Delivering Layered Object-Based Media using WebAssembly with Selective Cloud Rendering

Barry Porter
School of Computing and Communications
Lancaster University, UK
b.f.porter@lancaster.ac.uk

Rajiv Ramdhany
BBC R&D
Rajiv.Ramdhany@bbc.co.uk

Nicholas Race
School of Computing and Communications
Lancaster University, UK
n.race@lancaster.ac.uk

Abstract

Traditional media is delivered using segmented video, with variable bitrates, over protocols such as MPEG-DASH. This works well for media experiences with a single, or very few, pre-produced variants. When the number of possible variants increases, however, it results in an explosion of pre-produced whole-experience videos in a one-per-variant relationship; this in turn causes high storage costs and poor re-use of otherwise separable media assets. The paradigm of *object-based media* (OBM) offers a solution by keeping media entities distinct after the production phase, allowing them to be combined flexibly at the point of consumption. In this research we examine the delivery and playback of OBM using a novel WebAssembly-based media player running in the browser. This approach allows the selective render offload of different elements of an experience into the edge/cloud, by seamlessly migrating those pieces of code from the browser. Using three diverse exemplars of OBM, we (i) present a common meta-data format to capture flexible experiences, and (ii) measure the performance of our delivery pipeline in a range of on-device and compute-offload scenarios. As far as we are aware this is the first such study of generalised OBM media delivery.

ACM Reference Format:
Barry Porter, Rajiv Ramdhany, and Nicholas Race. 2026. Delivering Layered Object-Based Media using WebAssembly with Selective Cloud Rendering. In *ACM Multimedia Systems Conference 2026 (MMSys '26), April 04–08, 2026, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3793853.3795762

1 Introduction

Object-based media (OBM) is an emerging paradigm which offers a high level of end-user flexibility and configurability when consuming a piece of media. In contrast to traditional streamed media delivery, OBM separates out elements of a media experience so they can be combined in a variety of ways by the viewer. Elements may include different presenters (e.g., for hearing and non-hearing viewers), information overlays, picture-in-picture elements, camera angles, or generative scene aspects such as relighting. Each media frame is thus composed dynamically at the point of playback, using a suite of renderers for each object type selected for inclusion.

The delivery and render of OBM presents a set of novel challenges compared to traditional streamed content. These include methods to express an object-based experience and its configurability, including the way in which viewers can interact with and control that configurability; the way in which objects are delivered from streaming providers to viewers during playback; and the way each frame is rendered to the viewer. The rendering element in particular offers a range of options, from end-user devices rendering all elements of each frame, to selectively offloading part or all of the rendering to a cloud service for lower-capability playback devices.

We present an end-to-end systems-level study of layered OBM as a delivery paradigm; as far as we are aware this is the first such empirical study of its kind. To underpin our study we present a generalised metadata format to express layered object-based experiences, in terms of *source* and *transform* layers, and associated GUI controls that interact with an experience's variability. Media is rendered using a novel player in WebAssembly [?], allowing the player to run in modern web browsers while also allowing selective render offload to native cloud locations, using the same codebase in both locations – where the cloud version has access to hardware-accelerated media coding and highly parallel processing.

We present our methodology, including our player and metadata formats, using three object-based exemplars with diverse requirements. While our work is underpinned by delivery mechanisms that suit the cache and CDN realities of traditional media delivery infrastructure at streaming services like BBC iPlayer, our approach represents a major departure from approaches deployed today. For the purpose of this paper, each of our experiments therefore uses a single client (on a range of client device classes) operating together with our media server and compute offload cluster to demonstrate informative results in a controlled setting. Our results show that:

- All of our playback devices can render a single H.264 video at full HD in WebAssembly and maintain a median framerate of 25fps. When rendering a full OBM experience, however, multiple devices fail to maintain this framerate for some experience types, showing the need to offload compute to deliver a single pre-composited video stream.
- Our offload servers, using commodity hardware, deliver render of OBM experiences into composited HD video at twice real-time, including full decoding of multiple layers and re-encoding into a single video. Both CPU *and* bandwidth requirements at the client are reduced when offloading to the cloud, since segments of a single video are being downloaded, rather than assets for multiple videos and animations.
- The overall content storage costs tend to be lower for OBM vs. pre-baked versions of highly-flexible media experiences, ranging from being 90% less storage size to 20% less, depending on the exemplar; the bandwidth delivery cost for

client-rendering can range from slightly (17%) to significantly higher (94%), depending on the experience, while offloaded rendering has the same bandwidth cost to the client as traditional streaming of segmented media.

Our results serve as a baseline for OBM rendering and offloading and allow us to consider likely next steps; in future work we intend to conduct large-scale experiments to study the distributed pipelines of media delivery and offloading for many simultaneous clients.

2 Related Work

In this section we discuss closely-related work in media composition; streaming and rendering; and OBM itself. We observe that delivery and rendering mechanisms for OBM, in which media objects are first-class entities, have received little attention to date, prompting our research towards offloadable OBM playback.

Media Composition and Generation. Research on media composition and generation includes novel view generation, human-object interaction modelling, dynamic relighting, and tiled media for video conferencing. These works show a move towards increasingly dynamic media, including both composition of existing media and generation of new media, though many are used by producers rather than on end-user devices during playback.

Zhang et al. present methods for novel view generation [?], in which 2D image inputs are used to construct 3D scene meshes; these meshes support the potential generation of novel camera angles chosen by a user when no physical camera was in that position.

Synthesis of human-like animations allows the generation of custom presenters or other animated characters without extensive frame-by-frame input; Qin et al. present a method to convert still image portraits into animated presenters [?], while Cai et al study the use of text descriptions to generate realistic interactions between human characters and various objects in a 3D space [?].

Choi et al. examine the topic of dynamic relighting using a 3DGC approach [?], in which media of a human is inserted into an arbitrary scene, with the human lit appropriately to present a seamless effect of the two media entities being in the same physical space. The general ability to convincingly relight media objects offers a wide range of novel media composition approaches [? ? ?].

Finally, a range of research has recently explored optimisations available in the composition of tiled media – a media presentation format used extensively in video conferencing applications. Gunkel et al., for example, present an approach to dynamically tiling visual media [?] without a full transcoding step. Instead of decoding media and then re-encoding it, this research demonstrates the use of tiling encoders such as VP9 or AV1 to directly transplant tiles in encoded frames with new visual content.

Streaming and Rendering. Research in streaming of media is dominated by adaptive bitrate methods, generally studied in the context of traditional media delivery techniques. On the rendering side, recent research has tended to centre on pixel streaming or partial offloading for video game applications, and on neural rendering solutions to recover from packet loss or scale up a resolution.

In general media streaming, Wang et al. present an approach to real-time streaming, where a client is encoding frames in real-time for dissemination to viewers, where the client selectively drops frames to encode, and the server uses DNN-based interpolation to recover those frames [?].

On the theme of bitrate adaptation in particular, Huang et al. examine an adaptive bitrate approach to Quality-of-Experience [?] which integrates user feedback into decision-making, rather than relying on an aggregated mean opinion score. Chen et al. consider bitrate planning during a stream to avoid repeated bitrate switches, demonstrating a planning algorithm with polynomial complexity [?]. Agarwal et al. examine the use of data-driven bitrate learning in videoconferencing applications, finding that the use of real-time telemetry combined with alternative action reasoning can improve general decision-making on bitrate selection [?].

On rendering techniques that go beyond DASH-based delivery of segmented video, Artioli et al. examine the use of 3D background models overlaid with video of human performers as a bandwidth-reduction technique [?]. Instead of continuously sending high-resolution frames that include the background, such as an ice rink in coverage of an ice skating competition, a single 3D model of that background is sent to the client device instead, with transformations (camera pans, zooms, etc.) applied to that model in synch with the way in which the original background was framed. This opens new research directions in 3D scene composition, though requires significant render capability on the client device.

The partial offloading of processes required for multi-media delivery has seen a range of recent research: examples include Espindola et al. who examine WebAssembly-offloading of Extended Reality services [?]; Döka et al. who studied remote-rendering in mixed-reality applications [?]; and Jiang et al. who studied the viability of offloading head-tracking tasks for extended reality applications [?]. Yan et al. demonstrate the use of edge compute to assist with computer vision workflows, in which video semantic segmentation is carried out by IoT devices in close proximity to the user [?]. Carter et al. examine the use of real-time pixel streaming of composited media which is fully rendered in the cloud, with each frame delivered to the client using a just-in-time model [?].

OBM Research. Specific research in OBM is more limited, with recent efforts focusing on QoE modelling and on authoring pipelines for highly personalize-able audio experiences.

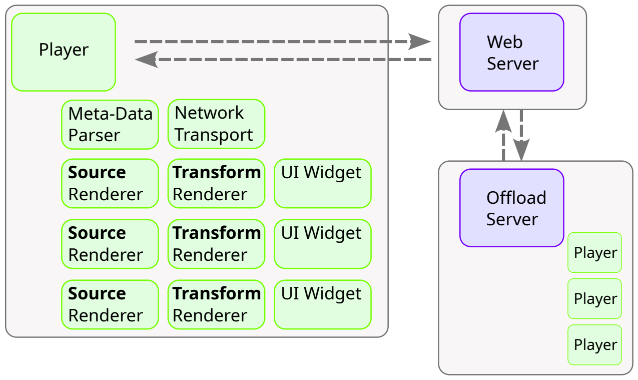
Research on Quality of Experience for OBM has recently identified a range of challenges in how quality is quantified by streaming protocols and subjectively observed by viewers [? ?]. These works note non-obvious subjective appreciation of raising or lowering the quality of different elements of an OBM experience; this work is highly complementary to our own and could be used as part of the foundation of novel QoE models for OBM delivery.

Beyond research into the visual aspects of OBM, Cieciura et al. have explored the authoring of inter-compatible layered audio for highly diverse personalisation [?]. They cover the tools, processes, and production challenges of authoring highly flexibly audio experiences; this is again complementary to our research, which primarily focuses on visual aspects of media delivery.

Gap Analysis and Contributions. Current research trends show an increasing interest in dynamic and re-compose-able media, and in alternative delivery methods such as the use of 3D meshes as backgrounds. As far as we are aware, our work represents the first *generalised* framework to support the **specification** and **delivery**

of layered media in which each media object is uniquely identified as a first-class entity; these media layers are separately delivered to the client device, and composed with other layers in a way that is highly customizable by the viewer. Our work is also the first to examine generalised offloading of layered media composition between a browser-based player and the cloud, using WebAssembly to support the same code being used in both locations. This represents a baseline of performance in dynamic layered media composition between browser and cloud renderers, and serves as a basis for a range of novel future research directions.

3 System Overview



Our object-based media delivery infrastructure is illustrated above. It is designed to stream segmented layered media, where layers can be primary source content (such as videos, animations, or graphical overlays including subtitles), or can be post-processing transformations (such as chroma-keying, various filters, or generative elements). The media of each layer is segmented into arbitrary-length chunks for streaming (such as 2-seconds or 10-seconds), in a similar way to current adaptive bitrate streaming approaches like MPEG DASH [?]. In our current version each segment is delivered via HTTP, but various drop-in replacements to the delivery protocol could be used (such as Media-over-QUIC [?]).

The media player includes a set of Source components, with one Source implementation for each available source content type (such as video or animation). It also includes a set of Transform components, with one Transform implementation for each available post-processing type. This list of Source and Transform implementations is extensible, with their types identified in a per-layer-variant manner in our metadata format.

The entry-point of our server infrastructure is a web server which serves segments of media as requested by a player. This is augmented with an *offload* capability, in which a player can send a HTTP request asking for a set of layers to be rendered by the server into a video segment for a given time-range (with the configuration of the experience specified in the request). When the server receives an offload request, it instantiates a version of the player at an offload site, uses that player instance to render the requested time segment, encodes each resulting frame as video, and responds to the client with the video segment; the video segment may then be cached or pushed into a CDN for re-use. This design allows our server-side infrastructure to operate with existing caching and CDN architectures, which are built around segmented video, while supporting arbitrary offloading of sub-elements of a media experience.

In the following sections we present the detail of our media player and metadata format, and describe how render offload works.

3.1 Player

The core of our design is a media player and its associated metadata format to describe an OBM experience. As summarized above, this same player implementation is used both on the end-user device and in the cloud when offload requests are made. Our media player is implemented in the Dana adaptive systems language [? ?] and compiles either to WebAssembly for web browsers, or to native code to run in the cloud. Our WebAssembly build uses only software-based video decoders, while our native build has access to both software and hardware-accelerated decoders and encoders. WebAssembly is an intermediate binary execution framework, in which native code (such as C) is ahead-of-time-compiled to run within web browsers with 'near-native' performance (though in practice performance varies per-task [? ?]).

The media player begins by downloading a main manifest metadata file which describes the overall media experience (i.e. an episode of a TV show). This file describes layout variants of the experience, the set of layers within each layout, the options available for each layer, and the user interface control types available to configure that variability. The concept of layer options is illustrated in Fig. 1. Each layer option references a second-level metadata file which is ingested by the renderer for that particular layer type. We use JSON [?] as the descriptor format for all metadata files.

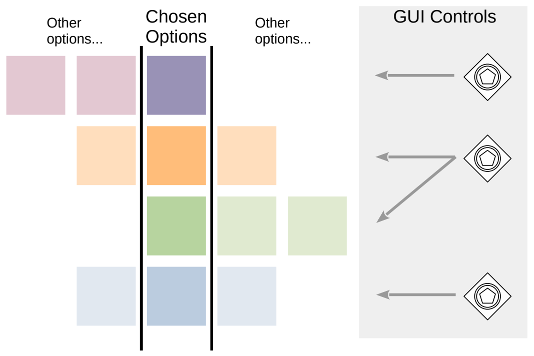


Figure 1: Layering Concept. The layer options between the two black lines are currently-selected. Each layer option can be of a different type, including the special 'none' type which disables that layer. The option for each layer is selected via custom UI controls defined in a show's meta-data manifest.

Once a manifest has been downloaded, along with associated secondary meta-data files, the player parses the meta-data and instantiates all Source and Transform instances for the current (or default) configuration of the media experience. It also instantiates all of the UI control widgets specified in the main manifest and associates their control functions with the appropriate layers. The player then enters a rendering loop, typically running at 25fps.

Each render loop iteration checks if all appropriate Source / Transform instances have sufficient buffered data, and if so calls a `render()` function of each Source / Transform to compose the current frame. The rendered output of each layer is placed on top of the rendered output of the previous layer.

Source instances represent primary content; the track types we currently support are as follows:

Video. Downloads segmented video content, using a fixed segment size, at a given resolution and bitrate, and decodes a frame of video for each frame of the media experience. We use H.264-encoded video [?] throughout all of our examples, where each segment begins with a key-frame.

Animation. Downloads segmented animation data using a simple bespoke animation format. Each segment begins with an animation key-frame which identifies all assets (e.g., PNG files) which are used in the animation. The remainder of the segment has one record per frame, each of which indicate the asset number(s) to render, their positions on the screen, zoom level, and rotation.

Subtitles. Downloads segmented subtitle data using a simple bespoke subtitle format. Segments contain one record per frame which describe the text to render, its colour(s), and relative position.

None. This special track type renders nothing to the screen, and is used to represent a layer effectively being turned off.

Transform types are post-production transformations applied to one or more lower layers. A transform applied to two lower layers, for example, has those two lower layers A and B each rendered to an isolated texture; A and B are then passed into the transform, which outputs a third texture C which is composited with the current frame. The layers that feed into a transform layer may either be source tracks or transform layers themselves. The transform track types we currently support are as follows:

Chromakey. This transform type operates over a single lower layer and applies a colour value to its input from that layer; any pixels which match this value are converted to transparent. This is often used when presenters or actors have their performance captured in front of a green screen to later be replaced with a background.

Transparency. This transform type operates over a single lower layer, and downloads segmented transparency masks; these masks are applied to the input data to convert pixels to transparent ones. This is often used to implement video-with-transparency, where an exact alpha mask is available from the production process.

Hue/saturation. This transform type operates over any number of lower layers and applies a re-colour operation to alter the hue and saturation levels of the input.

None. This transform type applies no modification to its input, acting as a pass-through to render its child layer(s) straight to the screen, and is used to represent a transform layer being turned off.

Finally, UI widgets allow the user to control the available variability in a media experience, and include:

Toggle. This control type cycles between the options for a given layer, and is often used to toggle a layer between the *None* pseudo-track and a content option.

Select. This control type presents a list of options, allowing the user to select one of them.

Select-multi. This control type presents a list of options, allowing the user to select zero-to-n of them for simultaneous presentation.

Layout-select. This control is always shown if the manifest provides more than one layout option, and when clicked allows the user to select which layout they want.

Source, transform, and UI controls each implement a common interface, allowing new types to be easily introduced.

We use JSON to represent all meta-data, as a compromise between human readability and compact representation. The top-level element of the JSON schema is a `layout variants` array; each variant may use a different resolution or screen orientation, and may internally contain a very different set of media layers, in different Z-orderings, or UI control types. The selection of different layout variants is thus the first level of variability for a media experience.

Each layout contains an array of `layers`, listed in their Z-order, such that the first layer is the bottom-most (furthest-away from the viewer's eye) and thus the first to be rendered. The second layer in the array is rendered on top of the first layer, and so on.

Each layer has basic descriptors: a name, the layer type (source or transform), the display type (full-screen or windowed, where the latter includes relative positioning and size), and any conditionality of the layer (such as only-available-if another named layer is turned on). The layer then lists an array of `options` for that layer; each option has a type (such as `video` or `animation`) which references the `Source` or `Transform` implementation that will be instantiated for this layer option. The layer option also features a name and a `spec` which links to a second-level JSON meta-data file.

If a given layer option is selected for inclusion in playback of a media experience, its metadata `spec` file is downloaded, with its `Source` or `Transform` implementation instantiated with the content of the `spec` file. Layer options often include one option of type `None` which equates to that layer being turned off. Finally, a layer specifies an `optionSelect` type, which names a UI widget type which should be used to interact with the variability offered by that layer.

3.2 Render Offload Protocol

One of the main benefits of OBM is that a media consumer can select from a rich set of options in how a given show is presented, without the streaming service having to maintain and store an combinatorial number of fully-rendered show versions: the streaming service has one copy of each source media object, with the media player combining appropriate objects at the point of playback.

The potential drawback of this approach is that the viewer's device must have sufficient compute resource to perform the scene composition / rendering during playback. Depending on the kind of show being consumed, and the set of layer options chosen, some device types may not be able to maintain a suitable framerate, and may therefore require additional compute resource to render their desired experience configuration.

Our player and server infrastructure therefore includes a render offload protocol in which a subset of layers (up to and including all layers) can be rendered remotely and streamed back to the media player as video. In our current implementation the set of offloaded layers must start at the bottom-most layer, to avoid encoding transparency. The media player formulates a HTTP request string which encodes the set of layers being requested, the time-span being requested, and the chosen options for each layer in the set. This encoded URL begins with the identifier `/offload/`, which receives special handling by our content server: where receiving such a request, it contacts an offload server and forwards the encoded URL. The offload server parses the URL, instantiates a media player instance with the layers configured accordingly, and renders each frame to a video. The server-side version of the player has access to hardware decoders and encoders, and is also able to perform highly parallel processing on transform layers such as chroma keys, by decoding and rendering all of the lower layers, then applying the transform to those decoded frames in parallel.

An example offload request URL would be:

```
https://mycontent.com/offload/shows/MyShow/
      variant/0/layers/2/100/160/base/alex
```

This URL requests a show known with the ID `MyShow`, requests the first layout variant of that show, requests the first two layers of that variant, between time indices 100 and 160, with the first layer configured to the option labelled `base` and the second layer configured to the option labelled `alex`.

Using URLs to encode offload requests in this way allows content servers to check if the requested media element is already cached before proceeding to render it. We note that the offload request using this approach contains the same information as the OBM content requests that the player is already sending, so there is no additional data leakage assuming that the offload site is under the same trust model as the media content servers.

An actual policy to determine *when* to offload is beyond the scope of this paper; our current working assumption is that a device's client identification data will be matched against a database to make a static decision, but determining offload status could also be an ongoing negotiation process between the client and server-side.

4 Exemplars

In this section we present three different exemplars of object-based media which we use for evaluation. These are a weather forecast, a Formula 1 race, and a short animated drama. Each of these shows is constructed using different combinations of the layout, layer, and control types we have already introduced.

We use these three examples because they represent three different levels of compute complexity, and three different application-level motivations for the use of OBM. The weather example represents the lowest compute demand, using a single presenter video layered with simple animation elements; its OBM motivation is on accessibility, with spoken and signing presenters, and regular, high-contrast, and low-clutter versions of the presentation. The F1 example is a mid-level compute demand case, rendering multiple time-synchronised videos simultaneously alongside animation layers; its OBM motivation is on viewer engagement, allowing the viewer to track the elements of an F1 race that most interest them. The animated drama is our highest-level compute demand example, rendering five synchronised layered videos, with four of those layers utilising alpha masks; its OBM motivation is on narrative versioning, including plain, horror, and visual clarity versions.

4.1 Weather Forecast

This example is a weather forecast decomposed into individual media objects, shown in Fig. 2 and Fig. 3. The weather map, weather iconography, and presenter all have alternatives to choose from, alongside subtitles being on/off in different languages.

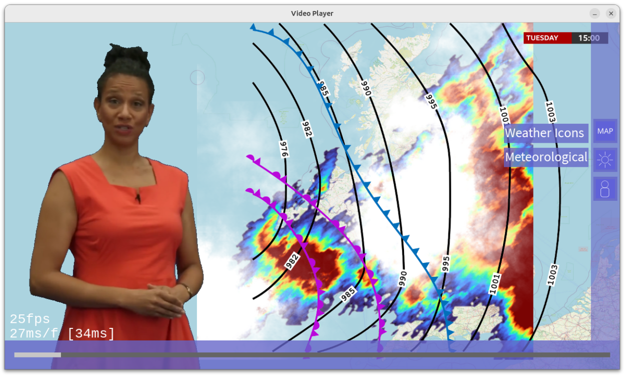


Figure 2: Weather Forecast

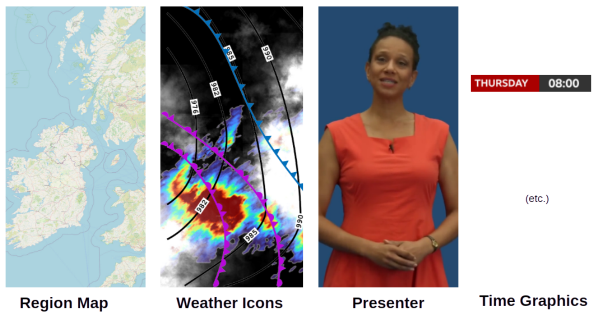


Figure 3: Weather Forecast Layers (map data in the left-most layer is © OpenStreetMap)

The experience has a landscape and portrait layout variant (with the portrait variant intended for use on a device such as a smartphone which is often held with the screen vertical).

Within both layout variants, the map is the bottom-most layer, followed by the weather iconography layer, then the presenter, and finally the time-of-day and subtitles. The map layer is an animation track, where the map is zoomed and panned throughout the presentation. The weather iconography layer is also an animation track, which has corresponding zoom and pan motion, but also changes the weather symbols being displayed through the course of the presentation to match the narrative. The presenter layer is a video with a (non-optional) chroma-key transform layer on top.

For layer-based variability, there are various map options including high-contrast (low-detail) and topographical (high-detail); there are various weather iconography options, including radar / satellite, or simple sun/cloud icons; and for the presenter layer there is a traditional audible/spoken presenter, a sign-language presenter, and an audio-only presenter ('narrator') with no visual element.

4.2 Formula 1

This example is a Formula 1 race, with a wide range of presentation variants, shown in Fig. 4 and Fig. 5. There are three layouts: the producer layout, driver layout, or track layout. The producer layout shows the producer's feed as the full-screen feature, with an optional picture-in-picture in-car driver feed, and an optional track-tracker showing positions of selected drivers on the track.

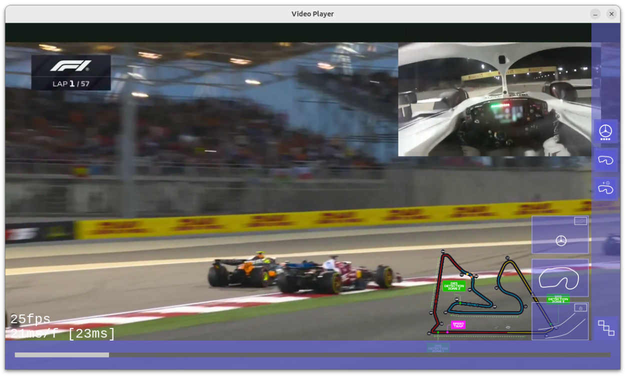


Figure 4: Formula 1

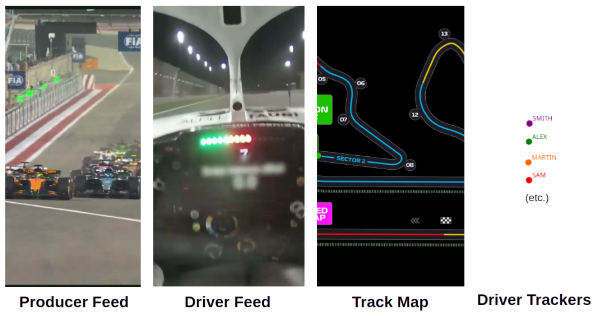


Figure 5: Formula 1 Layers

The driver layout shows an in-car camera feed as the full-screen feature, where the user can choose which car is being followed, with an optional picture-in-picture of the producer feed, and an optional track-tracker with positions of selected drivers on the track.

The track layout shows the track graphic animation as the full-screen feature, with the positions of selected drivers on the track, and has an optional in-car picture-in-picture feed, and an optional picture-in-picture of the producer's feed.

Each layout variant uses a similar combination of track instances, in different layer orders and different sizes. The producer layout, for example, has a video track as the bottom-most layer which is showing the producer's chosen camera angles and cuts of the race. The next layer is a picture-in-picture video layer positioned in the top-right of the screen, in which the viewer can choose to follow an in-car feed of a chosen driver. The next layer is an animation layer showing the track graphic (with transparency), which can be turned on or off, appearing as a picture-in-picture window in the bottom right of the screen. The final layer is set of animation tracks, which can only be used if the track graphic layer is turned on. Each animation track in this layer shows one driver's position on the track, where the user can select [0..n] tracks in this layer to populate the track graphic.

4.3 Animated Feature

This example is a one-minute animated drama, called the 'Changing Forest', which was specifically produced as an OBM experience by professional animators (shown in Fig. 6 and Fig. 7). The animation team used the 3D animation tool Maya to design the original animation, and as a second step generated each layer of the animation as a sequence of PNG frames (since animated elements across layers require precise transparency).

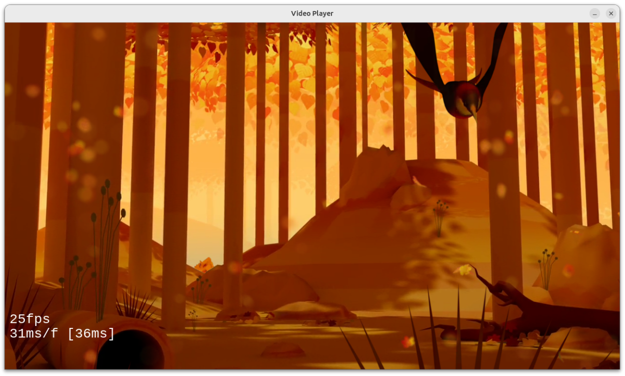


Figure 6: Animated Feature

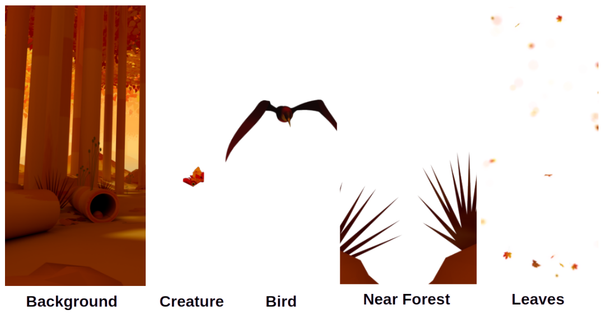


Figure 7: Animated Feature Layers

To realise this as a real-time renderable experience we encoded each layer's frames as a video, with a separate alpha mask. To save on the resulting size of the video layers, our encoding tool calculates the maximum width/height needed by the animated entity, and encodes a video at that width/height, to be shown as a picture-in-picture video anchored at an appropriate X/Y coordinate in the final presentation. This approach allowed us to exactly replicate the animators' production.

In our experience manifest for this example we again have layout variants for landscape and portrait versions, with the cropping bounds of these two variants provided to us by the animation producer. We then have a range of variability in each layer.

The bottom-most layer is the forest background, which is drawn as a full-screen video and has no transparency element. This layer has 'normal', 'horror', and 'high-contrast' versions. The next layer is a mystery creature in the forest, which is drawn as a picture-in-picture video with an alpha mask; this layer has 'normal', 'horror', and 'highlighted' versions. The next layer is a foreground creature element (a bird), which is again drawn as a picture-in-picture video with an alpha mask; this layer again has 'normal', 'horror', and 'highlighted' options. The next layer is a foreground forest, as another picture-in-picture video with low- and high-density options or 'off'. The final layer is a falling leaves element which adds depth to the production; this layer can be turned on or off.

5 Evaluation

We use our media playing infrastructure to answer three empirical research questions on the delivery and rendering of OBM in WebAssembly. Our evaluation is performed using a range of real client devices for media playback, and a cluster of servers in a real datacentre for content delivery and render offload provision. Client devices use WiFi to download content, with the datacentre being in the same local area network (but on a different subnet) as the client devices. Our client devices are:

- A HP EliteBook G2 laptop, with an Intel i7, using Firefox 141.0.3 (our highest-power device).
- A Lenovo ThinkCentre M920q, with an Intel i5, using Firefox 141.0.3 (our lowest-capability device).
- An iPhone 12, using Firefox 142.0.1.

This set of devices offers insight into a range of different hardware capabilities. Our server-side cluster uses different hosts for the web server and the render offload servers. The web server host is a rackmount server with an Intel Xeon Quad Core 3.60 GHz CPU and 16 GB of RAM; the offload hosts are rackmount servers with Intel i7 CPUs, 32GB of RAM, and Intel Iris Xe Graphics (TGL GT2) chipsets. Our prototype software is available for reuse [?].

The remainder of this section is structured to answer three empirical research questions, using our three OBM exemplars, beginning with a comparison between the performance of WebAssembly and native implementations. Overall we seek to measure:

- The framerate sustained during playback at client devices when those devices are performing frame-to-frame compositing, as an indicator of which kinds of experiences require offload support for which classes of client device;
- The render speed of media compositing at offload sites, as an indicator of both initial offload latency and compute load for streaming service providers; and
- The bandwidth required at the client device in both client-rendered or offloaded cloud-rendered versions of the same OBM experience, to demonstrate the tradeoff between locally rendering and offloading an OBM experience: locally rendering consumes more bandwidth at the client, as each layer is streamed separately, while rendering in the cloud incurs computational expense for the streaming service while reducing bandwidth to the client.

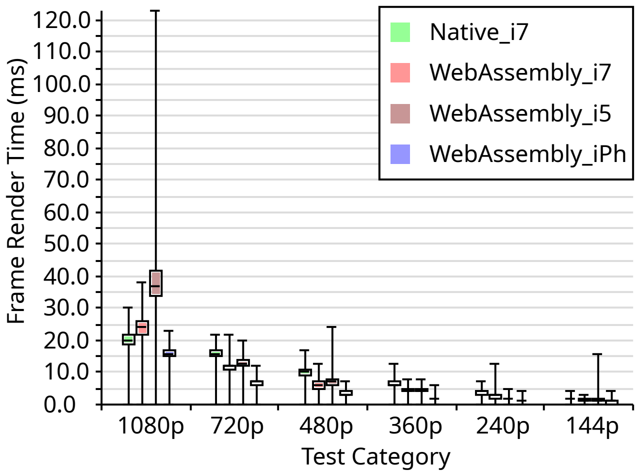


Figure 8: Frame render statistics for a single H.264 video file of a Formula1 race, using a software decoder running natively and in WASM, targeting 25fps.

5.1 RQ1: How does video decode, transform, and render performance compare between WebAssembly and native implementations?

We begin by examining the baseline performance of the WebAssembly (WASM) build of our player against the native build. Both builds use the same source code. We first measure the performance of single video decoding, then the performance of entire frame rendering (involving the compositing of all layers in our exemplars).

Across all of our experiments our target framerate is 25fps, which is a common target in general media presentation. This provides a notional maximum per-frame render time of 40ms to avoid compute-induced jitter during playback. All of our video content is encoded with H.264, using its baseline profile, which tends to have the fastest decode speed (at the cost of some additional file size). We also experimented with AV1-encoded video [?], but found software decode speeds in WASM to be too slow for viability. Note that we use software-based decoding in the WASM build of our player as WASM does not currently feature hardware decoding support.

We measure the decode speed of a single video across a range of common resolutions. Decode speed is measured by reading the system clock, in milliseconds, before and after the render of a frame, and taking the difference between the two readings. The result is stored in a pre-allocated array which has sufficient cells to store all frame timing readings. We carry out these frame timing readings for the first 20 seconds of each video, after which we send the array of readings to our web server using a HTTP POST request.

We measure frame timings using the producer feed of the Formula 1 example. This video has a high range of movement throughout the whole frame and features regular scene cuts as the feed switches between cameras. The results are shown in Fig. 8. At 1080p we see that the iPhone WASM build has the best overall frame render time, though its outliers overlap with the next two fastest platforms. The native i7 build is the next fastest, followed by the WASM i7 build. The WASM i5 build is the worst-performer, with the median lying very close to the 40ms limit and higher outliers than the presenter video. At resolutions of 720p and below we see all platforms keeping render times under 40ms. It is notable that the WASM builds on both the iPhone and i7 are slightly faster than the native i7 build. At lower video resolutions, from 720p downwards, all platforms render the video well within the 40ms limit.

Aside from being of interest in themselves, we note that these results also demonstrate the relative capability of our different platforms to present a *fully-render-offloaded* experience, which would be delivered to the player as a single video.

Our next experiment measures the average frame render times for the entire OBM experience of our three exemplars, in everything-turned-on permutations of those experiences when rendering is entirely performed on the end-user device. The results are shown in Fig. 9, considering only 1080p renditions. In almost all cases here the Intel i7 native build performs the best, able to render all elements of the full experience within the 40ms-per-frame target. The remaining platforms vary in where they lie; for the Formula 1 example the i7 and iPhone WASM builds have a similar median, with worse outliers on the iPhone. In the animated forest feature, the iPhone WASM build demonstrates a better median than the i7 WASM build, but again has worse outliers. The i5 WASM build generally performs the worst, with its median being far outside of the 40ms-per-frame target, though its outliers are slightly lower than those of the iPhone WASM build in the Formula 1 case.

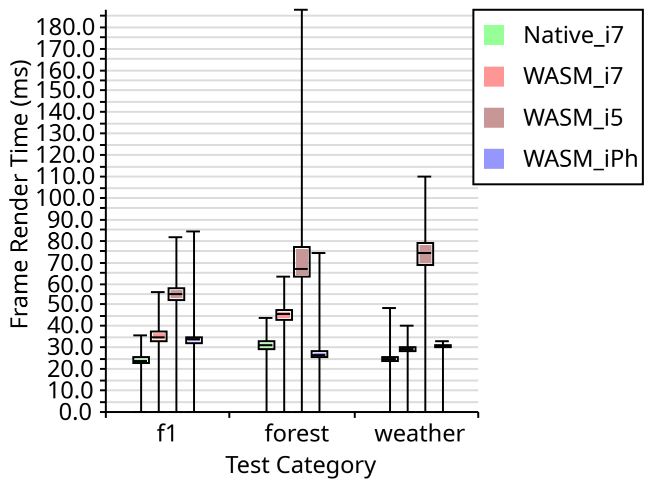


Figure 9: Frame render statistics for the full experiences of the F1, animated forest, and weather forecast, with every layer option turned on in each case.

These results show potential thresholds for offloading the compute of these experiences for different device classes and different experience types, such that the offloaded compute would yield a single video steam for the client device to consume – where the same device types are comfortably able to render a single video stream at the same resolution, as shown earlier in Fig. 8. Our results also correlate well with other measurement studies of WASM vs. native in general compute comparisons (rather than the media-specific comparison here), which found that WASM tends to under-perform native implementations in particular kinds of compute [**?**].

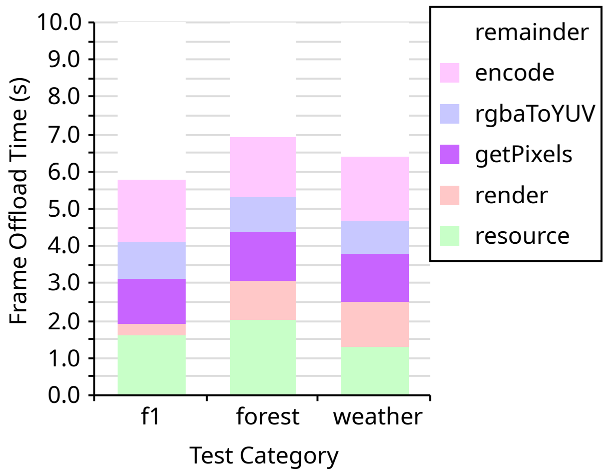


Figure 10: Performance at an offload site building a segment of media at 1080p. The coloured segments of the chart represent different elements of the pipeline needed to generate a media segment, while the white area which fills the remainder of the space up to 10.0 is the amount of remaining real-time for the requested segment length. For F1, for example, it took 5.8 time units to acquire assets for, render, and encode a media segment of length 10.0 time units.

5.2 RQ2: How viable is offloaded compute for on-demand object-based media?

In this section we consider offloaded rendering, for user devices which are unable to render a full object-based media experience. While our system is able to offload arbitrary groups of sub-layers, to reduce the experiment design space here we study only a full-offload scenario. As presented in Sec. 3, offloading involves sending a HTTP request to the content server for the desired layer configuration; this request is forwarded to an offload server which instantiates a media player, downloads the relevant media assets from a content server, renders each frame in the requested time range, and encodes those frames to a single video. Unlike the WASM builds of our media player, the offload server builds of the player have access to hardware-accelerated video decoders and encoders.

The general render process at the offload server media player is to download all assets and create a hardware texture on which to render. For each frame, the media player decodes any video frames and renders them to the texture, along with any animation layers and subtitles, in the Z-order specified by the experience metadata file. The final composited frame is then converted from RGBA to YUV format and pushed to a hardware-accelerated video encoder. We use the Intel VAAPI framework as our hardware decoder/encoder pipeline, with the Intel Iris Xe Graphics (TGL GT2) chipset.

The results are shown in Fig. 10, using a nominal 10 second segment of requested render offload. At a high level we see that the overall rendering process occurs faster than real-time, making the offload approach generally viable for these exemplars. The rendering cost is dominated by two main elements: primary asset acquisition (the bottom segment of the bars, shown in green) and

	OBM size	Composited size	Permutation count	OBM size of *all* permutations	Composited size of *all* permutations
Formula 1	13.9MB	8.3MB	123	106.8MB	1.02GB
Weather	1.7MB	1.2MB	12	3.6MB	14.4MB
Forest	38.4MB	4.0MB	12	38.4MB	48.0MB

Table 1: Storage requirements of 10 seconds of content, for our three different exemplars, when assets are stored as distinct objects or in fully-composited forms.

encoding of the final composited frames (the top-most segment of the bars, shown in pink). The other pipeline phases have lower costs, and include rendering of elements to the hardware texture, transport of texture data from the graphics buffer to the video encode buffer, and the conversion of texture data from RGBA to YUV (a pre-processing step required by the video encoder).

Between the three exemplars, asset download time represents the main point of variation; the forest has the highest asset cost as it involves multiple videos with corresponding alpha masks, with the weather forecast having the lowest asset cost with its single video and smaller animation-based assets.

5.3 RQ3: How does storage and bandwidth compare between pre-baked and OBM?

In this section we examine the storage cost and network delivery costs of object-based and fully-composited versions of the same media experience. This provides a more complete picture of the relative benefits and drawbacks of the object-based approach.

We begin by examining storage costs. The cost of 10 seconds of an individual permutation of each experience is shown in Table. 1, along with the number of permutations, and the resulting storage cost of all of those permutations if stored in their OBM or fully-composited form. We note that these figures are for a single quality level; in practice a content distributor is likely to store a quality ladder (multiple bit rate representations) of each experience.

In the Formula 1 case we see that the OBM version comes in very much cheaper, due to the large number of permutations (this is due to being able to follow a large number of different drivers in various layouts). The weather forecast case is also cheaper, though by a smaller margin. In the animated forest case, the OBM version has slightly cheaper storage cost for all permutations combined; the marginal gain here is due to the inclusion of one video per layer and their associated alpha masks, which take significant space.

We next consider the difference in network bandwidth costs. We measure this using a run-through of the first 500 frames of each exemplar, in which we capture all bytes downloaded by the media player. The results are shown in Fig. 11, measured in bytes-per-frame. In each case, note that the pre-composited bandwidth displayed on these graphs is *also* the bandwidth required by a fully-offloaded version of the experience; an offloaded version is rendered into segments of a single HD video and therefore has an equivalence with traditional segmented DASH delivery.

For the Formula 1 case we see an OBM and pre-composited bandwidth profile which are closely matched. This is likely due to bandwidth in the OBM case being dominated by video content (the main feed and picture-in-picture), with very small assets for the

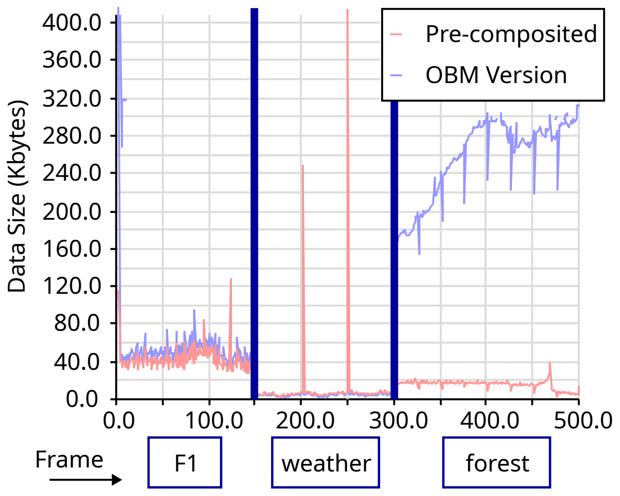


Figure 11: Network bandwidth consumed for the composited vs. OBM versions of each exemplar.

track-tracker animation element. Bandwidth spikes generally occur on video keyframes. The OBM case adds an average of 17.4% extra bandwidth per-frame for this case.

For the weather forecast case we see a more differentiated bandwidth profile. The OBM version here has a low-complexity video to deliver (dominated by blue-screen), with the rest of the media experience delivered using animations. These animations tend to download all assets at the start of the experience and then re-use those same assets throughout playback. The startup bandwidth of the OBM version is therefore higher, with ongoing bandwidth costs then being lower than the composited version. The OBM case adds an average of 38.1% extra bandwidth per-frame for this case.

Finally, for the animated feature case we see a very different bandwidth profile. The OBM version is significantly higher than the composited version, and also shows wide variation over the course of playback. The overall higher bandwidth volume is due to the much larger number of videos (5 separate videos) used to deliver the OBM version, plus the alpha masks that go with those videos. The wide variation during playback is caused by differences in the size of the alpha masks per frame, depending on how much of the lower layers the animated element is obscuring (such that larger obscured areas yield smaller alpha masks). On average the OBM case here adds an average of 94.1% extra bandwidth per-frame.

6 Discussion

Our research to date shows a generalised approach to delivering diverse, user-configurable, layered media experiences; our evaluation demonstrates the overall viability of OBM delivery via WebAssembly-based players in a browser with optional render offloading.

The path to widely deploying such an infrastructure features a range of outstanding challenges, however, which we discuss in this section, grounded in our empirical work from Sec. 5. We group these challenges into the themes of systems infrastructure, quality of experience, and production considerations.

6.1 Systems Infrastructure

Our current system infrastructure is designed for scale-out operations of both media servers and offload sites. However, a range of policy challenges exist which are beyond the scope of this paper.

Perhaps the most significant of these challenges is managing the relationship between content and compute, assuming scenarios where offloaded compute capability is required. In our current implementation, offload sites instantiate a media player and then pull in required assets from content servers before commencing the render; as shown in Sec. 5.2, this asset download is a significant portion of overall offload completion time. A common strategy when presented with this kind of result is to move the compute to the data, rather than the data to the compute (e.g. [? ?]); current content distribution providers are typically distinct from compute providers, however, with bespoke infrastructure tailored to one task or the other. Where these two provisions remain separate, there are two things we could do to improve performance: first, content could be pre-emptively pushed to offload sites using (e.g.) historical modelling; and second, offload sites can both cache assets and prefetch likely-next-assets in a stream, with client devices directed or 'pinned' to offload sites which already hold relevant assets.

On the topic of caching, the use of on-demand offloaded rendering presents a set of novel challenges in how caches work. First, assuming that offloaded rendered segments are themselves cached, it is no longer the case that every cache miss is equal: some rendered segments incur higher compute cost than others. Exploring cache miss cost, as part of cache eviction policies, may therefore be useful in minimising overall compute cost of a large infrastructure. Second, CDN and cache nodes have a novel balance to maintain in their overall cache space: the caching of individual object-based assets (to serve to consumer devices able to perform rendering on their own, or to serve to offload sites), and the caching of composited segments rendered at offload sites. Again, the miss costs of these two data classes are different, but a provider may also wish to skew response times towards consumers able to do their own rendering.

6.2 Quality of Experience

Our empirical work to date is based on a single quality level, with render offload when that level is unsustainable on a playback device. In practice, IP-based media delivery uses quality ladders to adapt to the link quality of each consumer [?]. In layered OBM, the quality of experience (QoE) equation changes in at least two dimensions.

The first is that individual layers can traverse a QoE ladder independently, raising a range of questions in subjective experience for different users. A naive QoE ladder policy may seek to degrade the most expensive layer first (either in bandwidth or compute), but this may deliver a worse subjective result than degrading a group of cheaper layers to achieve a similar resource conservation.

The second dimension is that dropping a QoE level (e.g., by moving to the next lowest render resolution) may change the relative capability of the end-user device to render the media playback, rather than relying on compute offload. In these terms, a QoE decision is no longer about bandwidth alone: consider a scenario in which a device is playing an offloaded version of a show it is unable to render locally, only for a bandwidth negotiation protocol to decide that a lower resolution is needed to stream the offloaded video to the consumer. This resolution drop may then enable the consumer device to render the experience locally, causing a significant change in compute load at the consumer device and the server-side infrastructure due to a notionally bandwidth-oriented decision.

These additional dimensions suggest that new models of QoE may be needed, with subjective testing to help inform runtime policies. Separately to this, an important area of future study is the QoE considerations of the offload-triggering protocol itself. This protocol could be triggered using a simple client device mapping, such that certain classes of device always offload all or part of an experience, or could be determined dynamically in continuous negotiation with the client player.

6.3 Production Considerations

Finally, the advent of render-during-playback for media consumption presents a set of novel trade-offs in resource costs – both for end-users and for streaming providers with a compute offload capability. Render cost, memory usage, and bandwidth characteristics offer a far higher level of flux than is the case in traditional media delivery. This is a question for content providers at an infrastructure level, as discussed above, but is also a question for production teams at the planning stage: given a compute and bandwidth ceiling, what are the best ways to deliver each individual element of an experience, with their variable compute and bandwidth costs? The ability to simulate costs during production is likely to become a key planning process in modelling the total lifetime cost of a project.

7 Conclusion

We have presented an object-based media player complete with a generalised metadata format to describe media experiences and their configurability, alongside a content delivery approach and compute offload infrastructure. As far as we are aware, this is the first empirical study of generalised layered object-based media.

We have presented three exemplar OBM experiences with diverse requirements and have empirically evaluated our work across a range of criteria. Our results show that delivering OBM via WebAssembly is practical, and that this approach is a good candidate to support modular offloading, with common code being used between the browser and the cloud depending on where different aspects of an experience are being rendered.

In future work we aim to introduce subjective testing to further validate our results on sustained framerate and offload latency, further broaden our exemplar set, address some of the challenges discussed in Sec. 6, and examine the ability to use object-based audio for similarly configurable (and offloadable) experience elements in the audio domain. We will also seek to implement media experience types which will allow more direct comparison with other rendering approaches, such as those that use 3D background meshes.

Acknowledgments

This work was partly supported by the UKRI EPSRC BBC Prosperity Partnership AI4ME: Future Personalised Object-Based Media Experiences Delivered at Scale Anywhere, EP/V038087. We thank Prof. Philip Jackson, Dr. Craig Cieciura, and especially the animator Maggie Kosek, for their assistance in preparing digital assets for the animated Changing Forest exemplar.