Video Vectorization

Vectorly is developing a completely new kind of video compression technology called "video vectorization", which can reduce video bitrates by up to 50 times for "vector friendly" content, while improving quality, and without the need for a decoder. This whitepaper summarizes how it works, how to use it, and provides benchmarks and demos.
Outline

Overview

What is Vectorization?
   Raster graphics
   Vector graphics video

What is "vector-friendly" content?
   Hybrid content
   3D content

Why Vectorization?

Why hasn't this been done before?

How it works
   Vectorizeability
   Vectorization
   Vector graphics format
   Packaging
   Playback (SVG + Javascript)

How to use it
   Frontend
   Backend
      Standalone video
      Vector optimization
      Vector transcoding
      Vector streaming

Demos

Benchmarks
   Encoding time
   Bitrate Ladder
   Performance (Frames per second)
   CPU usage

Further Development
Overview

Vectorly is developing a completely new kind of video compression technology called "vectorization", which can reduce video bitrates **by 10 to 50 times** more than HEVC or AV1, while **improving video quality**, all **without the need for a decoder**.

Vectorization is based on computer vision and vector graphics, and works very differently than normal video compression technologies. Some key differences include:

- Bitrates are an order of magnitude lower than H265 for "vector-friendly"* content
- We don't need a decoder because we use established vector-graphics capabilities such as SVG and OpenGL, which are present on 99% of devices.
- One Vector file can play at any resolution (480p, 4K, even 128K) at perfect quality
- Vectors can run in parallel with standard codecs, so you can have H264+Vector

Vectorization is primarily effective for "**vector friendly**" video content, which includes animations, screen-casts, many e-learning videos, gaming content and anything with computer graphics.
Vectorization requires no end-user change, is compatible with existing video players (like JWPlayer, Video.js) and streaming technologies (DASH/HLS) and is based on existing industry-wide standards.

What is Vectorization?

Raster graphics

Videos are composed of sequences of images (frames), which are shown in quick succession to create the illusion of motion. Each frame is composed of a number of pixels, each of which has an assigned color value.

4K video, at 30fps, has ~250 million pixels per second. Without compression, this would be ~750MB per second, or ~93,000 kbps.

Video compression algorithms such as H264, VP9 and AV1 use techniques like discrete cosine transforms to reduce the amount of information needed while preserving image quality.

Vector graphics video

Vector graphics is a radically different approach to visual information. Instead of using pixels, Vector graphics uses collections of mathematical coordinates, connected by lines or curves, to form shapes and polygons.
Vector graphics is an established technology, and is already widely used in graphics design and in animation software. Vector-graphics image codecs such as SVG and Canvas are already supported by 99% of browsers and devices, and vector graphics has already been widely used for video, in the form of Flash animations.

We did not invent vector graphics or even vector-graphics video (e.g. Flash), but Vectorly is unique in converting normal (raster) video to vector graphics video, and in treating this raster to vector conversion as a video compression algorithm (like AV1 and H265), and in making it work within modern video streaming architectures (like DASH / HLS).
What is "vector-friendly" content?

Vectorization is primarily effective for "vector friendly" video content, which is video content that can easily be rendered using vector graphics. Examples include animations, screencasts and screen sharing applications, many e-learning videos, gaming video and video with computer-graphics overlays (suchs as news tickers and sports scoreboards).

A quick rule-of-thumb is: if video content can be broken down into lines, curves, and shapes with solid colors, then it is likely "vector-friendly". A quicker rule of thumb is, if it was recorded entirely with a video camera, it probably isn't vector friendly. Formally, We have developed a numerical measure of "vectorizeability", and will release both a software tool and free API service to determine if video content is "vector friendly".

For non "vector-friendly" content, our technology performs considerably worse than existing algorithms, so we selectively apply our technology only to vector-friendly content.

Hybrid content

Some kinds of video content are partly "vector friendly", as they have computer graphics (vector friendly) in parallel with real-life video (not vector friendly). Examples would be e-sports streams or live screencasts with a picture-in-picture webcam, as well as news broadcasts with news tickers, or sports broadcasts with scoreboards.

For these kinds of content, we seperate the vector & non-vector portions of the video, and vectorize what is vectorizeable. We then combine the vector and non-vector streams together, such that when the user views the video - they see both.

3D content

Our vectorization algorithm is currently focused on 2D vectorizable content. We have experimented with architectures for 3D video content, but have chosen to solve technical hurdles for 2D content first.
Why Vectorization?

The main weakness of Vectorization is that it is only applicable for "vector-friendly" content such as animations, gaming and screencasts. Besides that, there are a number of benefits:

No need for a decoder

Vector graphics can be natively rendered in SVG or OpenGL, which is are existing and established standards which are supported on 98% of web-enabled devices globally. Native platforms such as iOS, Android, Roku and Fire TV also have native vector-graphics rendering APIs. Because these vector-graphics functionalities are hardware-accelerated, playback performance is comparable to native video playback for established algorithms. This enables us to deliver vector-graphics video without the need for a new decoder or video standard, and won't require end users to do or install anything.

Scalability

Vectorized videos don't have a resolution. Because they are based on mathematical equations, they can be played at any resolution (720p, 4K, 8K, or even 128K) with perfect visual quality. Vectorized videos are therefore futureproof, and won't need upscaling. It may even remove the need for adaptive bitrate logic, as one video stream can render to different resolutions on different end-user's devices.

Vectors are an enhancement, not a replacement

Vectors can run in parallel with normal video. Vectorization is not an alternative to other codecs (like VP9 or AV1), it is a complement to them. Unlike with normal codecs where you have to choose just 1 per video, you can use H264+Vector, or AV1+Vector or VP9+Vector.
Think of it like CPUs and GPUs. Most computation is done with CPUs. GPUs are limited, but are much better than CPUs for specific applications. A computer can have both a CPU and a GPU, so you don't need to choose just one.

In the same way - think of vectorization like adding a GPU - you're not giving up anything, but by adding it you do get a performance boost for specific types of video content.

**Why hasn't this been done before?**

This idea is not substantively different from the idea of Flash based animations about 20 years ago. Why has no one done vectorization before?

**SVG, WebGL and HTML5:** Over the last 10 years, browsers and devices have replaced proprietary web technologies like flash with native, open technologies like HTML5, WebGL and SVG. SVG and WebGL are now supported on more web-enabled devices than H.264, and enables us to leverage existing standards & device capabilities to render our vector-graphics videos, in a way that wasn't possible 10 years ago.

**Computer vision:** Our technology relies heavily on computer vision to convert raster-graphics videos to a vector format. Thanks to the rise of the AI industry over the last 5 years, Computer vision has become much easier to do, thanks to both a proliferation of Computer vision software tools and libraries, as well as the rise of GPU powered AI cloud computing.
How it works

Vectorizeability

The first step in our algorithm is to determine "Vectorizeability", a quantitative measure of how easy content is to display using vector graphics.

A vectorizeability score is calculated for each frame, at around 1000 fps. Frames with a high enough score are deemed to be vectorizable.

- If no frames are vectorizeable, we return the original content unmodified.
- If all frames are vectorizeable, we vectorize the entire video
- If some frames are vectorizeable, we vectorize only what is vectorizeable. We then stitch together vectorized scenes with non-vectorized scenes, preserving the original encoding for non-vector scenes where possible.

Vectorization

For videos that are vectorizable, we then use computer vision to analyze each frame of the video, identifying objects and their movement over time.
Our algorithm then finds a vector-graphics representation for each object's shape, and its changes or movements over time.

**Vector graphics format**

We then compile this vector data into a custom format, which is based off of SVG, with modifications to enable key animations concepts from the Flash format, such as:

- A timeline - To organize events within the animation
- Key-frames: Points in time where the full outlines of objects are specified
- Layers - Hierarchies of objects within the view, to enable foreground objects to cover background objects
- Transitions: Specifying movement and changes in shape over time

At this point, our vector file is a structured-text file, and has no pixel or frame information whatsoever.

**Packaging**

We then "chunk" the vector files into discrete time segments, and embed them within an MPEG container.
The resulting MP4 file can then be opened directly by the player (in the next step) or be packaged for streaming in DASH / HLS / CMAF architectures.

**Playback (WebGL + Javascript)**

Vectorized mp4s use our custom vector-codec, and video players will ignore them if our library isn't present.

If our library is present, it will detect the vector-video and intercept the vector data. This data is then rendered in SVG or WebGL in real-time, and inserted into the website or app, and rendered on top of the native video player. This is not fundamentally different from how modern video players like JWPlayer or VideoJS intercept video-data and overlay graphics and interfaces on top of native video interfaces.

The result is video playback which looks or feels like any other video, and operates within native video interfaces without any external codecs or plugins.
How to use it

Using vectorization involves two steps. (1) Vector encoding on the backend, and (2) Enabling playback on the frontend.

Frontend

To enable playback of vectorized videos on end user's devices, you will need to include a library (a javascript file for Web, or a mobile library for the appropriate platform).

```html
<head>
  <script src="video.js">
  <script src="vectorly.js">
</head>
<body>
  <video class="video-js">
    <source src="vectorized-video.mp4">
  </video>
</body>
```

That's it. You really just need that one line of code to enable playback on the frontend, and vectorized videos will play. You won't need to change your existing player, or change any other code or configurations.

This is because the library isn't a player itself, but rather operates at a lower level to intercept vector-graphics data and render it within the native video interface. The library does expose a low-level API for specific configurations of vector-playback.

Vectorly is building its own standalone player, built off of Shaka player.

Backend

Broadly, vector encoding takes in a video-file and outputs a vectorized video, but there are multiple options for encoding vector video, depending on your current video architecture.
Standalone video

For a single unpackaged video, vectorization just outputs a vectorized mp4, which can be played directly on the end-user's device through our library.

Vector optimization

In video streaming architectures, videos are transcoded to multiple presets and packaged for streaming via DASH, HLS or CMAF.

Vector optimization creates a vector-preset, and updates the manifest with the new preset.
By default, presets with unrecognized codecs are ignored, so end-user devices will fall back to an existing preset and end-user devices won't see any change.

Devices which do have our library will detect the vector-preset, and choose to play it back as long as it doesn't violate the specified ABR rules.

Vector transcoding

For new video assets, you can also run the transcoding and vectorization steps in parallel.

As with vector optimization, end-user devices without our library will ignore the vector track, and those with the library will choose to play back the vector track (subject to ABR rules).

Vector streaming

For client-to-client applications such as screen sharing or video conferencing, there is another architecture that we are developing:
In cases where software is present on the client's device, it is possible to send vector data directly to the cloud for encoding and broadcast to end users via vectors or video streams, enabling better end-user quality and lower latencies than H.264 streams.
Demos

Simpsons

Our first vectorized proof of concept / demo is a 20 second clip of the Simpsons located here @ 250kbps. The source video is a 1440p H264 encoded video located here.

Khan Academy

Our technology also works very well for e-learning, and especially Khan Academy-style videos. Here is a 20kbps clip, with the original located here.
Benchmarks

Here we profile our basic (version 0.0.2) vectorization algorithm for the "Simpsons 1440p" video clip provided in the demo section, and compare it existing codecs on

- Quality / File Size
- Encoding Time
- Playback rate (FPS)
- CPU Usage

The raw video for encoding comparison (3.3 GB) is located [here](#).

Encoding time

The following tests were run via ffmpeg version N-96835-g177c68e, built with libaom, libx265, libvpx and libx264 on Ubuntu 16.04 running on a Google Cloud using a n1-highcpu-8 instances.

Running the source raw video using a 1-pass CRF-28 output, resulting in the following runtimes:

<table>
<thead>
<tr>
<th>Vector</th>
<th>H.264</th>
<th>AV1</th>
<th>VP9</th>
<th>H.265</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>11</td>
<td>7600</td>
<td>85</td>
<td>145</td>
</tr>
</tbody>
</table>

Our vectorization code is still at a very early and unoptimized stage, and runs entirely in Python. We are confident that the run time of Vectorization by 1-2 orders of magnitude by re-writing computationally expensive parts of the algorithm, implementing the code in C++ and adding GPU acceleration.
Bitrate Ladder

Our basic vectorization algorithm doesn't have bitrate control, but by varying certain parameters, we can vary the output bitrate and quality.

Vectorization, even at a very rough early stage implementation, performs better than raster based codecs at very low bitrates. The measurements (from ffmpeg's PSNR filter) don't seem to make sense for AV1 or VP9, so it is perhaps better to visualize this difference by looking at the visual output for different codecs at 150kbps:

- **H264**
- **H265**
- **VP9**
- **Vector**

Libaom's AV1 encoder does not encode below 250 kbps for this clip (presumably for quality control), but AV1’s output at the 250kbps bitrate floor is very good.
**Bitrates & P-frames**

While there isn’t a 1-1 correspondence with raster-based codecs and vector-graphics, our algorithm currently uses the equivalent of i-frames for every frame, providing a complete standalone description of every object in any given frame. This is obviously very inefficient.

We have developed a technique for using differences in vector outlines, as well as key-frames, in order to predict vector outlines in in-between frames (the equivalent of p-frames and b-frames in raster based codecs). Initial experiments indicate a 5x bitrate reduction compared to full i-frames.

Implementing “vector p-frames” would let us push the average bitrate of Simpsons-like animations to 40-80kbps and potentially even lower, however this technique requires at least several more weeks of work to make robust and reach production level, and we have placed this on our development roadmap for mid 2020.

**Quality & Posterization**

The biggest outstanding quality issue in our algorithm is posterization. For example, consider the following frame with a gradient for the sky.

Vectorization currently breaks the gradients into bands of solid color, a known, common outcome of vectorization, but a dealbreaker for commercial video applications.
Fortunately, this problem is very solvable, through detection of gradients in the encoder, and rendering of gradients in our WebGL / OpenGL based decoder. Implementing this solution is the #1 priority on our roadmap as of mid May 2020, and we expect to have this issue solved in mid-2020.

**Performance (Frames per second)**

We profiled the performance (frames per second) for the 2Mbps 1440p output of each algorithm (and the single Vector version), using Google Chrome's rendering performance tool, with all tests done on a single window of Google Chrome v80.0.3987 build for Ubuntu 16.04, running on an 8-core Intel Core i7-8550U CPU @ 1.80GHz. These tests were done comparing our javascript-based video player to Chrome's native "video" tag, but have also included playback done via "VideoJS" for an apples-to-apples real-world comparison.

<table>
<thead>
<tr>
<th>Vector</th>
<th>H.264</th>
<th>AV1</th>
<th>VP9</th>
<th>H.265</th>
<th>H.264 (VideoJS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

Running the same clip on a Samsung A2 entry-level smartphone, using a 1.6GHz Samsung Exynos processor, on Android 8.1 and Chrome 70.0

<table>
<thead>
<tr>
<th>Vector</th>
<th>H.264</th>
<th>AV1</th>
<th>VP9</th>
<th>H.265</th>
<th>H.264 (VideoJS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30</td>
<td>N/A</td>
<td>30</td>
<td>N/A</td>
<td>30</td>
</tr>
</tbody>
</table>

The consistency of 60fps and 30ps stems from differing browser repaint rates on desktop and mobile web platforms. In either case, vector playback renders at the device refresh rate at 1440p, and is consistent with playback performance for other codecs.

**CPU usage**

Using the same build, below is the CPU usage for running the same video compared to native video playback, as well as playback via VideoJS.
Vector rendering is fully hardware accelerated via WebGL and OpenGL, so the CPU is not being used to actually render the video. Instead, Vector playback of our current demos has ~5% CPU usage because of the overhead from the Javascript player logic. In real world settings, established Javascript players use much more CPU time.

Vector playback requires some javascript logic to function in web runtimes, which adds some performance overhead, and therefore is a potential weakness of the technology.

In practice however, DRM, streaming, analytics and ad placement also require javascript logic to function in web runtimes, so in real world settings web-video playback can and does use a non-trivial amount of CPU time.

Either way, these numbers reflect the performance of our demos as of May 2020. We believe performance can be improved further, via WebAssembly, Webworkers and parallelization, to reach less than 1% CPU usage for the core player logic in web runtimes.
Further Development

Our vectorization algorithm is currently at a very early stage, and we have shown that vectorization can function as a viable video compression technology, which can be used in conjunction with alternatives such as AV1 and H265.

Further development in 2020 will be focused on

- Fixing the posterization and other pending quality issues
- Further reducing bitrates to below 100kbps for 4K video content
- Improving web performance and reducing CPU usage
- Applying the algorithm to a wide variety of content
- Optimizing the encoder runtime, through GPU, multithreading and C++