Project: Direct Manipulation Video Navigation for the World Wide Web

Michael Barger

September 24, 2017

TITAN
TITAN
achieve the strength
of the Titans
TITAN

Chapter 1

Scope

1.1 Goals

The objective of this project was to implement a novel method for interactive video navigation, ***3D Direct Manipulation Video Navigation*** (hereforth referred to as *3D DMVN*) as a service remotely accessible via the World Wide Web. The original implementation of 3D DMVN was a native PC executable; porting it to a web-based system requires different considerations in its systems architecture; for example:

- a web service is usually accessed through a remote client, and the data transferred from server to client must traverse the World Wide Web;
  - server operation cost and expectation of limited client internet bandwidth encourage careful consideration of network usage;
  - as a consequence, both overall data throughput and transfer latency must be taken into account;
- the hardware capabilities of the compute device (PC, phone, tablet, etc) that accesses the service is uncontrollable by the developer;
- the client of the service must run in another program, the Web Browser, and operate within its own limitations and security constraints; and
- preferably, for the purposes of encouraging widespread adoption as a practical solution, the solution should be built upon a standard Web Server, with no additional, custom back-end services.

1.1.1 Direct Manipulation Video Navigation

Direct Manipulation Video Navigation (or *DMVN*) is a proposed method by which a user is enabled to temporally navigate through a video by dragging an *Object of Interest* along a visual representation of its path of motion, instead of through the traditional linear timeline interface.

As one example of a DMVN implementation: a user might be able to select an object, be presented with a visualization of its motion trajectory path, and then use the mouse to drag the object through that path of motion, rather than through a timeline.

In other words, DMVN allows the user to interact with a video's seek-time spatially, rather than temporally.

1.1.2 *3D* Direction Manipulation Video Navigation

3D Direct Manipulation Video Navigation (*3D DMVN*) is a proposed extension of DMVN that allows the user to "rotate" the video and the object path, whose z coordinate corresponds to a unique position in time. This resolves many problems with the original DMVN:

1.1.3 Temporal Ambiguity

If the object pauses in place for some period of time, then the user will have no way to select any specific point of time in that interval with DMVN, since the path remains a constant (x, y) throughout that entire period of immotion. 3D DMVN resolves this, since *time* is given its own axis on the path (x, y, t), which can easily be seen and selected by the user by rotating the view.

Recurring Motion

If an object's motion trajectory demonstates recurring motion (for example, the pendulum of a clock), the simpler DMVN's motion path would be self-occlude, making the selection of a specific point in time impossible; when a user clicks, should it select the first time the motion recurred? Second? 3D DMVN resolves this again through the temporal axis, which makes redundant points impossible.

Self-intersecting Motion

If the motion trajectory ever crosses itself and a user is dragging along the path, traditional DMVN doesn't provide enough information to resolve which path to continue along to provide a temporally continuity of navigation. Again, 3D DMVN resolves this by ensuring that every point on the path has a unique coordinate, and thereby avoids confusion during the navigation process.

1.2 Limitations

This project is *not* meant to create a production-ready, polished product, but rather to serve as a *proof of concept* of a web implementation of 3D DMVN, and hopefully someday be of use as a foundation for demonstrating that DMVN can be practically useful in everyday applications.

- Does not implement the optional *Perspective Correction* described in the 3D DMVN paper, as it would not have been well suited for the example videos, and seemed outside the scope of effort.
- Leaves user-interface development for *mobile devices* for future development, as again this would be extra effort on the client beyond the scope of the client-server architecture as a whole.

1.3 Materials

The author of the 3D DMVN paper graciously provided materials from his research for my implementation. These included:

- Two example videos in AVI container format, well-suited to illustrate 3D DMVN:
  - ***Gym***: a scene where a man uses a weight machine. The scene is largely immobile, except for the man's arms and the weight pulley system. Both of these objects under motion repeat their paths in a self-occluding fashion (when viewed in 2D DMVN), and thus provides an example of the utility of 3D DMVN.
  - ***Painting***: a close-up scene where an artist paints with a paintbrush. This scene offers some challenges due to the complicated path of the brush, and the close-up field of view causes parts of the painter's hand and brush to leave the video for periods of time.
- Uncompressed binary data files relating to the above two videos.

1.4 Tools

I wanted to use the most basic toolset possible, eschewing third-party libraries (as much as practical) in order to keep the majority of the implementation manual, for educational purposes.

- `vim` for the authoring of this report and all project code;
- `python` language and interpreter,
  - along with the popular `numpy` and `opencv` libraries for the purposes of performance;
  - `python` was chosen because I was unfamiliar with the language (making for a good learning experience), and because it seemed like it would make a good compromise between coding efficiency and runtime performance (when utilizing C-based libraries such as above)
- `html`&`css` markup languages for the description and styling of the client-side web page;

- `javascript` for front-end development, as it remains the only native client-side language for the web, using only
  - the `underscore` functional library for the `_.flatten` function, which reinterprets multidimensional arrays as single-dimensional
  - the matrix and vector function library `mat.js` was written manually by myself for educational purposes;
  - it should be noted that I purposefully avoided higher-level 3D libraries, such as `three.js`, since I felt that it would take away from the effort of the project and might compromise the results;
- Google **Chrome** browser for viewing and debugging of the web front-end; and
- **LaTeX** for formatting of this report, since I had never used it before and figured now would be as good a time as any to learn.

Chapter 2

Implementation

2.1 Systems Architecture

2.1.1 Considerations

In the analysis of the systems architecture for this implementation, we must give prime consideration to which aspects of a WWW 3D DMVN differ from those of a native PC implementation.

Network Bandwidth

Network bandwidth is an important consideration in the design of a web-based system that is generally not a factor for native programs that operate locally on a single PC. Lack of consideration for this will cause the client to experience (potentially very long) long waits while information is transmitted to their PC over the Internet, and will cost the proprietors of the server money, as they will need to increase their upload bandwidth service in order to avoid service congestion.

It is thus paramount to minimize the amount of data that must be transmitted to the PC to be the absolute minimum necessary.

Network Latency

Similarly, client-server transactions will always take some amount of time, and so client dependency on a large number of small transactions should also be avoided, due to the detrimental effects on interactivity that network latency will cause.

For example, we should probably avoid a DMVN system where the nearest path vertex to a mouse click is detected by a request from the client to the server, where the answer is calculated there and returned to the client over the internet. This sort of architecture would cause an unresponsive user experience due to the delayed interactivity.

Client Performance

Since web clients can run on all sorts of computers, with various browsers and hardware capabilities, we should be cautious about making assumptions about the performance capabilities of the client. This again is different from writing a native application, where it may be possible to control which computer hardware it is executed on.

In general, to provide a positive, low-latency user experience, we want to keep client run-time performance a prime consideration, and be mindful of the potential impact of intense calculations and graphics on such.

Server (Back-End) Burden

Back-end server burden can take the form of:

- Data Storage
- Custom HTTP Services
- Computation (CPU Cycles)
- Bandwidth (as mentioned above)

All of the above incur monetary cost to the server host, and so should all be considered when designing a Web system.

2.1.2 Architecture

The client-facing front-end web page will provide all input/output user interactivity: it will show the 3D DMVN video and path, it will provide the means for the user to rotate the DMVN video and navigate through the motion trajectory path. It will load its requisite front-end files: HTML, CSS, JS (JavaScript), and video (H.264/MP4) are statically requested from the server by the web page, as is standard and expected by any HTML page on the World Wide Web.

However, since path data for every pixel is too large to make a static resource of the web page, we will instead make dynamic HTTP GET requests for that data, and only send the necessary information across the Net. In order to achieve this via standard HTTP services (and not require a custom back-end server), we will make a *separate path data file* for each pixel in each video.

On the Web front-end, we will provide interactivity through JavaScript (by default, since it is the only language supported natively), and use GPU acceleration through the WebGL API for the graphics processing required for 3D DMVN.

2.2 Back-End Preparation

Before we can implement the 3D DMVN front-end service, we need to first generate path data files from the provided video data.

2.2.1 Reverse Engineering

Since there was no description of the binary files associated with the videos provided, I needed to perform some reverse engineering in order to utilize them. I made the following observations:

- The filenames were of the format `uvXXX.bin`, where `XXX` represents some integral value:
  - `uv` appeared to refer to (u, v) relative coordinates in common computer graphics nomenclature;
  - The range of numbers, always starting with `1`, seemed to correlate with the number of frames in the associated video file;
  - The files' extension was `.bin`, so the contents were likely binary.
- All files had the exact same file size, reinforcing that these must be uncompressed binary files, and that the format of the contents must be related to the dimensions of the video in some way;
- Since both videos were 640×360, dividing their filesize by that product resulted in a per-pixel allotment of 16 Bytes. Divided in half (for the u, v coordinates), this would allow for 8 Bytes per unit of information. If this were optical flow information, as I expected it might be, then it would seem that the datatype must be 64-bit floating point.
- Inspecting the binary contents of the files through the `xxd` utility confirmed that there were repeating, regular patterns on 8-Byte boundaries.

Next, we attempt to read the files and display them on the screen: one common way to convert the (u, v) (the optical flow vector of $\vec{\Delta p} = \langle \Delta x, \Delta y \rangle$) information to displayable colors is by converting the information to a representation in HSL space as a kind of polar visualization:

$$\begin{aligned} h &= \theta \\ s &= 1 \\ l &= \|\vec{\Delta p}\| \end{aligned}$$

where θ is the angle between $\vec{\Delta p}$ and the vertical unit vector $\langle 0, 1 \rangle$. We can find the angle between those vectors by solving the definition of the vector dot product for θ:

$$\begin{aligned} \vec{a} \cdot \vec{b} &= \|\vec{a}\|\|\vec{b}\| \cos\theta \\ \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|} &= \cos\theta \\ \theta &= \cos^{-1} \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|} \end{aligned}$$

So, if $\vec{a} = \vec{\Delta p}$ and $\vec{b} = \langle 0, 1 \rangle$, then this simplifies to:

$$\begin{aligned}\theta &= \cos^{-1} \frac{\vec{\Delta p} \cdot \langle 0, 1 \rangle}{\|\vec{\Delta p}\| \|\langle 0, 1 \rangle\|} \\ \theta &= \cos^{-1} \frac{\Delta p_y}{\|\vec{\Delta p}\|}\end{aligned}$$

and thus our conversion from optical flow vector to HSL display pixel becomes:

$$\begin{aligned}h &= \cos^{-1} \frac{\Delta p_y}{\|\vec{\Delta p}\|} \\ s &= 1 \\ l &= \|\vec{\Delta p}\|\end{aligned}$$

However, the resulting visualization appeared to be spatially distorted:

After some thought, it occurred to me that perhaps these optical flow files were created through the MatLab SIFTFlow package; if that were the case, then perhaps the binary representation of the multidimensional array might be in Fortran order, rather than C order. This was the key, and the result was as below:

2.2.2 Optical Flow

Optical Flow is a means of determining the motion of pixels from one frame of video to another. The binary files associated with the provided videos are optical flow data, providing a $\Delta\vec{p} = (\Delta x, \Delta y)$ for each pixel, for each frame of video. This $\Delta\vec{p}$ expresses how the optical flow algorithm determined the pixel from the first frame to the next.

We intend on using this optical flow information to trace the path of motion from a starting pixel through each frame. We can define this algorithm recursively:

$$\vec{p_n} = \vec{p}_{n-1} + \vec{\Delta p}_n(\vec{p}_{n-1}) \tag{2.1}$$

where the initial pixel position $\vec{p_0}$ is given, and $\vec{\Delta p}_n(\vec{p})$ is the optical flow function for frame n at position $\vec{p}$.

Bilinear Interpolation for Subpixel Optical Flow Parameter

While it is always the case that our initial position is $\vec{p_0} \in \mathbb{N}$, $\vec{f_n}(\vec{p_{n-1}}) \in \mathbb{R}$. In other words, the optical flow field is only defined for pixels (with integral coordinates) and not subpixels (with real coordinates), so in order to find $\vec{f_n}(\vec{p} \in \mathbb{R})$, we must perform interpolation between several $\vec{f_n}(\vec{p} \in \mathbb{N})$.

The form of interpolation we choose is Bilinear Interpolation, because it is computationally simple while providing sufficient quality of results. In order to do this at $\vec{p} \in \mathbb{R}$, we will first fetch the four surrounding defined optical flow points $\vec{g}_n$:

$$\begin{cases} \vec{g}_1 &= \vec{f}(\langle \lfloor p_x \rfloor, \lfloor p_y \rfloor \rangle) \\ \vec{g}_2 &= \vec{f}(\langle \lfloor p_x \rfloor, \lceil p_y \rceil \rangle) \\ \vec{g}_3 &= \vec{f}(\langle \lceil p_x \rceil, \lfloor p_y \rfloor \rangle) \\ \vec{g}_4 &= \vec{f}(\langle \lceil p_x \rceil, \lceil p_y \rceil \rangle) \end{cases} \tag{2.2}$$

and then perform Linear Interpolation as follows:

$$\begin{cases} \vec{a} &= \text{lerp}(\vec{g}_1, \vec{g}_3, \{p_x\}) \\ \vec{b} &= \text{lerp}(\vec{g}_2, \vec{g}_4, \{p_x\}) \\ \vec{f}_{interp} &= \text{lerp}(\vec{a}, \vec{b}, \{p_y\}) \end{cases} \tag{2.3}$$

where $\{x\}$ is the fractional portion of x, or $\{x\} = x - \lfloor x \rfloor$, and Linear Interpolation (lerp) is defined as:

$$\text{lerp}(a, b, x) = ax + b(1 - x) \tag{2.4}$$

and $0 \leq x \leq 1$.

2.2.3 Path Generation

Now, using some programs written in `python` with the `numpy` and `opencv` libraries, we generated path information for the web page to use. Ultimately, we intend to export the generated path as JSON information; in order to do so, we will convert the set of path coordinates from Video (Pixel) Space to Normalized Device Coordinates, with the z coordinate set to be on $[0, 1]$, where 0 is the first frame, and 1 is the final frame.

Interactive Path Generation

The first program, written for development purposes, would load the first frame of the video and allow a user to select a single pixel with a mouse click. Then, it would trace the optical flow from the pixel, showing its path while playing back the video. Finally, the program provided JSON for the selected pixel's path; I used this for initial development and testing.

Batch Path Generation

The second program does the same as above, eschewing the visualizations and UI for performance, tracing the optical flow path for every pixel in the video, rather than a single pixel selected by a user. After some optimization by minimizing looped calculations and preloading all optical-flow files into memory, I was able to get execution time for the batch path generation down to about 12 seconds per video line for the `gym1` video, and about 15 seconds per line for the `painting1` video, which . However, it would still take more than an hour to generate the paths for every pixel in these relatively short and low-resolution videos. The final result of running this program was a directory full of JSON path files, with one file per pixel and a strict naming format: for example, `painting1-0235x0192.json`.

2.3 World Wide Web

2.3.1 Relevant Technologies

Our solution will be formed by a combined system of several discrete technologies.

HTTP

HyperText Transfer Protocol is an Application Layer (OSI) network protocol that serves as a foundation for the World Wide Web. Through it, a client (Web Browser) can request ("GET"), send ("POST"), or otherwise transceive data to or from the Web Server. HTTP is the core of our client-server system.

HTML

HyperText Markup Language describes the content of a web page. This was the original World Wide Web technology, and it continues to form the backbone today, allowing the author to list and name which DOM elements are present, and in what hierarchy.

CSS

Through *Cascading StyleSheets*, the styling of HTML elements can be managed; CSS was introduced to separate the concerns of content (HTML) and presentation (CSS), so that each could be modified independently of the other.

DOM

DOM (Document Object Model) is the API by which JavaScript can access and interact with the client's web page, the front-end for our application. Through the DOM, JavaScript can add event handlers; add, modify, or remove elements from the HTML tree structure; and change the styling and behavior of the webpage as a whole.

JSON

JSON (JavaScript Object Notation) is a format for structured data storage in an ASCII (as opposed to binary) representation. Its use is particularly popular in conjunction with JavaScript (and by association, the World Wide Web as a whole), since it is able to serialize and deserialize native JavaScript objects. Its primary weakness is verbosity (file size), especially when compared with a binary format. It is, however, less verbose than XML, which it has come to replace in most recent projects.

In our project, JSON is the format in which we store path data on the server; the client can request the JSON from the server via HTTP requests, and deserialize the data directly into a JavaScript objects for programmatic use.

JavaScript

JavaScript, also known by its proper name *ECMAScript*, is the sole native scripting language of the World Wide Web. When adding programmatic capabilities to a website, JavaScript is the only allowed language.

WebGL

WebGL is the API by which a client webpage can utilize the computer's GPU for graphics coprocessing. A standard from the Khronos Institute, it is derived from OpenGL ES 2.0, which is itself a subset of OpenGL intended for simpler mobile devices.

Interestingly, despite its implementation in JavaScript WebGL's API remains nearly identical to OpenGL's relatively archaic and obtuse C API: no high-level constructs or programming methodologies (such as Object Oriented or Functional Programming) are employed.

2.4 Hardware-Accelerated Graphics

Graphics have always posed a special problem for computers, due to the large amounts of data that must be processed in real-time. As early as the 1970s, special video accelerators and coprocessors existed to help make this real-time graphics processing possible without overloading the Central Processing Unit. Even with today's multi-core CPUs, running at several-Gigahertz clock speeds, special coprocessors are still necessary for real-time 3D graphics.

2.4.1 Graphics Processing Units

The Graphics Processing Unit hardware, as we know it today, was initially developed during the 1990s to provide both 2D and 3D graphics acceleration. While originally a discrete add-on card to a PC, it is now ubiquitous, built into nearly all popular CPU silicon dies, including those of mobile devices.

GPUs are SIMD (Single Instruction Multiple Data) coprocessors designed for massively parallel computation, with accelerated and specialized memory caching for 2D locality.

2.4.2 GPU APIs

GPUs must be controlled through an API, which acts as an interface between the CPU and the GPU's device driver. While there are several such APIs, the most widespread by far is known as *OpenGL*. OpenGL was developed in the 1990s, and today remains an archaic and challenging stateful C API.

WebGL is an implementation of OpenGL meant to enable access to GPU hardware in web browsers. It is based off of the OpenGL ES 2.0 version, which is a more recent, somewhat limited version of OpenGL for mobile devices. Ironically, even though WebGL is a JavaScript API (since JavaScript is the only language available natively in a web browser), its API is almost identical to the archaic C API. Most web developers who do not have knowledge in computer graphics might add 3D graphics to their websites by using a wrapper library around OpenGL, such as the popular `three.js`, but for the purposes of our project, we will stay as close to "bare metal" as possible for educational purposes, and use WebGL directly.

2.4.3 OpenGL Architecture

While a full description of OpenGL architecture is not within the scope of this report, a cursory review may be warranted. When using WebGL (which is, for all intents and purposes, the same as OpenGL), we must perform substantial setup, or write "boilerplate" code to create the graphics environment, which consists of the following objects:

- *Programs*, which are made of compiled and linked *Shaders*, written in the GLSL language;
- *Buffers*, which are arrays of data to be considered Vertex Attributes (attributes correlated to a vertex), such as position or texture coordinates;
- *Textures*, which are (generally) 2D image bitmaps.

When a *draw* function call is made on a Buffer, then the following basic process takes place:

1. The *Vertex Shader* executes *for each vertex being drawn*; the vertex shader takes as input the following:
    - *Attributes*, which are information that is correlated with the vertex and had been stored in a Buffer;
    - *Uniforms*, which are *not* per-vertex information, but global to all vertices.

   and outputs the following:

    - *Varyings*, which are information correlated to the current vertex that will be passed on to the Fragment (Pixel) Shader;
    - *Position*, in Normalized Device Coordinates, which will be used in the Clipping stage.

2. In the *Clipping* Stage, any drawn triangles where all vertices have at least one of their x, y, or z coordinates of their Position out of the bounds [-1,1] will be clipped and will not progress any further;

3. *Rasterization*, where unclipped polygons will be rasterized to the viewport based off of their vertices' Positions;

4. The *Fragment Shader*, also known as a Pixel Shader, which will be run independently for each of the pixels that were rasterized for the polygon. It takes as inputs the following:

   - Varyings, which are bilinearly interpolated from the values set by the vertices based off of the spatial position of the fragment (pixel) relative to those vertices;
   - Uniforms, which again are global information shared among all fragments.

   and as outputs the following:

   - Fragment Color, which is the RGBA value that the pixel will ultimately be drawn as.

2.5 3D DMVN WebGL Shader Programs

In our project, we need three different shader programs, for three different elements that must be drawn in different ways:

1. The video planar section (drawn as triangles);

2. The wire-frame box volume (drawn as lines); and

3. The motion trajectory path (drawn as a line strip)

2.5.1 Video Planar-Section Shader

The Video Planar-Section Shader shows the rotated rectangle with current video frame.

Buffers & Attributes

The rectangle is defined as six 3D Position points, with each set of three comprising a single triangle primitive. We define the points to cover the entirety of the xy plane in the Normalized Device Coordinate space, with $z = 0$.

There are also six 2D texture coordinate points defined to correspond to the six 3D Position point data. These are defined within the Normalized Texture Coordinate space, with $\langle 0, 0\rangle$ in the upper-left corner, and $\langle 1, 1\rangle$ in the lower-right.

Textures

The video frame is the only Texture used in our project, and it is updated (re-uploaded to the GPU) before every single draw call to ensure that it reflects the contents of the hidden `video` DOM element.

Uniforms

The only uniform, aside from the Texture Sampler, is the 3x3 transformation matrix, covered later in the Mathematics chapter. Unlike the other two shader programs, this transformation is applied without a prior shift along the z-axis, as the video is meant to always be positioned in the center.

Vertex Shader

The Vertex Shader simply applies the transformation matrix to each vertex, and reports that Position to the Clipping and Rasterization stages. Since depth buffering is not utilized, we report the Position as the projection onto the xy plane in Normalized Device Coordinates, and pass the z coordinate as a varying to the Fragment Shader for our own use.

Fragment Shader

While the Fragment Shader is also very basic, largely performing the operation of sampling the texture and pushing that sample's color as its output, it does take the z coordinate passed from the Vertex Shader stage into account to brighten or darken the pixel slightly based on its depth.

Since 3D DMVN uses orthographic projection, I decided that some indication of what is near or far is helpful to the user, and so chose to do it through this mechanism or brigthening closer pixels and darkening further pixels.

2.5.2 Wire-Frame Box-Volume Shader

The Wire-Frame Box-Volume shader draws an encapsulating volume to visually reinforce the depth component of the video and path, while the video remains translationally stationary in the center.

Buffers & Attributes

The only attribute of the Box-Volume vertices is the 3D position. Since they are drawn as lines, each primitive consists of a two vertices: a start and end point. Since the box-volume is defined as a rectangular prism, there are 12 line segments needed to draw it, and so a total of 24 vertices are defined.

In this case, we take advantage of the full range of positional values in the xy plane, $[-1, 1]$, so that the transform matrix will make the xy dimensions of the box the same as the xy dimensions of the video rectangle. We define z on the range of $[0, 1]$, so that its untransformed state would be the position it should be at the beginning of the video.

Uniforms

While the transformation matrix that is passed to the Video Rectangle shader program is also passed to the Box shader program, the Box also receives a `time` uniform, that tells it what the current seek time of the video is, on the range of $[0, 1]$.

Vertex Shader

The Vertex Shader subtracts the value of the `time` uniform from the z coordinate of the initial vertex position *before* applying the matrix transform. In this way, the box volume appears to slide backwards over time as the video plays, always containing the video rectangle within.

Fragment Shader

The Fragment Shader is the simplest possible: it passes through a constant white color with a small alpha value so that the box volume is as unobtrusive as possible.

2.5.3 Motion Trajectory Path Shader

The Motion Trajectory Path Shader draws the motion trajectory "seek" path over the video.

Buffers & Attributes

Unlike the other two shaders, the Path Shader receives vertex position information that is pre-transformed (both a negative z-axis shift, as well as with the matrix applied), since those transformations are pre-applied on the CPU for purposes of mouse-proximity calculation, and thus there was no need for redundant calculation. Whether the memory cost of retransmitting the positional buffers every frame is more than the perceived cost of redundant calculation is unverified.

Vertex Shader

The Vertex Shader simply reports the position of the vertices as that of the xy projection of the respective positional attribute. The z coordinate of the vertex is passed along to the Fragment Shader as a Varying.

Fragment Shader

While the Fragment Shader always reports of the color of the line to be yellow, it does change the alpha (transparency) of the pixel based on whether it is *in front of* or *behind* the video rectangle, to help the user determine depth.

Chapter 3

Mathematics

3.1 Coordinate Systems

As in most graphics applications, throughout our implementation we work with many different coordinate systems, and must convert between them.

3.1.1 Video (Pixel) Space

When operating on the video frames in Python using `opencv` and `numpy`, we generally consider those images as matrices (where each pixel is an element), and so use standard mathematical row-column indexing: M_{ij}, where i is the row and j is the column. This translates to M_{yx}, where y is the vertical coordinate and x the horizontal, which may be counterintuitive to some graphics programmers.

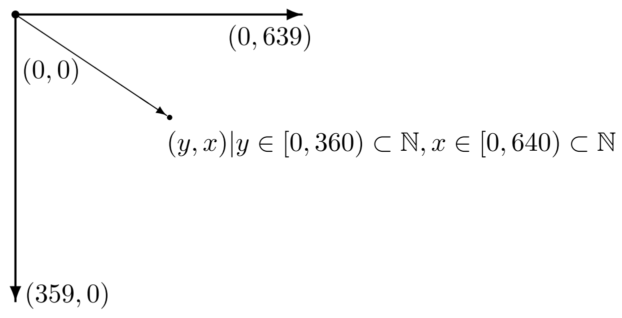
$(0, 639)$
$(0, 0)$
$(y, x)|y \in [0, 360) \subset \mathbb{N}, x \in [0, 640) \subset \mathbb{N}$
$(359, 0)$

Figure 3.1: Video (Pixel) Space

3.1.2 Normalized Texture Coordinates

OpenGL's shader language GLSL addresses *texels*, or pixels fetched from textures, through *Normalized Texture Coordinates*. Unlike many other image/texture addressing systems, this system uses real numbers instead of natural numbers, as the GPU Texture Unit hardware will perform bilinear interpolation to enable subpixel fetching.

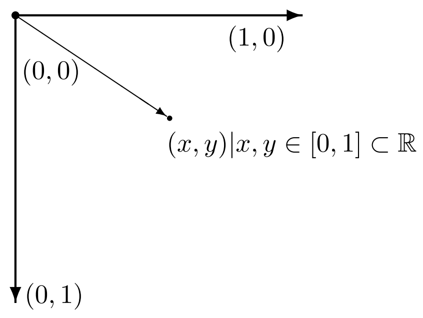

Figure 3.2: Normalized Texture Coordinates

3.1.3 Normalized Device Coordinates

Normalized Device Coordinates (NDC) are the only 3D coordinate system used in our project. It is used by OpenGL: the Vertex Shader stage must present a vertex position to the rendering engine in this coordinate space. A left-handed $\mathbb{R}^3$ space where x, y, and z are all on $[-1, 1]$, any draw primitive (usually a triangle) where all vertices are outside of that domain will be clipped (not rendered).

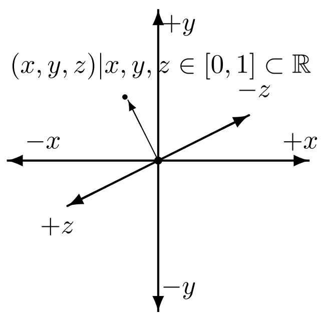

Figure 3.3: OpenGL Normalized Device Coordinates

OpenGL uses the following algorithm to determine whether to clip a polygon:

1. For each vertex $\vec{v}$ in each polygon (triangle) drawn:
2. If $|v_x|$, $|v_y|$, or $|v_z| > 1$, then that vertex is marked for clipping.
3. If all vertices $\vec{v}$ in a polygon are marked for clipping, then the polygon is not drawn (is clipped)

3.1.4 DOM Space

In the context of HTML pages, positions (particularly: mouse positions and such) are measured in "DOM Space", an integral pixel space with origin in the upper-left corner *of the containing DOM element.* For example, if handling an event of a `canvas` element, then the DOM Space's origin will be in the upper-left corner of that `canvas` element, and its domain also bound by that element's pixel width and height.

As DOM Space's origin is measured as $(0, 0)$, its domain is $x \in [0, w) \subset (N)$ and $y \in [0, h] \subset \mathbb{N}$, where w is the width and h is the height of the containing object in pixels.

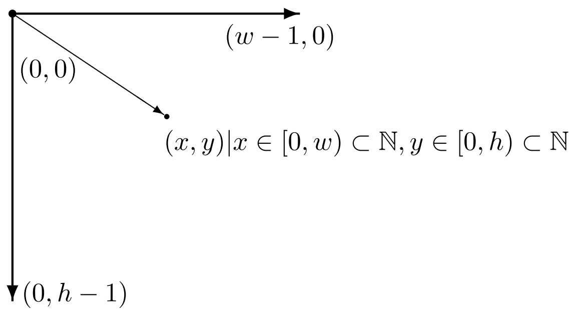
$(w-1, 0)$
$(0, 0)$
$(x, y)|x \in [0, w) \subset \mathbb{N}, y \in [0, h) \subset \mathbb{N}$
$(0, h-1)$

Figure 3.4: DOM Space

3.2 Coordinate Transforms

We can use simple linear operations to convert between these coordinate spaces. To demonstrated, we will show the conversions to or from Normalized Texture Coordinates. In some cases, both Video and DOM Space will be referred to as Pixel Space, whenever the order of the coordinates is not relevant. (w is the width and h is the height of the image in pixel space.)

Pixel Space $\vec{p}$ to Normalized Texture Coordinates $\vec{t}$

$$\vec{t} = \langle \frac{p_x}{w-1}, \frac{p_y}{h-1} \rangle \tag{3.1}$$

Normalized Device Coordinates $\vec{n}$ to Normalized Texture Coordinates $\vec{t}$

$$\vec{t} = \langle \frac{n_x+1}{2}, \frac{n_y+1}{2} \rangle \tag{3.2}$$

Normalized Texture Coordinates $\vec{t}$ to Video Space $\vec{v}$

$$\vec{v} = \langle ht_y, wt_x \rangle \tag{3.3}$$

Normalized Texture Coordinates $\vec{t}$ to DOM Space $\vec{d}$

$$\vec{d} = \langle wt_x, ht_y \rangle \tag{3.4}$$

Normalized Texture Coordinates $\vec{t}$ to Normalized Device Coordinates $\vec{n}$

$$\vec{n} = \langle 2t_x - 1, 2t_y - 1, 0 \rangle \tag{3.5}$$

3.3 Spatial Transforms

3.3.1 Scaling Matrices

$$S(\vec{s}) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tag{3.6}$$

3.3.2 Rotation Matrices

Rotation About the x Axis

$$R_x(\phi) = \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix} \tag{3.7}$$

Rotation About the y Axis

$$R_y(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{3.8}$$

3.3.3 Projection Matrices

For 3D DMVN, we opt to use Orthogonal instead of Pespective Projection, in order to minimize distortion of the video and path under rotation. Since we've further optimized the vertex positions of the objects to be drawn to Normalized Device Coordinates, all our Projection Matrix needs to do is to compensate for the aspect ratio of the viewport window; we can do this with Scaling Matrix, as above.

3.3.4 3D DMVN Matrices

Transform Matrix

The transformations that must be performed are, in order:

1. Scaling to the video aspect ratio η_{video} and the "reduction factor" ρ, which describes how much smaller the video should be than the viewport as a whole.

2. X Rotation, based off of the user mouse input
3. Y Rotation, based off of the user mouse input
4. The Projection Matrix, which is simply a Scaling Matrix to correct for the `canvas` element's (viewport's) aspect ratio $\eta_{viewport}$.

This Transform Matrix T is derived as follows:

$$\begin{cases} S' &= S(\rho, \rho\eta_{video}, 1) \\ R' &= R_y(\theta)R_x(\phi) \\ P' &= S(1, \eta_{viewport}, 1) \\ T &= P'R'S' \end{cases} \tag{3.9}$$

Application

For the drawing of the Video Box itself, we apply the transform matrix T to each vertex $\vec{v}$ of the video rectangle to obtain the transformed vertex $\vec{v}'$:

$$\vec{v}' = T\vec{v} \tag{3.10}$$

For the path and box, since we want them to move translationally with video time $0 \leq t \leq 1$, we first modify each vertex's z coordinate before applying the transform matrix:

$$\begin{cases} \vec{v}' &= \langle v_x, v_y, v_z - t\rangle \\ \vec{v}'' &= T\vec{v}' \end{cases} \tag{3.11}$$

3.4 DMVN Interactivity

The user interacts with the DMVN three different ways:

1. rotating the video and path;
2. dragging along the path to temporally navigate the video; and
3. selecting a pixel to derive a new path from.

3.4.1 Rotating the Scene

Whenever the user *right-click drags* their mouse, we translate the mouse movement in (x, y) DOM Space to Euler angles (yaw and pitch, ϕ and θ), ensuring that neither ϕ nor θ never exceed ± 45 deg. ϕ and θ are used in the generation of the 3D x and y rotation matrices for our 3D DMVN Matrix.

3.4.2 Temporal Navigation

Clicking the left mouse button triggers temporal navigation. While holding the button down and dragging, we iterate over the entire set of path points $\vec{p}$ and choose a "nearest" one to the normalized mouse location $\vec{m}$ as follows:

$$t' = \arg\min_{p_t} \sqrt{(p_x - m_x)^2 + (p_y - m_y)^2 + \kappa(p_t - m_t)^2} \tag{3.12}$$

This is a straightforward Euclidean distance between our "point of selection" $\vec{m}$ and $\vec{p}$. t is the time of the current video frame, and κ is a scaling constant meant to be tuned to allow for smooth navigation. (We found $\kappa = .9$ to work well.)

We did find preferable behavior in using a simpler, atemporal distance when doing the "initial seek":

$$t' = \arg\min_{p_t} \sqrt{(p_x - m_x)^2 + (p_y - m_y)^2} \tag{3.13}$$

as when a user initially clicks to begin navigation, they are rarely considering temporal coherence.

We did optimize the performance of the code by removing the square-root operation, since it isn't ultimately relevant to the result, as

$$\arg\min \sqrt{x_i} = \arg\min x_i \tag{3.14}$$

3.4.3 Path Pixel Selection

3.5 Ray Tracing

We wanted the user to be able to select a pixel from the video, regardless of the current orientation of that video in $\mathbb{R}^3$ space and despite all of those above transforms. In order to do this, we must cast a ray where the user middle-clicks with their mouse, down into that $\mathbb{R}^3$ space, and determine its point of intersection with the planar surface of the video in a way in which we can then determine the pixel coordinates of such intersection.

3.5.1 Designing a Parametric Intersection Equation

We decided to solve this problem by expressing it as a system of parametric vector equations. Since our video surface, prior to transformations, exists as a "full-screen quad" in the XY plane from $[-1, 1]$, we start with three points on that quad: the upper-left corner $\vec{a} = \langle -1, 1, 0\rangle$, the upper-right corner $\vec{b} = \langle 1, 1, 0\rangle$, and the lower-left corner $\vec{c} = \langle 1, -1, 0\rangle$. Then, we apply our global scaling/rotation transformation matrix, M to those three points; for example, $\vec{a}' = M\vec{a}$.

The parametric form of a plane can be expressed as $\vec{p_0} + \vec{u}s + \vec{v}t$, where $\vec{p_0}$ is any reference point on that plane, and $\vec{u}$ and $\vec{v}$ are any rays along its surface. In order to utilize this parametric form to solve my problem, I will let $\vec{u}$ define the horizontal axis of my video frame on $[0,1]$, and $\vec{v}$ represent the vertical axis of my video frame, also on $[0,1]$, similar to the *Normalized Texture Coordinates* described above. By doing this, we can easily convert these normalized coordinates to pixel coordinates, and we can also easily test for rays that do not intersect the transformed video plane by checking if the point of intersection lies outside of $[0,1]$ regardless of the pixel dimensions of the video.

Since $\vec{a}'$ is the origin of our desired coordinate space, we will use it as $\vec{p_0}$. As $\vec{b}'$ is also the point where the horizontal axis of the video is maximally in bounds, we can define $\vec{u} = \vec{b}' - \vec{a}'$; similarly, for the vertical axis we can define $\vec{v} = \vec{c}' - \vec{a}'$. It follows, then, that we can express any point $\vec{z}$ on our plane as

$$\vec{z} = \vec{a}' + \vec{u}s + \vec{v}t \tag{3.15}$$

where s and t are the parametric variables.

For the ray that is cast by the user's middle-mouse click, we can express it also in parametric form as $\vec{p_0} + \vec{d}r$, where $\vec{p_0}$ is the position vector for the origin of the ray, $\vec{d}$ is the direction vector, and r is the parametric variable. When we receive the middle-mouse click event from the DOM, it will be in *DOM Space*, and so will need to be normalized to the dimensions of the `canvas` object that the WebGL context is in. If we assume that this ray is cast "downward" from the (x, y) point, or $\vec{m} = \langle m_x, m_y \rangle$ point where the user selected, then we can take $\vec{p_0} = \langle m_x, m_y, 0 \rangle$ and $\vec{d} = \langle 0, 0, -1 \rangle$. Therefore, the parametric equation for a point $\vec{z}$ on our ray should be:

$$\vec{z} = \langle m_x, m_y, 0 \rangle + \langle 0, 0, -1 \rangle r \tag{3.16}$$

You will notice that I've set both of the above equations equal to $\vec{z}$; by having the same point equal to both the parametric equation for the plane as well as the ray, $\vec{z}$ becomes a point of intersection between the two. However, it then also becomes redundant; we can now describe the intersection of the ray and the plane as:

$$\vec{a}' + \vec{u}s + \vec{v}t = \langle m_x, m_y, 0 \rangle + \langle 0, 0, -1 \rangle r \tag{3.17}$$

3.5.2 Solving the Intersection Equation

While our parametric equation above is in vector notation, it could equivalently described as a system of equations as follows:

$$\begin{cases} a'_x + u_x s + v_x t &= m_x \\ a'_y + u_y s + v_y t &= m_y \\ a'_z + u_z s + v_z t &= -r \end{cases} \tag{3.18}$$

We have three unknowns (s, t, and r) and three equations in our system. Our plan is to solve them using matrices, so our first order is to reorganize the equations so that all unknowns and their coefficients are represented on one side of the equation, and constants on the other:

$$\begin{cases} u_x s + v_x t + 0r & = m_x - a'_x \\ u_y s + v_y t + 0r & = m_y - a'_y \\ u_z s + v_z t + 1r & = -a'_z \end{cases} \tag{3.19}$$

This simultaneous equation can then be expressed as a matrix equation by separating the coefficients, unknown variables, and constants into three separate matrices:

$$\begin{bmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ u_z & v_z & 1 \end{bmatrix} + \begin{bmatrix} s \\ t \\ r \end{bmatrix} = \begin{bmatrix} m_x - a'_x \\ m_y - a'_y \\ -a_z \end{bmatrix} \tag{3.20}$$

We will solve this equation whenever the user middle-clicks on a part of the WebGL `canvas` element by taking advantage of the following property:

$$AX = B \iff X = A^{-1}B \tag{3.21}$$

There are many methods to find the inverse of the 3×3 matrix; we'll do it through this process:

1. Calculate the Matrix of Minors:
   (a) For each element m_{ij} in the matrix A, collect a 2×2 matrix of all elements that are in neither the row i nor column j of the current element;
   (b) Find the determinant of that 2×2 matrix, and assign that value to m_{ij} of the Matrix of Minors.
2. Calculate the Matrix of Cofactors by negating m_{01}, m_{10}, m_{12}, and m_{21} of the Matrix of Minors.
3. Transpose the Matrix of Cofactors to obtain the Adjugate.
4. Calculate the determinate of the original matrix A:
   (a) This can be optimized by using the already-calculated values in the Matrix of Minors
5. The inverse matrix A^{-1} is the Adjugate divided by the determinate of A: $A^{-1} = \frac{\text{adj}A}{|A|}$

Chapter 4

Future Work

4.1 Frame Independence of Path Selection

The primary disappointment of my project is that I could not determine a method (reasonably within the scope of the project) to let the user select the path's origination pixel from any frame except for the first.

4.2 Mobile Device Support

The Web now enjoys a substantial user base using mobile platforms, such as smartphones or tablets. There exists an opportunity to expand this project's client interface to support such devices. While performance optimizations (discussed below) are particularly beneficial on mobile systems, several unique challenges present themselves:

4.2.1 Fat Finger Problem

The so-called "Fat Finger Problem", where a user's finger occludes the very area of interest on the screen that the user is attempting to interact with, preventing fine manual interaction.

4.2.2 Motion Sensors

A Majority of mobile devices contain *motion sensors*, which generally constitute some combination of:

- Gyroscopes, which can detect changes in angular orientation;
- Accelerometer, which can detect translational acceleration, including determination of the Gravity Vector; and
- Magnetometer, which can be used as a compass to detect an absolute bearing, relative to Magnetic North.

Often, the information from the above sensors is passed through *motion fusion* algorithms, in order to negate the individual weaknesses of any one variety of sensor, and derive between them all an absolute attitude (or orientation) of the device in real space.

Access to this orientation is provided through the HTML DOM, and can be used as a novel means of user interactivity. For example, the user could physically rotate the device in order to rotate the video and its motion trajectory path, instead of dragging the mouse.

4.3 Optimization

4.3.1 Server Data Reduction

My implementation of a WWW 3D DMVN system requires a large amount of stored data on the server: several Gibibytes per short, low-resolution sample video. This represents a meaningful cost to the host of such a DMVN service, and also has a significant impact on the overall network traffic between server and client. We have several opportunities to reduce that burden:

Storage Format

JSON was selected for its JavaScript-native deserialization support, but as a text format it is very verbose. A significant amount of space could be saved by transitioning to a binary alternative, such as *BSON* (Binary JSON).

Motionless Pixels

Many pixels do not move significantly throughout a scene. Not only does this causesubstantial waste in storing meaningless data, but it can also be confusing to a user when they click on a pixel and get a straight line. If a path is nearly motionless, it could be either:

1. Noted in that pixel's respective file as a "motionless pixel", rather than explicitly list every frame's vertex. This would produce effectively the same behavior as presently.

2. Noted in that pixel's respective file as a "motionless pixel", with redirection to the nearest (2D Euclidean) pixel that *does* have motion, so that the user does not need to search for semantically useful objects to interact with.

3. Noted in that pixel's respective file as a "motionless pixel", causing no generation of a new path when the user attempts to select that pixel, with perhaps some negative feedback (such as a tone or screen blink).

Of these options, I suspect #2 might be preferable.

Path Resolution

Currently, we supply one path vertex per frame, or the maximum possible path resolution. This may not be practically necessary; a vertex every second or even fifth frame may suffice for an acceptable user experience.

Elision of z Coordinate

The inclusion of the z coordinate for each vertex is not truly necessary, as that information could be reconstructed on the client side by dividing each array element's index by the element count, minus one. This would add a small amount of performance overhead to the client process, but since it only occurs once each time a path is loaded, its overall impact should be minimal for the amount of space saved on the server (nearly $\div 13$).

4.3.2 Client Performance

Since user selection of a point along the path (path vertex with nearest temporally-weighted Euclidean distance) is a *reduction* operation over the set of vertices, it is ill suited for GPU implementation. Hence, we resolved to perform the per-frame matrix-multiplication (transformation) of the path vertices on the CPU, and then send those to the GPU for display, since otherwise the transformed vertices necessary for user-mouse interaction would be redundantly calculate.

In other words, since this fairly heavyweight calculation was necessary to compute on the CPU, we decided not to *also* reperform the computation on the GPU; the transformed data is instead sent to the GPU. However, there is an advanced feature of OpenGL called *Transform Feedback* that would allow the transformed vertices from the GPU to be made available in a buffer accessible to the CPU. The end-result would be identical in that redundant calculation is avoided, but instead of the singular calculation happening on the CPU, it would happen on the GPU instead, which should be more performant.

4.4 Semantics & Intuitiveness

4.4.1 Semantically Interesting Motion

The fundamental purpose of DMVN is to allow for a more intuitive video-navigation method: the user is enabled to drag an "object of interest" through space to navigate to the corresponding time in the video. These "objects of interest" may often be people, animals, or objects. However, motion beyond of these "objects of interest" would generally serve to cloud the user experience, especially in situations that may challenge conventional computer vision techniques, or otherwise cannot be resolved to a singular, clear path of motion. For example, fluids such as smoke, rain, rustling leaves, bodies of water, or perhaps even clouds do not exhibit the type of motion that we wish to capture and express to the user as a "motion path" for the purposes of navigation.

We can categorize these kinds of unwanted motion as follows:

Incoherent Motion

The motion of leaves rustling, smoke billowing, sea foam. To address this, I might:

1. segment the optical flow image into blobs of areas of coherent motion, or
2. otherwise perform some statistical analysis of pixel neighborhoods to filter out motion that doesn't show cohesive motion over a relatively large pixel area

This would of course risk filtering out smaller objects that *would be* semantically interesting to track, such as a flying bird in the distance.

Global Motion

"Global motion" covers issues such as:

- camera motion, either rotational, translational, or both;
- rain, fog, mist, or other atmospheric motion (this also applies if the video is entirely of a body of water); and
- noise, such as that from the video source (CCD/CMOS image sensor or film) or from compression.

4.4.2 Immobile Regions

In our example videos, and in many others where the camera is immobile, a majority of the video may be considered to either be an "immobile region": when the user selects a pixel to generate a motion-path, they will be greeted with what appears to be a straight line. This may cause confusion to the user, as they have been told that DMVN helps them navigate by dragging objects, and yet selecting an area that is not an object causes what may appear to them to be confusing behavior.

4.4.3 Camera Motion

Camera motion provides some unique challenges to DMVN; the examples demonstrating DMVN avoid this issue, because it is difficult to overcome. Since the intent of the motion path is to be able to visualize the motion of the object throughout a scene, the true path would become confusing if the frame of reference of that motion were to change mid-stream.

While solutions exist for camera pose and motion estimation (and for the related field of video stabilization), how can these techniques be used in the context of DMVN? I would propose experimentation of the following scheme:

Basic 3D DMVN allows the user to rotate the video and path in the x- and y-axes; the video planar section otherwise stays at the origin (translationally), and is not rotated in the z-axis. In an enhanced model, when the camera undergoes some transformation (translation, rotation, etc), then the video planar section could counter-transform in the opposite direction, but same amount, in order to achieve a kind of "path stabilization", to ensure that the path's frame of reference remains constant throughout the scene.

4.4.4 Semantic Segmentation

The premise of DMVN is the ability to navigate a video by dragging an object through its motion path; not a pixel and its motion path, as I've implemented it. In order to improve this aspect, there are several questions that need to be answered:

- What are the objects under motion?
- What are the objects under motion *that are semantically interesting*?
- What kinds of motion are interesting enough to use as a means of video navigation?
- What *parts* of an object are worth tracking?

This would be more of an artificial intelligence research area. If the number of semantically interesting paths could be automatically detected and simplified to one or two, then DMVN becomes trivial for both the user and for the host (in terms of reduced data storage needs, user-interactivity requirements, etc).

This can be very challenging, because for example, a person may or may not be one object to track: their hands and legs may be moving in different directions that may be interesting of their own accord, or other times the user may be interested in the path of the whole person.

4.4.5 Scene-Change Detection

Many videos have scene changes, especially those which have been post-produced from a collection of "takes", or live-broadcast videos where there is a mechanism for a Producer to switch between cameras. Generally, it would probably be wise to reset the DMVN path tracking when a scene change is detected.

There may, however, be some instances, where it might be possible to correlate the path between two scenes. As an example, let us imagine a guitar tutorial video: the camera is positioned to show the guitarist's hand positioned on the fretboard so that the viewer can see and analyze the fingerings. The scene changes to a camera that is from the guitarist's point of view, so that the viewer can understand how the fingering might look from their own perspective. It could be possible to track the path of the finger between the two scenes.

If an affine transform of the camera's scene change can be derived, then it may be possible to keep the finger's path as a continuous line, but to perform that affine transform on the video's rectangle.

4.4.6 Continuity

References