the same content type as the content currently being decoded. (EX1002, ¶183.) This is because a decoder of one particular type can only decode audio or video streams of that same type. (*Id.*) Accordingly, checking “if decoder can accept CAPS” refers to checking if the new CAPS contains content that can be decoded by the decoder compatible with the first content, *i.e. has the same content type*.

f) 1[e]

g) 1[f]

Hervey discloses these limitations. These limitations require (1) a “playback pipeline” for the first piece of content that has a source element, a demultiplexer, an audio decoder, and a video decoder, and (2) based on determining the new content has the same content type, determining whether to preserve an audio decoder or a video decoder of the “playback pipeline,” and (3) based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer while replacing the audio and video decoder. (EX1002, ¶¶184-85.)

“Playback pipeline”: Hevey’s decodebin3 architecture explicitly discloses a

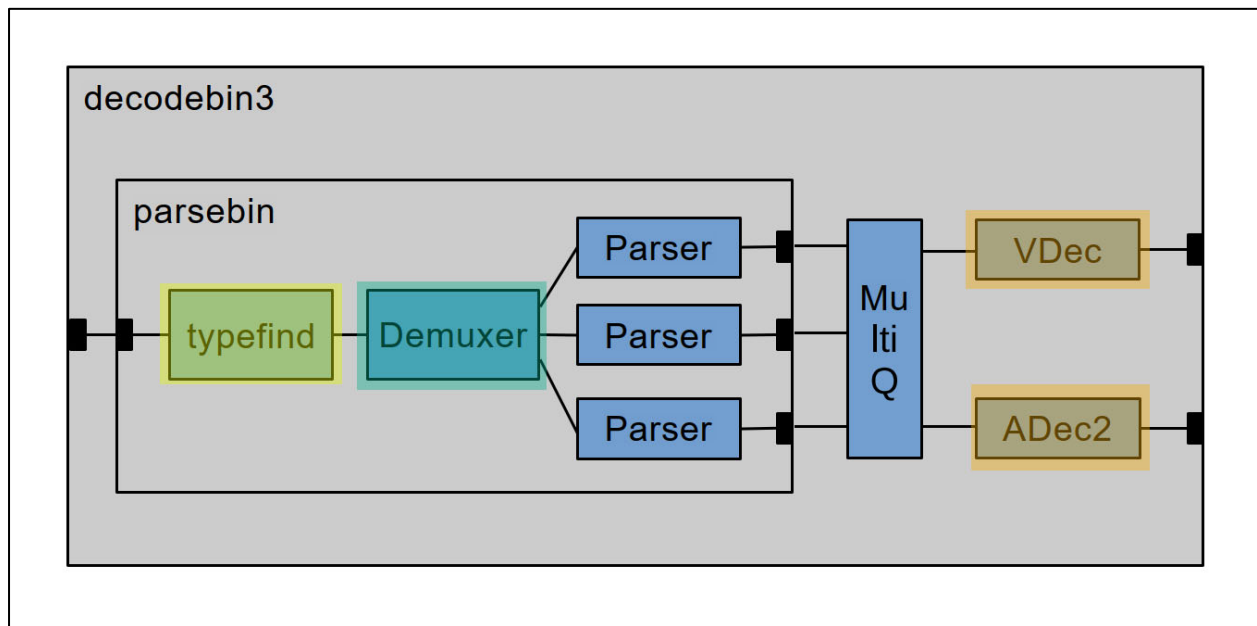
playback pipeline comprising the claimed *source element, demultiplexer, audio decoder, and video decoder*. (EX1002, ¶186.)

“Source element.” Hervey teaches that decodebin3’s *playback pipeline* has a “single input sink pad,” *i.e.* a single element for receiving media input. (EX1009, 25.) Hervey teaches that decodebin3 has a “typefind” element that connects to a “Demuxer.” (EX1009, 24.) (EX1002, ¶187.)

“A demultiplexer.” Hervey discloses that decodebin3’s *playback pipeline* has a “Demuxer,” which is a demultiplexer. (EX1009, 24.) The demuxer outputs a “GstStreamCollection,” which is a “collection” of streams, indicating that it demultiplexes the audio and video signals. (EX1009, 20.) (EX1002, ¶188.)

“An audio decoder.” Hervey discloses that decodebin3’s *playback pipeline* has an “ADec2,” *i.e.* an *audio decoder*. (EX1009, 24.) (EX1002, ¶189.)

“A video decoder.” Hervey discloses that decodebin3’s *playback pipeline* has a “VDec,” *i.e.* a *video decoder*. (EX1009, 24.) (EX1002, ¶190.)



“based on the new piece of content being the same content type as the first piece of content, determining whether an audio decoder or a video decoder of the playback pipeline used to play back the first piece of content should be preserved”:

As discussed above in limitation 1[d], Hervey teaches that if the new content’s container type (*i.e.* **content type**) is the same as the old content, Hervey will reuse the demultiplexer. Subsequently, Hervey teaches that it will determine if **the audio and video decoders used for the old content should be preserved**. Specifically, Hervey teaches its pipeline will “[r]e-use decoders,” specifically: “Check if decoder can accept CAPS. If so, just link. If not, unlink, insert new decoder, link.” (EX1009, 29.) (EX1002, ¶191.)

Re-using decoders

- STREAM_START / CAPS on multiqueue source pad
 - Checks if in requested / pending selection
 - Check if there is an output available to be re-used
 - Check if decoder can accept CAPS
 - If so, just link
 - If not, unlink, insert new decoder, link

This is a determination of whether to preserve the existing audio and video decoders. If the decoders can decode the new content, the decoder is preserved (“just link”); if not, it is replaced (“unlink, insert new decoder”). (EX1002, ¶192.)

“based on a determination that the audio decoder and the video decoder should not be preserved, preserving a source element and a demultiplexer of the playback pipeline while replacing the audio decoder and the video decoder”:

As just discussed, Hervey teaches that based on whether the “decoder can[not] accept CAPs,” *i.e. a determination that the audio and video decoders should not be preserved*, it will “unlink” the old decoders and “insert new decoder[s],” *i.e. replace the audio and video decoder*. (EX1009, 29.) (EX1002, ¶193.)

As discussed above for limitation 1[d], Hervey discloses preserving the ***demultiplexer*** if the new content’s container type (*i.e. content type*) is the same as the old content’s container type. (EX1002, ¶194.)

Hervey also discloses preserving the ***source element***. Specifically, Hervey

discloses “[o]ne single input sink pad.” (EX1009, 25.) In addition, Hervey teaches that it attempts to “[r]e-use as many elements as possible.” (EX1009, 34.) (EX1002, ¶195.)

2. Claim 2

a) 2[pre]/2[a]

Hervey teaches detecting codec information for new content. In GStreamer, a stream’s CAPS describes the media type of that stream. (EX1018, 2 (“Caps (capabilities) are lightweight refcounted objects describing media types.”)). Hervey describes examining the CAPS of new streams at the multiqueue output: “STREAM_START / CAPS on multiqueue source pad.” “Check if in requested/pending selection.” “Check if there is an output available to be re-used.” “Check if decoder can accept CAPS.” (EX1009, 29.) (EX1002, ¶¶196-97.)

Re-using decoders

- STREAM_START / CAPS on multiqueue source pad
 - Checks if in requested / pending selection
 - Check if there is an output available to be re-used
 - Check if decoder can accept CAPS
 - If so, just link
 - If not, unlink, insert new decoder, link

Because a decoder of one particular media type can only decode audio or video streams of that same type (for example, H.264, AAC), a POSITA would

understand that examining a stream's CAPS to determine decoder compatibility detects codec information in the stream's CAPS to determine the decoder needed to decode the stream. (EX1002, ¶198.)

b) 2[b]

The “Re-using decoders” slide describes checking whether the existing decoder can accept the CAPS of the new stream: “Check if decoder can accept CAPS. If so, just link.” (EX1009, 29.) This check, comparing the existing decoder's capabilities against the CAPS (codec requirements) of the new stream, is the determination recited in this limitation. If the decoder's capabilities match the CAPS, the decoder is “of the types indicated by the codec information.” A POSITA would recognize this as a disclosure of this limitation. (EX1002, ¶199.)

Re-using decoders

- STREAM_START / CAPS on multiqueue source pad
 - Checks if in requested / pending selection
 - Check if there is an output available to be re-used
 - Check if decoder can accept CAPS
 - If so, just link
 - If not, unlink, insert new decoder, link

3. Claim 3

Hervey expressly teaches full pipeline preservation when the CAPS check succeeds. When “decoder can accept CAPS,” the result is “just link.” Meaning, the

existing decoders are preserved in place and connected directly to the new stream. (EX1009, 29.) Hervey further teaches that, when the video and audio streams are format-compatible, the demuxer and source elements are also preserved: “Video stream remains the same, but other streams change. No need to remove/add pad for the video stream. No breakage in data stream.” (EX1009, 30.) (EX1002, ¶¶200-01.)

Demuxer handling

- Decodebin3 backwards compatible with current behaviour
- Demuxers (ex: tsdemux) can be smarter
 - Ex: Video stream remains the same, but other streams change
 - No need to remove/add pad for the video stream
 - No breakage in data stream
- Any element can add/remove streams at any time
 - Ex: [CC] parsing in video parsers

4. Claim 4

a) 4[pre]/4[a]

As discussed above with respect to claim 2, Hervey teaches examining the CAPS of each stream at the multiqueue output to determine whether the existing decoder can accept the new stream’s CAPS. (EX1009, 29.) When the existing decoder cannot accept the CAPS, *i.e.* the existing decoder is of a different type than that indicated by the codec information, Hervey teaches: “unlink, insert new decoder, link.” (EX1009, 29.) Hervey’s decodebin3 architecture maintains separate

decoder slots for audio and video streams, as shown in slide 24, which depicts distinct VDec and ADec components in the pipeline. (EX1009, 24.) Slide 26 further confirms that decoder selection occurs independently for each stream type at the multiqueue output based on that stream's CAPS: “Only thing that (might) need to be plugged is a decoder. By default only expose/decode one stream of each type.” (EX1009, 26.) (EX1002, ¶¶202-23.)

b) 4[b]

Hervey teaches obtaining new decoders when the existing decoders are incompatible. Following the “unlink” step, Hervey teaches “insert new decoder.” A new decoder is instantiated and inserted into the pipeline. (EX1002, ¶¶204-05.)

Re-using decoders

- STREAM_START / CAPS on multiqueue source pad
 - Checks if in requested / pending selection
 - Check if there is an output available to be re-used
 - Check if decoder can accept CAPS
 - If so, just link
 - If not, unlink, insert new decoder, link

The new decoder is selected based on the CAPS of the new stream: the GstStreamCollection mechanism and GStreamer's auto-plugging logic identify the appropriate decoder type based on the stream's codec descriptor. (*See, e.g.*, EX1009, 20-21, 31.) (EX1002, ¶206.)

5. Claim 8

a) 8[pre]/8[a]

Hervey teaches full pipeline deconstruction when content types are incompatible through decodebin3's stream switching and decoder management logic. The “Re-using decoders” slide teaches that when the existing decoder cannot accept the new stream’s CAPS (meaning the encoding types are incompatible), decodebin3 executes “unlink, insert new decoder, link.” (EX1002, ¶¶207-08.)

Re-using decoders

- STREAM_START / CAPS on multiqueue source pad
 - Checks if in requested / pending selection
 - Check if there is an output available to be re-used
 - Check if decoder can accept CAPS
 - If so, just link
 - If not, unlink, insert new decoder, link

The “unlink” step releases the existing decoder from the pipeline. For cases of fundamental content-type incompatibility extending beyond the decoders to the demuxer and source elements (such as the new content having a new container type), Hervey’s “Demuxer handling” slide teaches that when streams change at the demuxer level (meaning the new content requires a different demuxer configuration) decodebin3 handles that reconfiguration by releasing and replacing the affected elements: “Any element can add/remove streams at any time.” (EX1009, 30.)

(EX1002, ¶209.)

Demuxer handling

- Decodebin3 backwards compatible with current behaviour
- Demuxers (ex: tsdemux) can be smarter
 - Ex: Video stream remains the same, but other streams change
 - No need to remove/add pad for the video stream
 - No breakage in data stream
- Any element can add/remove streams at any time
 - Ex: [CC] parsing in video parsers

Hervey further teaches through the GstParseBin mechanism that when input changes require complete reconfiguration, GstParseBin handles the full pipeline reconfiguration, releasing existing elements and acquiring new ones: “GstParseBin reconfigurable; Handles the reconfiguration; And we just get new streams.”

(EX1009, 32.) (EX1002, ¶210.)

Gapless playback in decodebin3

- Can we re-use more elements ?
- Just re-use elements for gapless playback
- When EOS reaches all multiqueue slot input, emit 'about-to-finish'
 - Users have a chance to change the input of decodebin3
 - Earlier than with legacy playbin (EOS on output of decodebin)
- GstParseBin reconfigurable
 - Handles the reconfiguration
 - And we just get new streams... as if it was a demuxer update

A POSITA would understand that in the scenario where new content has a different container and encoding format, decodebin3's reconfiguration mechanism releases all of these pipeline elements. (EX1002, ¶211.)

b) 8[b]

Hervey's GstParseBin reconfiguration mechanism handles rebuilding of the pipeline: "GstParseBin reconfigurable. Handles the reconfiguration. And we just get new streams." (EX1009, 32.) (EX1002, ¶¶212-13.)

Gapless playback in decodebin3

- Can we re-use more elements ?
- Just re-use elements for gapless playback
- When EOS reaches all multiqueue slot input, emit 'about-to-finish'
 - Users have a chance to change the input of decodebin3
 - Earlier than with legacy playbin (EOS on output of decodebin)
- GstParseBin reconfigurable
 - Handles the reconfiguration
 - And we just get new streams... as if it was a demuxer update

The phrase “we just get new streams” reflects that GstParseBin, upon reconfiguration, automatically identifies and instantiates the elements needed to process the new content, acquiring resources applicable to the new content type. Hervey further teaches through the GstParseBin description that it “recursively figures out ‘decodable’ elements needed. Demuxers, depayloaders, parsers... but not decoders” (EX1009, 25) for any given input stream, and that the output of GstParseBin feeds the multiqueue at which new decoders are selected and instantiated based on the stream’s CAPS. (EX1009, 26.) (EX1002, ¶214.)

GstParseBin

- One single input sink pad
- Recursively figures out “decodable” elements needed
 - Demuxers, depayloaders, parsers ... but not decoders
- No queueing
- Creates/Posts GstStreamCollection/GstStream if the element didn't create it
 - All pipelines can get new API support \o/
- Longer term : Reconfigurable
 - On input changes, re-use elements if possible, else switch

decodebin3

- Output of GstParseBin(s) fed to multiqueue
 - Parsed Elementary Streams
 - Multiqueue “slots” (sink + src pad) are typed (audio, video,...)
 - If available slot of proper type, re-use, else create new one
- Selection is done post-multiqueue
 - Only thing that (might) need to be plugged is a decoder
 - By default only expose/decode one stream of each type
 - But you can also expose everything...
 - Or just what you selected via GST_EVENT_SELECT_STREAMS

6. Claim 9

Hervey expressly identifies HLS and DASH as content types that decodebin3’s stream selection architecture is designed to handle. Slide 22 identifies

“Alternate HLS/DASH” as an example of hidden streams that can be activated using the GST_EVENT_SELECT_STREAMS API. (EX1009, 22.) (EX1002, ¶¶215-16.)

Selecting Streams

- GST_EVENT_SELECT_STREAMS
 - List of stream-id to be selected
- Elements can now reliably know which streams will be needed downstream
 - Avoid processing (decoding anyone ?)
 - Hidden streams to activate Alternate HLS/DASH, switching DVB channel, ...)
- Not tied to playbin
 - Custom pipeline ? Win again

7. Claim 10

Hervey’s playbin/decodebin3 stack collectively performs all of the operations recited in claim 1 and corresponds to the “player” as described in the ’139 patent. Playbin is the top-level player component that manages stream listing and selection and handles content switching: “Uses uridecodebin. Output of uridecodebin to inputselector(s) (to switch streams),” and “[a]llows stream listing/selection.” (EX1009, 14.) (EX1002, ¶¶217-18.)

PlayBin

- Uses uridecodebin
 - Output of uridecodebin to inputselector(s) (to switch streams)
 - Into playsink
- Allows stream listing/selection
- Extra-uri (for subtitle in separate URI)
 - Creates a new parallel uridecodebin
- Gapless support (about-to-finish / next-uri)
 - Prerolls another uridecodebin and switches

Playbin receives content selections and manages the transition to new content through its gapless and stream-switching mechanisms. (EX1009, 14; “Gapless support (about-to-finish / next-uri). Prerolls another uridecodebin and switches”). The content-type determination and pipeline preservation decision are performed by decodebin3, which operates within and under the management of playbin: “Shifted most of the logic to decodebin3 (stream selection, gapless, multi-input support).” (EX1009, 33.) (EX1002, ¶219.)

This passage also describes that stream selection and gapless switching logic (including the preservation determination) is part of the integrated player stack managed by playbin. (EX1002, ¶220.)

8. Claim 11

See claim 8. (EX1002, ¶221.)

9. Claim 12

See claim 8. (EX1002, ¶222.)

10. Claim 14

a) 14[pre]

b) 14[a]

Hervey expressly references the CPU as a hardware resource consumed by the media playback pipeline. The “Summary” slide identifies reducing “CPU” usage as a primary goal of decodebin3: “Avoid as much unneeded processing as possible (CPU, MEM, I/O, ...).” (EX1009, 34.) (EX1002, ¶¶223-225.)

Summary

- Avoid as much unneeded processing as possible (CPU, MEM, I/O, ...)
- Re-use as many elements as possible
- Still WIP

The “Decodebin2: Stream Switching” slide similarly identifies “Useless CPU/MEM/IO usage” as a problem with the prior decodebin2 architecture that decodebin3 is designed to address. (EX1009, 7.) (EX1002, ¶226.)

Decodebin2: Stream Switching

We might switch streams later on ...

We don't want the other stream to be drained completely

Else we will have a gap when switching
multiqueue throttles unlinked streams

But ... we're decoding everything :(

Useless CPU/MEM/IO usage

These references to CPU confirm that the playback pipeline described in Hervey executes on a CPU, a hardware processor, and that the operations performed by decodebin3's stream switching, codec detection, and pipeline reuse logic consume CPU resources. A POSITA would understand these explicit references to CPU usage as disclosing that Hervey's system operates on one or more hardware processors within the meaning of this claim. (EX1002, ¶227.)

c) 14[b]

Hervey expressly references memory (MEM) as a hardware resource consumed and managed by the decodebin pipeline: "Avoid as much unneeded processing as possible (CPU, MEM, I/O, ...)." (EX1009, 34.) (EX1002, ¶¶228-29.)

Summary

- Avoid as much unneeded processing as possible (CPU, MEM, I/O, ...)
- Re-use as many elements as possible
- Still WIP

The “Decodebin2: Stream Switching” slide (EX1009, 7) and the "Chain pitfalls" slide identifies "Increased memory usage (multiqueue)" and "Increased CPU usage (duplicated elements)" as specific resource problems that decodebin3's design addresses by reducing the number of instantiated pipeline elements. (EX1009, 11.) (EX1002, ¶230.)

Chain pitfalls

- New “pending” DecodeGroup
 - Increased memory usage (multiqueue)
 - Increased CPU usage (duplicated elements)
- Input and output of decodebin is no longer fully linked
 - Ex : seek event ending nowhere :(
- Want to just add/remove a stream ?
 - Still need to re-create a new bag of source pads
 - Breaks playback (switch video decoder in GOP)
 - Note: wasn't possible in 0.10 due to SEGMENT limitations

These references confirm that decodebin3’s pipeline elements are software

components that occupy memory and execute on the CPU. A POSITA would understand that software components of this kind that consume CPU and memory resources are necessarily stored as instructions on a storage device that, when executed by the CPU, configure it to perform the pipeline operations described in Hervey. (EX1002, ¶231.)

a) 14[c]-14[h]

See limitations 1[a]-1[f]. (EX1002, ¶¶232-37.)

11. Claim 15

See claim 2. (EX1002, ¶238.)

12. Claim 16

See claim 3. (EX1002, ¶239.)

13. Claim 17

See claim 4. (EX1002, ¶240.)

14. Claim 18

See claim 8. (EX1002, ¶241.)

15. Claim 19

a) 19[pre]

See limitations 14[a] and 14[b]. (EX1002, ¶242.)

b) 19[a]-19[f]

See limitations 1[c]-1[f]. (EX1002, ¶¶243-248.)

XI. Ground 5: Hervey/Hsu

A. Motivation to Combine

A POSITA would have been motivated to integrate Hsu's bitrate initialization into Hervey and would have had a reasonable expectation of success in doing so. (EX1002, ¶249.) A POSITA would have been motivated by the known benefits of reducing latency and degradation of the stream quality by reusing the current bitrate for new content, as changing to the default bitrate for new content requires time and resources, and also causes the new content to initially stream at a lower quality. (*Id.*)

A POSITA would implement Hsu's bitrate initialization teaching in Hervey's source element, configuring it to begin downloading new content at the previously achieved bitrate rather than the system default lowest bitrate. This integration requires no architectural changes to either reference and produces the predictable result of a faster, higher-quality content transition. (EX1002, ¶250.)

A POSITA would have been motivated to address the same fundamental user experience problem that Hervey and Hsu identify: the latency experienced when switching between pieces of content on a streaming playback device. Hervey describes reusing pipeline elements to avoid the overhead of teardown and reconstruction when switching streams, which reduced buffering and latency to provide reduced gaps and smoother transitions between content. (EX1009, 7 (“We might switch streams [but w]e don’t want the other stream to be drained completely.

Else we will have a gap when switching.”); slide 11 (just adding/removing a stream “[b]reaks playback.”); 23 (the new decodebin3 “[r]e-use[s] as many elements as possible,” and “[r]educe[]s buffering.”); EX1002, ¶251.)

Hsu addresses the bitrate initialization component, eliminating the quality degradation and latency caused by changing to the default lowest bitrate at the start of each new session. (EX1007, 5:33-39, FIG. 2). Instead, Hsu teaches using the current bitrate for a new piece of content, allowing for a seamless transition and increase in streaming quality. (*e.g.*, EX1007, 6:1-5.) Hervey and Hsu’s solutions are complementary that a POSITA would naturally implement together, and have a reasonable expectation of success. (EX1002, ¶252.) A POSITA combining Hervey and Hsu would implement Hsu’s bitrate initialization teaching in the source element of Hervey’s pipeline by configuring the source element to initiate the new content download at the bitrate derived from Hsu’s historical tracking mechanism, rather than at the system default lowest bitrate. (EX1002, ¶253.)

The combination of Hervey and Hsu requires no architectural changes to either reference and produces the predictable result of a faster, higher-quality content transition. Moreover, the modification is well-within the skills of an ordinary artisan, as bitrate initialization was well-known and straightforward to implement. Neither reference teaches away from the combination. A POSITA would therefore have a reasonable expectation of success. (EX1002, ¶253.)

In addition, both Hervey and Hsu are directed to adaptive HTTP streaming contexts. Hervey expressly addresses HLS and DASH pipelines. (EX1009, 22.) Meanwhile, Hsu is directed to HTTP-based adaptive bitrate streaming. (EX1007, 3:62-4:27.) A POSITA working in adaptive HTTP streaming would consult both references as directly applicable to the same technical context, and would recognize the bitrate initialization teaching of Hsu as a natural complement to the pipeline reuse teaching of Hervey. (EX1002, ¶254.)

1. Claim 6

For the reasons discussed above for Ground 1, Claim 6, Hsu teaches this limitation. When combined with Hervey, the combination renders obvious this limitation. (EX1002, ¶¶255-58.)

2. Claim 7

For the reasons discussed above for Ground 1, Claim 7, Hsu teaches this limitation. When combined with Hervey, the combination renders obvious this limitation. (EX1002, ¶¶259-62.)

3. Claim 13

For the reasons discussed above for Ground 1, Claim 13, Hsu teaches this limitation. (EX1002, ¶¶263-65.)

As noted with respect to claims 6 and 7, a POSITA would implement Hsu's bitrate initialization and adjustment teaching in Hervey's source element. The source

element initiates the new session at the historically derived bitrate (e.g., EX1009, 6:1-5, claim 1, FIG. 1B) and adjusts downward when the bitrate cannot be maintained. (e.g., EX1009, 3:66-4:5.) This claim is thus rendered obvious by the combination of Hervey and Hsu for the same reasons set forth above. (EX1002, ¶¶266.)

XII. Ground 6: Hervey/Lim

A. Motivation to Combine

A POSITA would have been motivated to integrate Lim's prefetching of metadata into Hervey and would have had a reasonable expectation of success in doing so. (EX1002, ¶267.)

A POSITA would immediately recognize that Lim's prefetched metadata reduces latency by supplying in advance the stream and codec information needed when Hervey switches to new HLS or DASH content. Rather than obtaining this information at the time of switching to HLS or DASH content, a POSITA would have integrated Lim's prefetched metadata to obtain the stream and codec information before switching, thereby reducing the latency typically needed when switching to new content and allowing for a seamless transition to video. This is the straightforward application of prefetching, a well-known technique for reducing access latency. (EX1002, ¶268.) Hervey and Lim are both motivated by the same user-facing problem: the delay experienced when a user switches from one piece of

content to another on a playback device. (EX1002, ¶271.) Hervey describes reusing pipeline elements to avoid the overhead of teardown and reconstruction when switching streams, which reduces buffering and latency to provide reduced gaps and smoother transitions between content. (EX1009, 7 (“We might switch streams [but w]e don’t want the other stream to be drained completely. Else we will have a gap when switching.”); 11 (just adding/removing a stream “[b]reaks playback.”); 23 (the new decodebin3 “[r]e-use[s] as many elements as possible,” and “[r]educ[e] buffering.”).) Lim is similarly directed to reducing the delay between a user’s content selection and the commencement of playback: “when the user changes the content source (e.g., radio stations, video stations, etc.), the mobile device starts playing without reduced buffering time.” (EX1008, 4:1-7.) Both references thus share the same design objective, minimizing perceived latency at content switch time, and a POSITA working on that problem would naturally consider both as relevant to the same design challenge. (EX1002, ¶269.)

Hervey identifies a limitation in its stream selection architecture that Lim’s prefetching teaching directly addresses. The “Pitfalls of Stream Selection” slide (EX1009, 15) identifies the potential absence of pads (e.g., HLS/DASH) as a problem. A POSITA reading both references together would immediately recognize that Lim’s prefetching architecture fills an information gap that Hervey itself identifies, and have a reasonable expectation of success. (EX1002, ¶¶270-72.) Lim

provides detailed disclosures regarding implementation of prefetching, and it would have been well-within the skill of an ordinary artisan to implement such functionality into the software or hardware of Orr's multimedia manager. Such a modification would only require ordinary programming knowledge, and therefore be reasonably expected to succeed. (*Id.*)

1. Claim 5

Hervey teaches the codec detection step itself, as set forth above with respect to claim 2: examining the CAPS of the new stream to determine decoder compatibility. (EX1009, 29.) Hervey teaches obtaining this CAPS information at the time the stream data arrives at the multiqueue output, rather than from prefetched metadata. (EX1002, ¶¶273-75.)

For the reasons discussed above for Ground 1, Claim 5, Lim teaches prefetching metadata. (EX1002, ¶276.) When combined with Hervey, the combination renders obvious this limitation. (EX1002, ¶¶277-78.)

Respectfully submitted,

Date: May 22, 2026

/s/ James M. Glass

James M. Glass (Reg. No. 46729)
QUINN EMANUEL URQUHART &
SULLIVAN LLP
295 5th Avenue, 9th Floor
New York, NY 10016
Email: jimglass@quinnemanuel.com
Phone: 212-849-7142

Petition for *Inter Partes* Review
U.S. Patent No. 11,825,139

Fax: 212-849-7100

CERTIFICATION UNDER 37 C.F.R. § 42.24

Under the provisions of 37 C.F.R. § 42.24, the undersigned hereby certifies that the word count for the foregoing Petition for *inter partes* review (excluding the table of contents, table of authorities, mandatory notices, certificate of service or word count, and appendix of exhibits or claim listing) totals 13,916 words, which is within the word limit allowed under 37 C.F.R. § 42.24(a)(i.)

Date: May 22, 2026

/s/ James M. Glass

James M. Glass (Reg. No. 46729)
QUINN EMANUEL URQUHART &
SULLIVAN LLP
295 5th Avenue, 9th Floor
New York, NY 10016
Email: jimglass@quinnemanuel.com
Phone: 212-849-7142
Fax: 212-849-7100

CERTIFICATE OF SERVICE

Pursuant to 37 C.F.R. §§ 42.6(e), 42.105(a), the undersigned hereby certifies service on the PO of a copy of this Petition and its respective exhibits at the official correspondence address for the attorneys of record for the '999 patent as shown in USPTO PAIR via FedEx:

Kudelski Group & MAIER & MAIER, PLLC
345 SOUTH PATRICK STREET
ALEXANDRIA, VA 22314
UNITED STATES

Courtesy copies were also sent via FedEx and electronic mail to Patent Owner's counsel of record in *OpenTv, Inc. v Pinterest, Inc.*, Case No. 1:25-cv-01404-JCG (D. Del.):

RKOpenTVPinterest@RobinsKaplan.com
TWaller@RobinsKaplan.com
CMorton@RobinsKaplan.com
BLinden@RobinsKaplan.com
akipp@robinskaplan.com

Date: May 22, 2026

/s/ James M. Glass

James M. Glass (Reg. No. 46,729)
QUINN EMANUEL URQUHART &
SULLIVAN LLP
295 5th Avenue, 9th Floor
New York, NY 10016
Email: jimglass@quinnemanuel.com
Phone: 212-849-7142
Fax: 212-849-7100