Despite the dynamic nature of the Web, most people view a static snapshot. Search engines, browsers, and higher level end-user programming environments only support observing and manipulating a single point in time—the "now." To better support interactions with historical data, we created the Zoetrope system [cite] that provides a visual query language and environment for end-user manipulation of historical Web content. By supporting such interactions from *within the context* of the Now Web (i.e., through the "present" copy of a page), the Zoetrope user does not need to identify the location of historical content. For example, a user may place a lens—a visual marker—on the "current" lowest used book price on an Amazon page, and have the *historical* price automatically extracted and visualized over the history of the page (see Figure 1). While focused on access to the ephemeral Web, the design of Zoetrope has a number of implications to the design of any end-user programming environments in its ability to "debug" such programs on historical content and generate additional training data. Integrating the temporal dimension provides both new challenges as well as many new opportunities which we explore within the context of Zoetrope.

Zoetrope

Zoetrope functions as a complete system from crawling pages, storing content and indices, an internal dataflow language for representing queries on data, and a front end for "programming" queries and visualizing output. The main visual operator in Zoetrope is a lens. Lenses are rectangular objects drawn on any part of a Web page (see Figure 1). Each lens can be manipulated through a slider, giving user interactive access to historical content from "past" to "present" and providing the illusion of dynamic content from static information[1]. Data selected within the lens can then be visualized in a number of different ways: time series for numerical data, movies constructed from many static images, time lines, and so on. Although there are a number of different types of lenses to target different types of data, all are highly interactive and respond to the user's manipulation of the slider instantly. Interactivity is a key feature and distinction for Zoetrope driven by the fundamental observation that most applications that rely on screen-scraping style extractions will fail at some point due a drastic change in page structures. By allowing the Zoetrope user to *see* how their selection functions over time, corrections can be made at points in history where the extraction has failed.

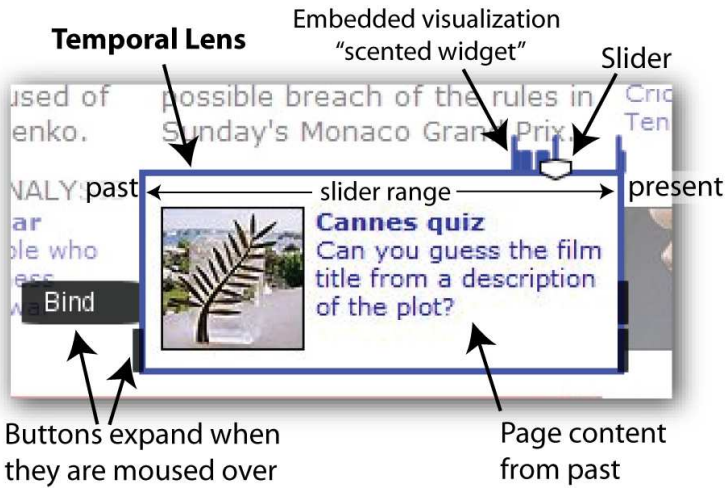

Figure 1: Anatomy of a lens

Additional lens features include filtering with keyword or other features and the binding of multiple lenses (on the same and different pages) to explore correlated data (e.g., gas prices on one page versus oil prices on another versus news stories mentioning the Middle East on yet a third).

As described in [cite], Zoetrope contributes:

[1] The Zoetrope system derives its name from the old mechanical Zoetropes. where a cylinder containing frames from an animated sequence were observed through vertical slits cut into the cylinder. When spun, the Zoetrope provided the illusion of motion to the observer fixing their gaze on the cylinder.

- a novel visual programming toolkit and a set of interactions for rapidly building and testing temporal Web queries,
- a semantics for temporal data streams,
- a set of recomposable operators to manipulate temporal data streams,
- indexing structures for fast processing and interaction with Web content over time,
- and a unique dataset collected by a new crawler design.

In this chapter we explore in detail the particular design choices for Zoetrope and how the notion of time can be employed in end-user programming environments regardless of their emphasis on past or present. These applications share a common goal with Zoetrope in their desire to function as well as possible into the *future*.

The Zoetrope Architecture

Zoetrope consists of a number of components that taken together provide a complete solution for collecting, storing, and querying historical versions of the same page. This architecture is illustrated in Figure 2.

Crawler

The Zoetrope crawler is a modified Firefox with two (also modified) plugins. The first, WebPageDump [cite], outputs the browser-internal Document Object Model (DOM) representation of the page. Capturing this representation is important as it is a) a compliant XML file that can be indexed in an XML database, b) a frozen version of the page that does not contain any Javascript and can thus be guaranteed to render the same way each time the page is loaded. The modified WebPageDump waits some period the Javascript on the page has either stopped executing or some waiting period has expired. Though sometimes imperfect, this allows any Javascript code that modifies the DOM to complete the process. In addition to the serialized DOM, the plugin stores a single CSS page specifying formatting and all objects (images, flash files, etc.) associated with the page. A second plugin, Screengrab! (www.screengrab.org), also produces a capture of whatever is being displayed in the browser. Based on these plugins, the crawler collects an accurate representation of the page as it looked at the time of retrieval (through an image) and sufficient data so that the page can be correctly re-rendered if necessary. Though strictly, only the un-rendered content is needed, by storing and caching the rendered content we are able to better support a number of the interactive features.

At present, the crawling infrastructure resides on a "headless" Linux server (i.e., one without a display system). Firefox instances, running in the background, collect nearly 1000 pages every hour. This is not a limit of the machine but rather the population of pages we have chosen to crawl. Each page is crawled once per hour at the fastest rate with a backoff protocol that slides this to once a day if the page remains unchanged. Although crawling speed can be greatly increased (further increasing version granularity), there are "politeness" limits that need to be respected. A machine would likely be able to crawl many times this number of pages, and

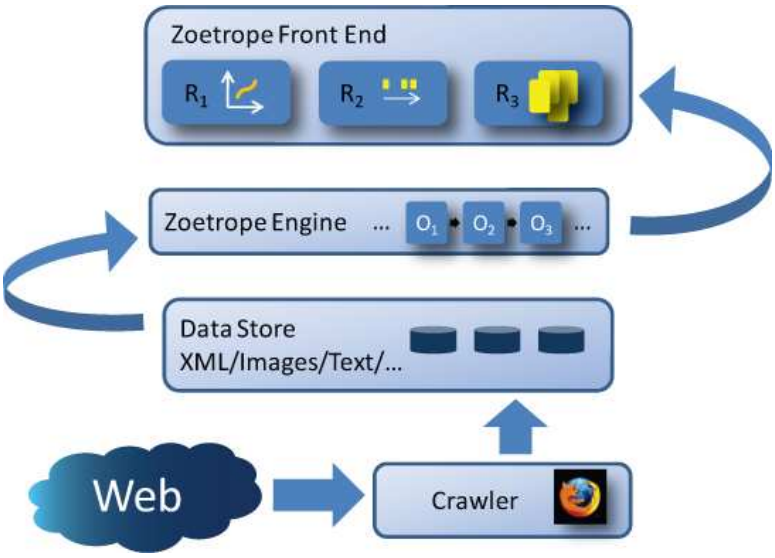

Figure 2: The Zoetrope Architecture

one could envision a service that collects page snapshots at different intervals. Furthermore, using a Firefox plugin has the further advantage that people can add to their own database as they browse the Web. The combination of personal visit archives and periodic snapshots might be sufficient for a number of applications.

Figure 3: A time series visualization displays the Harry Potter book sales, or Muggle Counter, over time.

Storage & Data

The DOM for each version of a page are loaded into an in-memory XML database. Saxon (www.saxonica.com), provides XPath query functionality and allows us to rapidly find matching elements or entire versions of pages. An in-memory index also tracks DOM elements by their rendered x and y coordinates. This structure allows us to quickly find which elements are clicked or selected. An early analysis of storage needs [cite], revealed that each incremental copy for a 5 week crawl (crawled once an hour) was 15% (2Kb average, 252 bytes median) the size of the original, compressed, copy of the page. This is encouraging from the perspective of managing many copies of the same page.

The Zoetrope Engine

Internally, Zoetrope is built on a simple dataflow architecture where *operators* act on a *content stream*. Conceptually, a content stream is a sequence of tuples (i.e. pairs), $<T_i, C_i>$, where C_i is a content item, such as a webpage or some piece of a page, and T_i is the time when that content was sampled from the Web. When a person creates a lens or some other visualization, Zoetrope generates a sequence of *operators*, which process the content stream. There are presently three main types of abstract operators which act on content streams in different ways:

- *Transform* operators modify the content payload of tuples. Each tuple is processed by this operator and one or more new operators are generated, replacing the processed tuple. For example, if the content item is an image, a transform operator may crop the image.
- *Filter* operators modify the content stream by removing or allowing a given tuple to pass through. For example, filter operator may only allow tuples with a data greater than some number or only tuples where the content contains a certain string.
- Finally, *render* operators interface between the engine and GUI by rendering some visualization of the content stream that contains one or many tuples. A time series renderer, for example, will depict the fluctuations of numerical values over time.

The Zoetrope Interface

The primary Zoetrope interface is a zoomable canvas based on Piccolo [cite] within which the Zoetrope user can explore past versions of any number of Web pages. Though most Zoetrope features can be implemented in modern Web browsers, interactivity is still technically challenging (though this will likely become less of a problem as we new browser technologies are released). Zoetrope lenses are drawn directly into this interface as well as any visualizations, providing a single workspace for exploration. Although Zoetrope displays the rendered webpage as an image, the system maintains the interactivity of the live webpage.

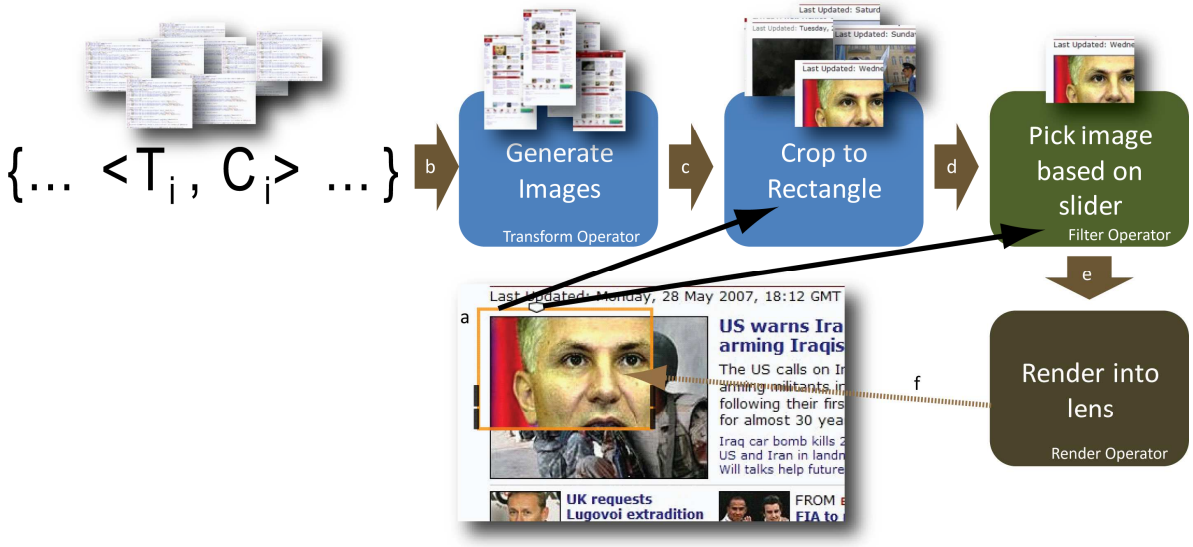

Figure 4: A Visual lens in action. The user specifies a visual lens (a) on top of the page. This causes Zoetrope to take all versions of the document in a content stream and push those to a transform operator (b) that renders (or looks up) how the page looked at every time step. The tuples, which now contain rendered pages images, is push to a second transform which crops the images to the dimensions specified by the original placed lens. This steam is then pushed to a filter operator this is parameterized to the slider state and which picks the single tuple closest to that time. Finally, a render operator takes that single tuple and displays it inside the original lens.

Clicking on a hyperlink opens a browser for the present version of the hyperlink target (or the historical version corresponding to the slider selection, if it exists within Zoetrope).

Figure 3 displays such a workspace where the user has loaded the Amazon homepage, selected a element on the page (the number of pre-sales of Harry Potter) and a then visualized those presales in a time series. Zoetrope supports binding between multiple lenses or between lenses and visualizations. By dragging a line from one lens' bind button to another's a user explicitly connects the two so that motion between one

Lenses come in three distinct flavors: visual, structural, and textual. The different variations are intended to track content on the page based on its stability. For example, content that is stable in rendered coordinates can be tracked with a visual lens. On the other hand if the content being extracted by the lens is visually in a different place, but is always retrievable by the same path through the DOM hierarchy.

Lenses are in part inspired by the Video Cube [cite] and Magic Lenses work [cite]. The former allows video stream frames to be layered and "sliced" to find an abstraction of the video. In the latter, a magic lens is a widget that can be placed directly on a document to illuminate the underlying representation while maintaining the visual context of the document. Architecturally, a number of database visualization systems are related to the stream and operator design in Zoetrope. Particularly, DEVise [cite] was originally constructed to operate on streams of data that required visualization. Similarly, Polaris [cite] operates to visualize relational data. In both, the operators provided by the system are primarily targeted at the rendering step (deciding how graphs, charts, and other visualizations should be constructed).

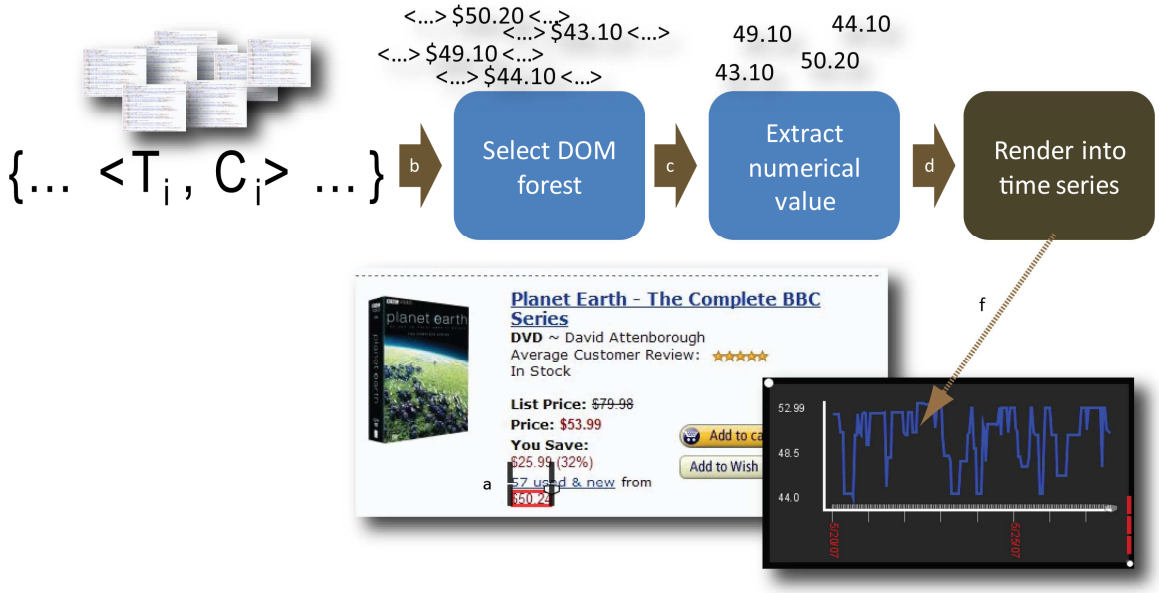

Figure 5: A structural lens and time series visualization. The user specifies a structural lens (a), selecting the used price of the DVD. Zoetrope takes all versions of the document in a content stream and pushes them to a transform operator (b) that selects a portion of the document corresponding to the selection (i.e., the XPath). A second transform operator (c) strips the text and extracts the numerical value which is pushed to a renderer (d) and finally a visualization (f).

Zoetrope Lenses

A lens allows a person to select some content and track it over time (i.e., the *temporal extraction*). Although there are various flavors of lenses, their creation and use is nearly identical. A person creates a lens simply by drawing a rectangular area on the webpage surface. In the underlying semantics of the Zoetrope system, the creation of a lens produces a parametrized transform operator that acts on the original page content stream, an optional filter (or set of filters) that processes the transformed stream, and a renderer that displays the historical data in the context of the original page. The specific selections of transforms and filters depends on the lens type.

Visual Lenses

The simplest Zoetrope lens is the visual lens. To create this type of lens, a person specifies a region on the original page (e.g., a portion of the BBC homepage as in Figure 4). The specification produces a lens with a slider. The slider is parameterized on the width of the lens and the range of data being displayed. As the slider moves, the lens renders the corresponding data. Figure 4 illustrates how a visual lens is implemented using the internal Zoetrope operators. When created, each version of the page becomes a tuple in the content stream. Each of these tuples is then rendered (or looked up if cached), cropped, filtered, and then rendered inside the lens.

Structural Lenses

Not all webpages possess sufficient stability for a visual lens. Slight shifts in rendering or more significant movement of elements can cause distracting jumps when a person moves the lens slider. To counter this

effect, and to allow for more precise selections, Zoetrope provides structural lenses. Structural lenses are created in the same way as visual lenses, by drawing a rectangle around an area of interest, but they track selected HTML content independent of visual position. Specifically, when created, a structural lens defines a DOM forest within the structure of the page. This is specified through an XPath expression [cite] that can be used to select a sub-tree of the structure. For example, in Figure 5, in order to track price over time in the page, the user selects the price element using a structural lens. The structural lens then parameterizes a transform operation which is able to pull out the element containing the price information over many past versions of the page.

Textual Lenses

Visual and structural lenses are dependent on certain types of webpage stability. A visual lens relies on stability of the rendering, whereas a structural lens takes advantage of structural stability. Both are reasonable in many scenarios, but it is also worth considering selections based on unstable or semi-stable content. For example, consider tracking a specific team in a list of sports teams that is ordered by some changing value, such as rank (see Figure 6). As teams win and lose, the team of interest will move up and down the list. Specifying a rectangle at (100,400), or the fourth row in the list, will not work when the team moves from this position. To address this type of selection, we introduce the notion of a textual lens, which tracks a textual selection regardless of where the text is located on the page. A textual lens can track exactly the same string (e.g., a blog story) or approximately the same string (e.g., a sports team name with a score, where the score changes from time to time).

In its most general form, a textual lens tracks arbitrary text regardless of where it appears on the page, which DOM elements contain the text, and the size of those elements. This generalization is unfortunately too computationally intensive, even in scenarios where the text is unchanging, and the problem becomes intractable for an interactive system. To make our textual lenses interactive, we restrict the search space by making use of the document structure. Textual lenses often track items that appear in tables, lists, or structurally similar sub-trees (e.g., posts in a blog all have similar forms). We take advantage of this structural similarity to only search among DOM elements that have similar tree structure to the original selection. From this initial set of possibilities, Zoetrope will compare the text in the original selection to the possible matches, picking the most likely one. The comparison is currently done through the Dice coefficient, which calculates the overlap in text tokens between two pieces of text (i.e., $SIM(A,B)=2*|A \cap B|/(|A|+|B|)$, where A and B are sets of words).

Applying Filters to Lenses

Given the large volume of data encoded in a content stream, it is natural to want to focus on specific information of interest. Zoetrope uses filters to provide this capability. We have already seen a few different kinds of filter operations, but it is worth considering additional types:

- Filtering on Time: One may wish to see the state of one or more streams at a specific time or frequency (e.g., 6pm each day).

- Filtering on a Keyword: The selection condition may also refer to C_i, the content half of the tuple $<T_i,C_i>$. If C_i

Figure 6: A textual lens can track content regardless of where it appears on the page, such as the Toronto Blue Jays, which over time shift in the ordered list above.

contains text, then keyword queries may apply. For example, one might only be interested in headlines that contain the word "Ukraine."

- Filtering on Amounts: One may also select content using an inequality and threshold (e.g., >k). If the content is numeric and the inequality is satisfied, then the tuple is kept; otherwise it is filtered. Similarly, one can select the maximum or minimum tuple in a numeric stream.

- Duplicate Elimination: It may also be useful to select only those tuples whose content is distinct from content seen earlier in the stream.

- Compound Filters: Logical operations (conjunction, disjunction, negation, etc.) may be used to compose more complex selection criteria.

- Trigger Filters: An especially powerful filter results when one stream is filtered according to the results of another stream's filter. For example, Ed can filter the traffic page using a conjunction of the 6pm time constraint and a trigger on the ESPN page for the keyword "home game." We will return to this when considering lens binding.

Because filtering is useful for many tasks, it is provided as an option whenever a visual, structural, or textual lens is applied. When selecting a region with filtering enabled, a lens is created based on the underlying selection and a popup window asks for a constraint to use in the filter, such as a word or phrase. Other appropriate constraints include maximum, minimum, and comparison operators.

Filtering is visually depicted with a scented widget [cite] which is displayed as a small embedded bar graph (Figure 1). The bar graph is displayed above the slider, indicating the location in time of the matching tuples. As a person moves the slider, the slider snaps to the bars, which act like slider ticks. Note that the bars need not be all of the same height and may reflect different information. A tall bar can indicate the appearance of new content that matches a filter, and a short bar can indicate content that appears previously but still matches the filter.

Binding Lenses

People are often interested in multiple parts of a page or parts of multiple pages, as they may be comparing and contrasting different information (e.g., what does traffic look like on game days?). Zoetrope flexibly allows for the simultaneous use of multiple lenses. Lenses can act independently or be bound together interactively into a synchronized bind group. Sliders within a group are linked together, causing them all to move and simultaneously update their corresponding lens.

People may bind lenses for different reasons. For example, to check traffic at 6pm on home game days, Ed can bind a lens for traffic maps at 6pm with a lens for home games from his favorite baseball site. Each lens in a bind group constrains its matching tuples to only include versions allowed by all other lenses in the group. Recall that this is achieved through a trigger filter. Each lens can add a new trigger filter parameterized to the time intervals that are valid according to other members of the bind group. Only tuples that satisfy all trigger filters are allowed. Thus, the resulting stream shows traffic data at 6pm only on days for which there are home baseball games.

Lenses can also be bound disjunctively. For example, one may want to find when book A's price is less than $25 or when book B's price is less than $30 (i.e., one of the two books has dropped in price). Zoetrope supports this type of bind, which is currently obtained by holding the shift key while performing the bind operation. However, this operation creates an interesting twist as it causes data to be un-filtered. When binding two lenses in this way, filter operators can be thought of as operating in parallel rather than serially. A tuple passes if it matches any filter.

Stacking Lenses

In addition to binding, Zoetrope also supports the stacking of lenses. For example, consider a person who creates one lens on a weather page, filtering for "clear" weather, and would like to further apply a filter that restricts the selection to between 6 and 7pm daily. Explicitly drawing one lens over the other and then binding them is visually unappealing and does not take advantage of the underlying semantics of the language. Instead, we introduce the notion of lens stacking. The toolbar in the Zoetrope window, which allows people to select the type of the lens, can also be used in a specialized binding operation which we call stacking. By dragging a lens selection from this toolbar to the bind button of the lens, a person indicates that they would like to further filter the existing lens. The original lens is replaced, and a new combined lens is generated, which takes the transform and filter from the original selection and augments it with additional transforms and filters. This new lens satisfies both the selection and constraints of the original lens as well as the new one. Furthermore, because some filters and transforms are commutative, stacking provides the opportunity to reorder the internal operations to optimize the processing of tuples.

Finally, we consider the partial stacking of lenses where a person wants to make a sub-selection from an existing lens. For example, a person may apply a textual lens that tracks a specific team in the ranking. The textual lens will track the team no matter where they are in the ranking, but the person would further like to pull out the wins for that team at various time points. Thus, they may create a second structural lens that consumes the selection of the textual lens and selects the wins. While most lenses can be easily stacked without modification, lenses that stack on top of textual lenses require a slight modification to utilize relative information (paths or locations). This subtle modification is necessary because the textual lens selects information that is not in a fixed location in either the x,y space or the DOM tree. Because the textual selection is variable, the structural lens must utilize a path relative to the selection rather than an absolute path.

Lenses Design Considerations

The data Zoetrope manipulates irregularly changes, shifts, and vanishes, and thus our design had to address and accommodate this unpredictable behavior.

Windows in Time

It is worth briefly considering the design decision to display past content in the context of a webpage. In our design, the historical information overwrites the part of the page where the lens is located. An alternative would be to respond to slider motion by loading the entire page for a specific time and offsetting that page so that the selected area is underneath the lens (similar to automatically panning the lens "camera" at each step). Although we can enable this mode, we found two main reasons why this type of movement was unappealing.

First, the motion and replacement of the entire page (rather than a small area) was highly distracting. Rather than visually tracking changes in a small area of interest, a person must track changes across the entire screen. Second, it is not clear what should happen when a person creates multiple lenses on the same page. As described above, multiple lenses represent different selections within a page. These selections may not always have the same visual relationship (for example, being 10 pixels apart in one time and 500 pixels apart in another) and may be set to different times. Given these constraints, it may not be possible, without distortion, to offset the underlying page to a position that works for all time periods or combinations of lenses.

Lenses for Debugging

Figure 7: This timeline visualization shows the duration and frequency of news articles on the cbc.ca website.

The robustness of end-user programs on the Web is a recognized problem (e.g., [cite]) as page templates will likely change at some point [cite] causing the program to fail. Because users can interactively detect failures in the extraction, they may refine their selection, adding or removing restrictions, improving filtering conditions or creating another lens for the "failed" interval. The interesting side-effect of the Zoetrope interaction techniques is that although the user may not know how the extraction will fail in the future, they may nonetheless improve the robustness of their extractions by observing failures in the past.

Users, as constructors of these programs and extractions, have a unique ability to determine if a program has succeeded. By simulating the program—whether an extraction, or something more sophisticated—on historical data, the programmer can identify failure conditions that may recur in the future and increase the robustness of his programs. User specified failures also give the system an opportunity to automatically adjust its behavior. For example, by identifying a failure on past data, the user is providing examples to the system, which can be used to train a failure detection mechanism. Supplying a "fix" to this failure can help train exception handling mechanisms that will allow a program to continue working or adapting despite future changes to page structure.

Zoetrope Visualizations

Lenses enable viewing of Web content from different moments in time, but this exploration is likely just the first part of satisfying an information need. For example, a book's cost at specific points in time is interesting, but a person may also want to graph the price over time, calculate averages, or test variability. To facilitate this type of analysis, we have created a number of renderers that visualize or otherwise represent selected data. Visualizations, like lenses, create a sequence of transforms, filters, and renderers to display results. Although visualizations can exist independently of a lens, a lens typically defines the data displayed in the visualization. Lenses can thus also be used as a prototyping tool for testing selections and aggregations.

The transforms, filters, and processed streams generated by the lens can be directed to visualization rendering components that implement the visualization itself. For example, in a typical workflow one might place a (potentially filtered) lens on a book price, move the slider to test the selection, and click on the visualization button to graph the price over time. Internally, the visualization step reuses the transform module of the lens and connects it to a time series renderer. Clearly, many temporal-visualizations are possible [cite] and could be implemented in Zoetrope or externally. As we describe below, a number of default renderers provide visualization alternatives in the current implementation.

Timelines and Movies

The simplest Zoetrope visualization type is the timeline (see Figure 7), which displays extracted images and data linearly on a temporal axis. This visualization allows, for example, viewing weather patterns over the course of a year, headline images in stories that mention Iraq, or unique articles about a favorite sports team

(all ordered by time). As before, the rendered images visualized in the timeline are live and a person can click on any of the links. Double clicking on any image in the visualization synchronizes the page (or pages) to the same time, allowing a person to see other information that appeared on the page at a particular time. This visualization can also eliminate duplicates and display a line next to each image depicting its duration. This type of display shows when new content appears and how long it stays on a page (e.g., a story in the news, a price at a store) To prevent the timeline from running indefinitely to the right, the visualization can fold the line into a grid with each row denoting activity over a day (or other interval).

The timeline visualization gives an instant sense of everything that has happened over some period. However, other examples are best served by cycling through the cropped images to produce an animated movie. Although this is equivalent to simply pulling the slider through time, a movie visualization automates and regulates transitions and looping while the visualization cycles through the images. For example, a static USGS earthquake map can be transformed into an animation of earthquakes over time, helping to pinpoint significant events.

Clustering

Our timeline visualization is a simple example of a grouping visualization, where extractions are grouped by some variable (time in this case). However, other more complex groupings are also possible. Clustering visualizations group the extracted clips using an external variable derived from another stream. For example, a clustering visualization can merge data from two different lenses, using one lens to specify the grouping criteria while the other lens provides the data. For example, in a cluster visualization of traffic and weather data we create a row for every weather condition (e.g., sunny, clear, rain), and every instance from the traffic selection is placed in the appropriate row depending on the weather condition at the time of the clipping. If it was rainy at 8:15pm, for example, the traffic map from 8:15pm is assigned to the rainy group.

Time Series

A variety of interesting temporal data is numerical in nature, such as prices, temperatures, sports statistics, and polling numbers. Much of this data is tracked over time; however, in many situations it is difficult or impossible to find one table or chart that includes all values of interest. Zoetrope automatically extracts numerical values from selections of numerical data and visualizes them as a time series. Figure 10 shows a visualization of the number of Harry Potter book sales over time. The slider on the x-axis of the time series visualization is synchronized with the slider in the original lens selection. When the person moves the lens slider, a line moves on the time series visualization (and vice versa).

Exporting Temporal Data

Zoetrope offers many useful visualizations, but was also designed for extensibility. Data extracted using Zoetrope lenses is therefore also usable outside of Zoetrope. Systems such as Swivel (www.swivel.com) or Many Eyes [cite] excel in analysis and social interactions around data, but are not focused on helping people find the data in the first place. Zoetrope is able to generate a temporal, data-centric view of different websites to meet this need. To export data outside of Zoetrope, we created a Google Spreadsheet "visualization" that sends the lens-selected values to the external Google Spreadsheet system. When a person selects this option, appropriate date and time columns are generated along with the content present at that time interval (either strings or numerical data). This scheme greatly expands the capabilities of Zoetrope by allowing people to leverage external visualizations, integrate with Web mashups, or perform more complex analyses.

Conclusions

The World Wide Web is generally treated by users and system designers as a single snapshot in time. This limited view ignores the importance of temporal data and the potential benefits of maintaining a Web history. By considering the past, end-user programming environments can be enhanced for the future. Zoetrope represents one possible solution in the space, concentrating specifically on how a user might analyze and visualize data from the temporal Web from the familiar context of the Now. By maintaining historical copies of the Web, Zoetrope provides users with the ability to simulate programs on historical data. This has implications for other end-user programming tools for the Web which similarly depend on structural stability and benefit from allowing a user to test and validate their selections.

[citations – I am travelling and have no access to my EndNote, citations will be added shortly]