ModernDvi: A high quality rendering and modern DVI viewer

Antoine Bossard and Takeyuki Nagao

Abstract

TEX users have long relied on the device independent file format (DVI) to preview their documents while editing. However, innovation has been scarce in this area, and users have to rely on years-old, or even decades-old software, facing increasing compatibility issues with modern systems. In this paper, we describe ModernDvi, a new DVI viewer Windows Store application, offering high quality and fast rendering, wait-free, outperforming existing solutions in these areas. Additionally, ModernDvi has been built around today's usability standards and expectations: tablets, touch-friendly, high-resolution output are examples of addressed issues.

1 Introduction: The DVI file format

DVI is a file format, namely the "DeVice Independent" file format. DVI documents are typically produced by the TEX and LATEX typesetting programmes. TEX and its high-level abstraction LATEX, which is written in the TEX macro language, were introduced by Donald E. Knuth [7] and Leslie Lamport [10], respectively, as solutions for producing high-quality documents dealing with mathematics and science in general, and especially their complex formulæ and notations. Hàn Thế Thành later created an extension of TEX called pdfTEX [5] enabling direct output of portable document format (PDF) in addition to the traditional DVI format. There are thus two different kinds of output by TEX and LATEX, viz. DVI and PDF. Although the PDF format has become more popular, some people still need and depend on the classical DVI format. In fact, an experiment with the total 5814 LATEX documents collected from the preprints of arXiv [6] in the single month of January 2012, shows that 4168 items (approx. 72%) can be compiled by both LATEX and pdfLATEX, and that 540 items (approx. 9%) work with LATEX but not with pdfLATEX [13].

The DVI format is minimalistic. Roughly speaking, it is a binary format consisting of commands to (i) define or select a font to utilize, (ii) draw a single character or a filled rectangle at the current reference point, (iii) manipulate internal integer registers (including the current reference point and the identifier of the current font), (iv) include binary data (called DVI specials) for various purposes, and (v) mark the beginning and ending of pages/documents.

The simplicity of the DVI format facilitates creation of tools and applications to help authors prepare manuscripts and post-process existing articles. Such tools include DVI viewers which render and display the contents of a DVI file on screen, and also converters to various formats including PostScript (dvips), PDF (dvipdfm), bitmap images (dvipng), etc. It is much harder to create such tools for the PDF format, since that format is more complicated and thus difficult to parse and analyse the encoded data.

A major drawback of the DVI format is that it requires external files and/or tools to completely render its contents in some use cases. For example, if an author includes a figure in his paper (e.g. using Encapsulated PostScript format), then the generated DVI file contains a DVI special that consists of a code fragment in the PostScript language to include the specified image file. This means that the DVI file is not self-contained, and one needs a rasterizer of PostScript, e.g. Ghostscript, to render its contents. Another common issue is the lack of the feature to embed fonts. A DVI file actually contains only the name (such as cmr10) and the size of the utilized fonts. There is no standardized way to embed raster or vector fonts to a DVI file. This difficulty can be overcome by using pdf(LA)TEX which provides the features of including external PDF files and embedding TrueType and Type 1 fonts.

2 Previous works

A handful of DVI viewers exist; however, many are not updated any more: mdvi [2] and windvi [15] are examples. Well-known viewers for Unix-based platforms include xdvi [17]. Still usable alternatives are even harder to find on the MS Windows operating system. YAP [16] is the DVI viewer of choice on Windows as it is bundled with the MiKTEX distribution [16]. One can also cite dviout [14] as another DVI viewer on Windows.

A user looking for a viable solution to work with DVI files will face several issues. First, all existing viewers are now considered legacy software: they have been designed for decades-old operating systems and do not meet modern requirements regarding software, hardware, interface, and usability in general. The first problem a user may encountered is software compatibility: for example, as of the Mountain Lion release of Mac OS X, X11 is not included any more, which will thus hamper the installation and usage of the usual viewer on Unix, xdvi. Even more radical changes to an OS (e.g. at the device driver layer) may completely break compatibility and make a legacy viewer unusable. Additionally, classic workstations are now on their way out, and mobile devices, touch screens or other advanced interface mechanisms are in full swing. An unadapted user interface such

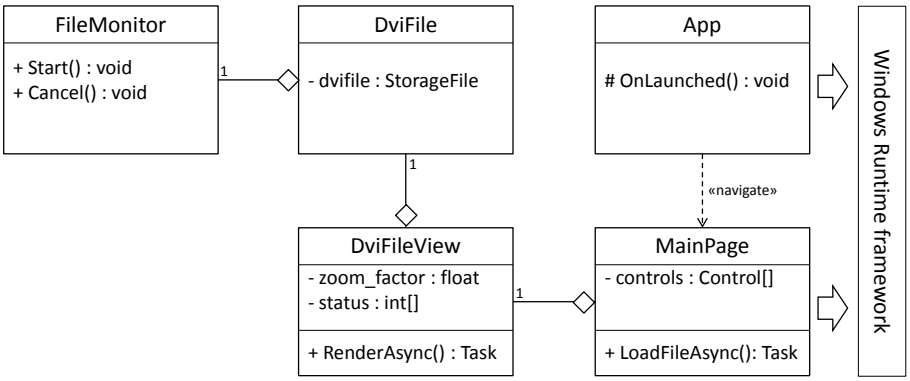

Figure 1: Simplified UML class diagram.

as that of a legacy viewer will thus severely harm usability, or even make it completely unusable.

Hardware evolution affects more than user interfaces. Processors have also seen important changes in their architectures. Now, every device uses a multicore CPU, thus allowing faster or more calculations at the same time: this is parallel processing. Legacy DVI software has usually not been designed to utilise such increased computing power and will thus perform poorly compared to modern applications in general, such ad word processors.

Finally, rendering quality of DVI viewers is usually not on par with modern displays and their high resolutions. Again, when they were introduced, these systems had to cope with much more restrictive hardware and software environments that we have now. Nowadays, users' expectations regarding display quality are very high, and legacy viewers are lacking in this area.

We could continue enumerating shortcomings in legacy DVI viewers (e.g. font generation before rendering, slow page scrolling, etc.), but the point is clear. By proposing our new DVI viewer, named ModernDvi, we are first aiming at filling the gaps of legacy DVI viewers, gaps which have been increasing over time due to constant technology evolution. The introduction of original features as detailed in Section 4 is also an important part of our work.

3 ModernDvi: System overview

In this section, we give some insight into ModernDvi's structure by first addressing software engineering considerations, then rendering techniques, and finally parallel processing.

3.1 Software engineering

ModernDvi is a Windows Store application [11]. It makes use of the Windows Runtime API and is thus compatible with x86 and ARM processor architectures. Windows Store applications can be deployed to any Windows 8 device, including PCs and tablets. Porting ModernDvi to the Windows Phone platform is also possible, but this remains work in progress.

This app has been developed using the C# programming language, and thus features an object-oriented architecture. Let us give an overview of the main objects defined in ModernDvi. Common to every Windows Store app, the entry point of the application is located inside the `App` class. The app itself is built around the `MainPage` class which defines the application view port, displaying controls and visual elements in general. Each DVI document is associated with an instance of the `DviFile` class which, amongst others, importantly stores the system handle to the DVI file, a critical app issue detailed further in Section 5. The `DviFile` class also holds a reference to a `FileMonitor` object which is in charge of monitoring file changes made to the DVI document. Then, in order to display the content of a DVI document (i.e. an instance of `DviFile`), the class `DviFileView` is instantiated. Such an object is in charge of rendering each page of the DVI document, and thus contains several view settings such as page dimension and zooming information. Changing parameters like the zoom factor will automatically create a new `DviFileView` object. Figure 1 shows a simplified UML class diagram of ModernDvi.

An important point is that `DviFileView` objects are completely isolated, especially from the view port objects of `MainPage` so as to facilitate the multi-thread approach detailed in Section 3.3 below. The sole relation between these two classes is a reference to a `DviFileView` instance inside `MainPage`; this reference is used to add (i.e. display) the view inside the view port. The DVI document is loaded with the `LoadFileAsync()` method of `MainPage`, and rendered with the `RenderAsync()` method of `DviFileView`.

3.2 Rendering

The rendering of a DVI file is performed in two stages. In the first stage, the content of the DVI file is parsed and the result stored in an object, which contains a font table (i.e. a mapping from font numbers to font names) and the DVI commands for each page. All the required TEX font metrics and other files (such as PK fonts and virtual fonts) are loaded into memory for later reference. The bounding box of every page is computed at this point, in parallel using multiple threads.

In the second stage, the contents of each page is rendered lazily. More precisely, it is only at the moment when a page is about to appear in the view that the rendering of the page is undertaken. The engine prepares an off-screen buffer for the page, and rasterizes the glyphs onto this buffer according to the DVI commands corresponding to the page. The off-screen buffer is then converted to an image file (using Portable Network Graphics format) and stored in memory. Compression is utilized here in order to save memory space.

3.3 Parallel processing

As recalled in Section 2, parallel processing is now a common feature of our modern devices, from PCs to smartphones via tablets. Modern applications are thus expected to make use of this increased computational power available, and this is what we achieved with ModernDvi.

ModernDvi uses parallelisation for two distinct tasks. First, DVI rendering, as briefly detailed in Section 3.2, needs to compute the bounding box of each page of the DVI document. This is a time consuming task. So, by executing these calculations in parallel, we were able to significantly speed up this process. In practice, we relied on the `Task.Run()` function of the `System.Threading.Tasks` class of the API which queues its parameter to run on the thread pool managed by the framework.

Then, once pages have been prepared in memory, comes the display phase: bringing inside the view port each of the (visible) pages. This task is also time consuming since it involves I/O stream operations, UI elements' instantiation and display, and of course placement routines for correct positioning of the pages and their corresponding visual elements in the view port. So again, instead of performing these tasks sequentially, one after the other, we have devised a parallel solution for this lengthy process and observed significant time gains.

Because of the strongly asynchronous nature of C# for apps, user actions in the UI are often partly postponed after their start via the keyword combination `async/await`, and the program returns to the UI thread. The merit of this approach is that the UI is always responsive, that is non-blocking, lock-free and wait-free. However, this can also be problematic as it means, in our case, that a user can request a document load (i.e. view refresh) several times, with most of these requests being still processed, that is not completed yet. To address this issue, ModernDvi tracks the rendering state of each page of the DVI document through a `status[]` array indexed on document pages, whose values are as follows:

0: page not displayed (and not pending); this is the default status of each page.

1: page pending; the program is preparing the document page.

2: page ready to be displayed; the program has finished preparing the page.

3: page displayed; the program has finished adding the page into the view.

Each view refresh task is thus accessing this array, which is a class member of `DviFileView`. So, we have to regulate the accesses of these asynchronous tasks to this array to avoid race conditions and concurrent access problems. This problem is solved by using the compare-and-swap (CAS) mechanism which is implemented in C# by the `CompareExchange` method of the `System.Threading.Interlocked` class. Using this, we are able to automatically check and update the rendering status of a document page. The practical result is that the Interlocked functionality allows us to (1) avoid performing the rendering of one page several times, and (2) avoid displaying the same rendered page several times.

So, thanks to this approach, our DVI document rendering process is (1) performed asynchronously, thus always retaining the UI responsiveness with, for example, very smooth scrolling, and (2) handling multiple page preparations and displays in parallel, thanks to multiple threads. The number of threads is managed by the framework thread pool and is thus transparent to the developer. Obviously, the more cores in your device, the more threads can be running concurrently. An excerpt of the corresponding source code is given in Figure 2.

4 Notable features

We present in this section several notable features of ModernDvi.

4.1 No font generation needed

Existing DVI viewers generate on-demand PK font files from, for instance, METAFONT source files [8, 9] or PostScript font files [1]. The main problem with

```
IEnumerable<Task> tasks = Enumerable.Range(0,
                      visible_pages).Select(i => {
 return Task.Run( () => {
  int current_page = first_page + i;
  int original_status = System.Threading.Interlocked
   .CompareExchange (ref status[current_page], 1, 0);

  if (original_status == 0) {
   pages[current_page] = DviFile.ctx
               .CreateRenderablePage(dvifile.Document
               .GetPage(current_page));
   status[current_page] = 2;
  }
 });
});
await Task.WhenAll(tasks);
```

Figure 2: Parallel execution with `Task.Run()`.

Table 1: Most-used fonts in 2012 arXiv.org preprints.

Rank	Font	Freq.	%	Cumul. %
1	cmsy10	64 830	4.69	4.69
2	cmr10	60 321	4.36	9.05
3	cmmi10	52 705	3.81	12.87
4	cmbx12	48 774	3.53	16.39
5	ptmr8r	44 209	3.20	19.59
6	cmex10	41 872	3.03	22.62
7	cmr8	39 221	2.84	25.46
8	cmbx10	36 725	2.66	28.12
9	cmr6	33 900	2.45	30.57
10	cmmi8	30 895	2.23	32.80
11	(others)	928 916	67.20	100.00
Total		1 382 368		

this approach is the rendering time delay faced by the user upon the DVI document loading.

ModernDvi has adopted a different approach: PK font files are generated in advance and bundled inside the application so that no font generation phase is required at any time. Thus, we can achieve a significant speed-up of the initial loading phase compared to legacy DVI viewers.

Of course, to maximise usability we need to include with ModernDvi the fonts needed by users. Many hundreds of fonts are available for TEX usage; looking at the CTAN font area [4] gives a good overview of the situation. So, we conducted an experiment to measure the popularity of these fonts, and thus obtained a list of the most-used fonts. Specifically, we collected preprints published on arXiv [6] for the year of 2012 and analysed the resulting 48 772 samples of LATEX source files to see which fonts they used, i.e. which font files are needed for their rendering. Table 1 shows the results with, not surprisingly, Computer Modern leading the list.

We observed that about 1,000 fonts sufficed to render all gathered DVI files, which we took as representative of most documents, given the broad area covered by arXiv. We thus gathered the corresponding METAFONT source files and generated PK files for each font using the `mktexpk` utility [3]. Particular care has been taken to avoid any licensing violations for the fonts bundled in ModernDvi, with problematic fonts replaced by freely available ones; for instance, Linotype Palatino has been replaced by URW Palladio. Out of the ≈1,000 fonts identified, 769 have been included in ModernDvi. Excluded fonts are either rarely used or for exotic languages.

4.2 File change monitoring

ModernDvi includes a file monitoring system which triggers a new document rendering (refresh) upon any file change. Concretely, if this feature is enabled, ModernDvi will check at a specific time interval whether changes have occurred to the loaded document. Such changes are detected as follows.

1. Initialise a `LastModified` object of type `DateTimeOffset` with the `DateModified` value of the `BasicProperties` instance returned by a first call to `GetBasicPropertiesAsync` on the current document.
2. Start the timer (`DispatchTimer` object).
3. On timer tick, retrieve a new instance of `BasicProperties` via a call to `GetBasicPropertiesAsync` on the current document. Then compare the stored `LastModified` value with the `DateModified` value of the `BasicProperties` instance just retrieved. If `LastModified` is smaller (i.e. older) than `DateModified`, request a new rendering of the DVI document. Finally, set `LastModified` to `DateModified`.

Additional care needed to be taken regarding `GetBasicPropertiesAsync`. Even though it is not mentioned in the documentation [12], multiple calls to `GetBasicPropertiesAsync` on the same file will throw an exception. Only one call to `GetBasicPropertiesAsync` is allowed at a time: one has to wait for the previous call to return before making another call. And because this method is asynchronous (i.e. awaited, see Section 3.3), we have to enforce a guard to avoid doing so. This is again achieved by using the CAS mechanism `CompareExchange` method of the `System.Interlocked` class.

Lastly, we must note that this monitoring system does not work for DVI files that are contained inside an archive (see Section 4.5). This is due to the limitations imposed by the Windows Runtime API, limitations induced by security concerns. See Section 5 for additional details.

Figure 3: Horizontal display mode.

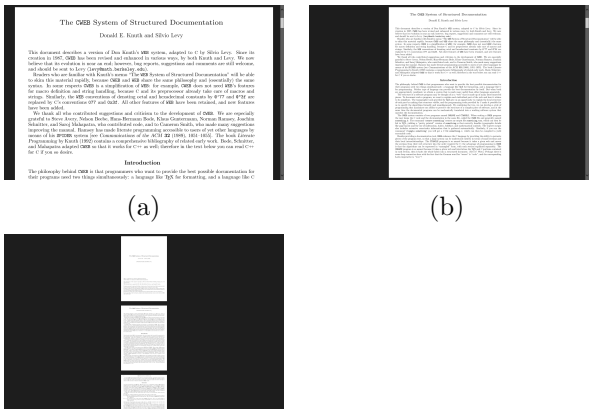

Figure 4: Vertical modes of (a) window fit, (b) page fit and (c) thumbnails zoom levels.

4.3 Vertical and horizontal display modes

It is a steady trend: screens are getting wider and wider. In order to take full advantage of such hardware configurations, we have implemented two different page flows in ModernDvi: vertical and horizontal display modes.

The vertical display mode is the classic top-down document flow that can be found in almost all viewers or WYSIWYG editors. The horizontal display mode is an original left-right document flow that proves comfortable when working on wide displays. Effectively, multiple pages can be displayed side-by-side on a wide screen at the same time, enabling a seamless reading experience. An illustration is given in Figure 3.

In addition to a global setting defining the default display mode of ModernDvi, the user interface contains an easily accessible "Switch flow" button that enables the user to quickly (instantly) switch the document display mode, without having to reload or render anything.

4.4 Automatic zooming

In the current iteration of ModernDvi, we have implemented three zoom levels: window fit, page fit and thumbnails. The user can choose between these modes to render the DVI document by automatically adapting to the screen resolution. Because the document rendering process is repeated when changing the zoom level, the rendering quality remains crisp at any time. An illustration of these three zoom levels is given in Figure 4.

To adapt to the user's screen, our `DviFileView` class (see Figure 1) registers the `Loaded` event of the view port. When fired, this event signals the availability of the `ActualWidth` and `ActualHeight` properties of the view port, giving the user screen resolution in pixels (precisely the screen area allocated to the view port). Because of the asynchronous nature of applications and especially their user interface, failing to use the event model will most likely result in null values for these two properties. In practice, the UI thread may not have completed the interface setup work when reading these two properties.

4.5 Archive formats and file types

DVI documents are often distributed as archives. For instance, most documents in the arXiv preprint repository are stored as tarballs. To facilitate display of such documents, we implemented routines in charge of uncompressing and extracting archives. Files are stored in the temporary folder of the app. Table 2 summarises the file formats ModernDvi supports.

Additionally, so as to correctly handle these different file formats, we implemented an accurate file type detection routine that can recognise each of the supported file formats given a file, regardless of its extension. In practice, one cannot be sure that a file will have an extension, or that it is accurate. And this is without mentioning that there often exist many different extensions for a single file format. Thus, we analyse a file's header data to determine its type.

Table 2: Supported file formats.

Format (MIME type)	Usual extension
application/x-dvi	.dvi
application/x-tar	.tar
application/gzip application/x-gzip-compressed	.gz
application/zip application/x-zip-compressed	.zip
application/x-compressed	.tar.gz, .tgz

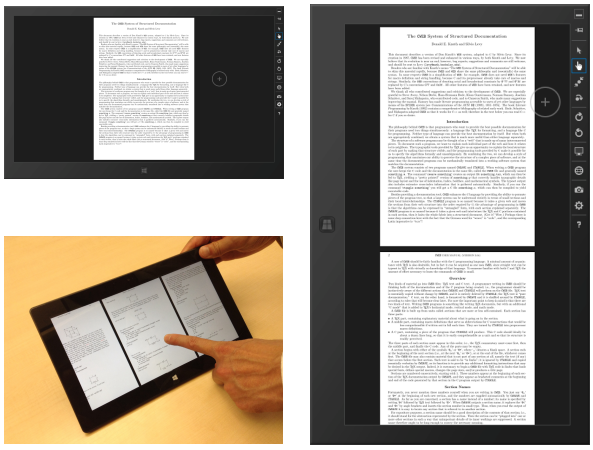

Figure 5: Running ModernDvi on a tablet: full touch and rotation support.

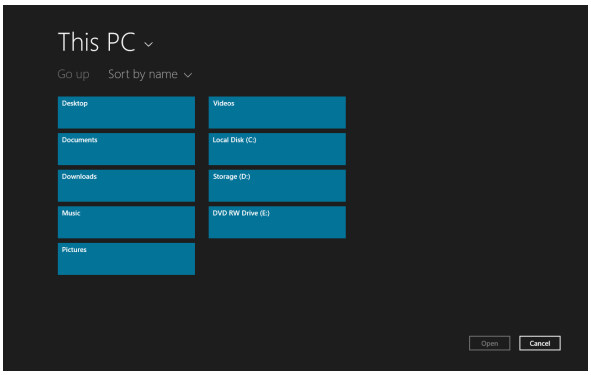

Figure 6: A file selection dialog: the only way to access a user's files.

4.6 Modernity

One of the key aspects of ModernDvi is its recognition of new technologies, devices and interfaces. In recent years, touch screens have heavily changed our habits, and software design had to be significantly overhauled to adapt to such new interfaces. ModernDvi embraces this evolution by providing an innovative, touch-friendly user interface such that it can be similarly used with either a classic mouse-keyboard setup or a touch-enabled device such as a tablet. It is also worth mentioning that ModernDvi is by design compatible with multiple CPU architectures: x86, x64 and ARM. So, computers equipped with (at least) Windows 8 and devices running Windows RT, such as the Microsoft Surface, are all capable of natively running the application. Smartphones running the Windows Phone 8+ operating system can be expected to follow soon due to the common architecture with Windows 8 (NT kernel).

In addition to touch support, ModernDvi has full rotation support: no matter how the user holds the device, the application will update its layout so as to present correctly-oriented content. Figure 5 shows ModernDvi running on a tablet (emulation of an x86 tablet environment). By combining these two features (touch and rotation), the user can conveniently and naturally go through the document as if turning pages of a book by selecting the horizontal display mode and holding his tablet vertically; then, a simple finger sweep will then display the next page.

Finally, ModernDvi has docking support: the user can move ModernDvi onto the side of the screen to interact with another application. This is highly useful for the "Edit-and-preview" use case as detailed in Section 6.1.

5 "App" considerations

Developing a Windows Store application has many advantages compared to a classic, legacy program. First, its distribution, deployment and promotion are greatly facilitated since everything is handled by the operating system through the official Windows Store application (installation, removal, etc.). Also, installing software via the Windows Store is a security guarantee for the user: applications are reviewed before they are added to the Store, and importantly, they run inside a protected environment, ensuring a minimum footprint on the operating system as detailed below.

By design, Windows Store applications (the same is true for Apple Store applications) run in a restricted (sandboxed) environment for security reasons. Thanks to this feature, the user need not worry about system modifications by the app: they are simply not allowed. The application footprint on the system is thus minimal, unlike legacy programs.

One of the main implications of this design is that applications have no direct access to the file system, except for the application's own data folder. To be precise, for an application to perform I/O operations on a file outside the application's data folder, the user must first manually select the file through a file selection dialog control such as a `FileOpenPicker` instance (see Figure 6).

This limitation has a major impact on an application such as ModernDvi. For example, there is no possibility of accessing external files, such as images, which may be referred to in the DVI document. And there is no possibility of running an external program, such as Ghostscript, to delegate PostScript work. A Store application is required to be completely independent. It has thus been a significant challenge to produce a fully functional DVI viewer application.

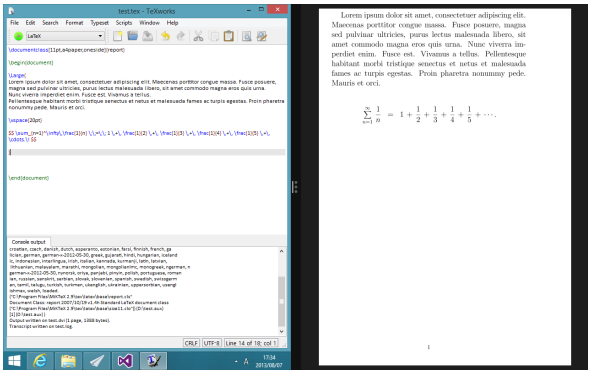

Figure 7: Editing (left) and preview (right) of LaTeX source via the Windows 8 screen splitting feature.

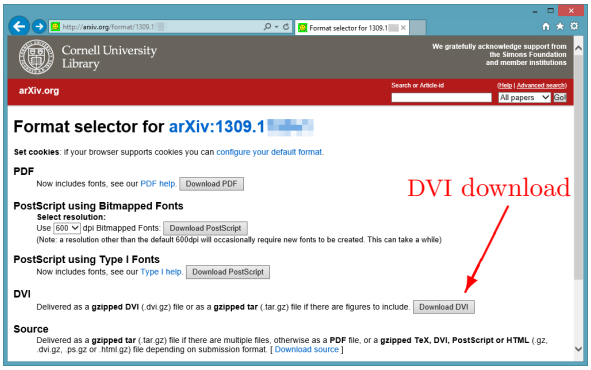

Figure 8: Reading a preprint downloaded from arXiv.org.

As an example, let us consider the file monitoring feature of ModernDvi (see Section 4.2). The Windows runtime does have a folder monitoring capability, namely the `ContentsChanged` event of the `StorageFolderQueryResult` class, but because accessing a file object (`StorageFile`), say via a file picker, does not grant permission to access the containing folder (`StorageFolder`), this is unusable for us. So, we implemented a file monitor by using a timer object (`DispatcherTimer`) to check the value of the `DateModified` property of the `BasicProperties` class instance returned by a call to the `GetBasicPropertiesAsync` method on the file on each clock tick. Because of the high timer resolution, this file monitoring solution is fully satisfactory.

6 Use cases

This section presents two different use case examples for ModernDvi: edit-and-preview, and reading mode.

6.1 Edit-and-preview mode

The user is preparing an article to be submitted to a symposium, using the LaTeX typesetting system. The user opens his LaTeX source file in his favourite editor, say, TeXworks. A first compilation by `latex` is triggered by the user, generating a DVI file. The user double clicks this DVI file to start ModernDvi and load the DVI file into view. As the user wants to continue editing, s/he docks ModernDvi onto the side of the screen, and puts TeXworks on the other side. The screen is thus split into two areas as shown in Figure 7. The user continues to make changes in the LaTeX source file and recompiles the document, still using `latex`. The DVI file generated by the previous run is overwritten by the new one. ModernDvi automatically detects that changes have been made to the DVI file and refreshes its view. The user thus has an instant preview of the changed version.

6.2 Reading mode

The user finds a preprint on the online repository arXiv.org, and presses the link to request the file, which is a `.dvi.gz` compressed DVI document. The browser silently decompresses this gzip compressed content and serves the DVI file to the user. The user's device system asks what should be done with the file: open, save, etc.; the user presses the "Open" button, which automatically loads the document into ModernDvi. See Figure 8. The user selects the "window fit" zoom mode for improved readability and can start going through the document.

7 Comparing rendering quality

In this section, we compare ModernDvi with other DVI viewers regarding rendering quality. It is difficult to compare usability, including speed, since ModernDvi is targeting a different platform and interface. For each of the comparisons, default settings were used.

7.1 vs. YAP

First, let us compare the rendering quality with that of YAP [16]. In both cases, the zoom level is set to match the screen width. We can see in Figure 9 that the rendering quality of ModernDvi is better than that of YAP.

7.2 vs. dviout

Then, we compare the rendering quality with that of dviout [14]. In both cases, the zoom level is set to match the screen width. Again, we can see in Figure 9 that the rendering quality of ModernDvi is better than that of dviout.

7.3 vs. Microsoft Reader

Lastly, let us compare the rendering quality of ModernDvi with that of the PDF viewer included with

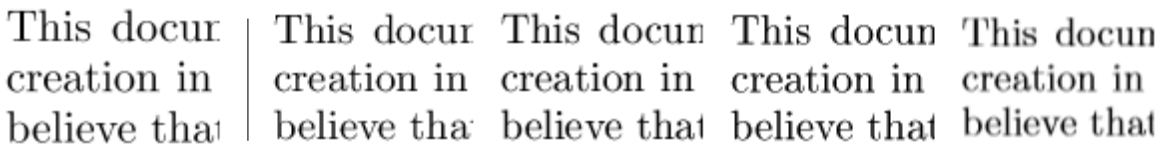

Figure 9: Rendering by (a) ModernDvi, (b) YAP (PostScript mode), (c) YAP (PK mode), (d) dviout and (e) Microsoft Reader (PDF).

Windows 8.1 (Microsoft Reader 6.3.9431.0), after converting the DVI to PDF format with the DVIPDFM*x* utility. In both cases, the zoom level is set to match the screen width. As illustrated by Figure 9, the rendering quality of ModernDvi is significantly better.

8 Conclusions

We have presented in this paper ModernDvi, a new DVI document viewer targeting modern platforms and interfaces, such as tablets. After describing the software architecture, design and features, we compared the rendering quality of other viewers; ModernDvi is leading on that point too. Windows Store applications are sandboxed, and thus severely restricted regarding file system access. We have circumvented these challenges to produce a fully functional DVI viewer application.

Future work includes refining our rendering technique by going down to sub-pixels, as well as improving the rendering speed by continuing the work on glyph caching. Lastly, user interface improvements, such as being able to handle multiple documents at once, are also planned.

A ModernDvi in the Windows Store

ModernDvi can be found in the Windows Store, or directly at `http://apps.microsoft.com/windows/app/moderndvi/bfa836ff-4eb0-454b-ad9e-a2405197f23b`.

References

[1] Adobe Systems Incorporated. *Adobe Type 1 Font Format*. Reading, Massachusetts: Addison-Wesley, 1990.

[2] Matias Atria. MDVI — A DVI previewer. `http://mdvi.sourceforge.net/`. Accessed August 2013.

[3] Edward Barrett. Porting TEX Live to OpenBSD. *TUGboat*, 29(2):303–304, 2008.

[4] CTAN: The Comprehensive TEX Archive Network. Available fonts. `http://www.ctan.org/tex-archive/fonts`. Accessed August 2013.

[5] Hàn Thế Thành. *Micro-typographic extensions to the TEX typesetting system*. PhD thesis, Masaryk University Brno, October 2000.

[6] Allyn Jackson. From preprints to e-prints: The rise of electronic preprint servers in mathematics. *Notices of the American Mathematical Society*, 49(1):23–32, 2002.

[7] Donald E. Knuth. *The TEXbook*. Reading, Massachusetts: Addison-Wesley, 1984.

[8] Donald E. Knuth. *Metafont: The Program*. Reading, Massachusetts: Addison-Wesley, 1986.

[9] Donald E. Knuth. *The Metafontbook*. Reading, Massachusetts: Addison-Wesley, 1986.

[10] Leslie Lamport. *LATEX: A document preparation system: User's guide and reference*. Reading, Massachusetts: Addison-Wesley Professional, 1994.

[11] Microsoft. Windows Store. `http://windows.microsoft.com/en-us/windows-8/apps`. Accessed October 2013.

[12] MSDN. StorageFile.GetBasicPropertiesAsync. `http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.storagefile.getbasicpropertiesasync.aspx`. Accessed August 2013.

[13] Take-Yuki Nagao. Automatic recognition of theorem environments of mathematical papers in LATEX format. *Bulletin of Advanced Institute of Industrial Technology*, 7:81–87, 2013.

[14] Toshio Oshima, Yoshiki Otobe, and Kazunori Asayama. dviout — a DVI previewer for Windows. `http://www.ctan.org/pkg/dviout`. Accessed October 2013.

[15] Fabrice Popineau. windvi — MS-Windows DVI viewer. `http://www.ctan.org/pkg/windvi`. Accessed August 2013.

[16] Christian Schenk. Yet Another Previewer. `http://www.miktex.org`. Accessed October 2013.

[17] Paul Vojta. xdvi — A DVI previewer for the X Window System. `http://math.berkeley.edu/~vojta/xdvi.html`. Accessed October 2013.

◇ Antoine Bossard and Takeyuki Nagao
Big Data Laboratory
Advanced Institute of Industrial Technology
1-10-40 Higashiooi
Shinagawa-ku, 140-0011
Japan
`abossard (at) aiit dot ac dot jp`
`nagao-takeyuki (at) aiit dot ac dot jp`