

# The Design and Implementation of ViPER

David Mihalcik and David Doermann

Institute for Advanced Computer Studies

University of Maryland, College Park

([davidm@umiacs.umd.edu](mailto:davidm@umiacs.umd.edu))

*A video sequence may contain any number of persons, objects and activities. ViPER-GT is a tool for annotating a video with detailed spatial and temporal information about its contents. This paper is for people who wish to extend it for their own needs, or curious users who wish to understand some of its design choices. It presents information about ViPER-GT's predecessors (and antecedents), information about its design and implementation and some use cases.*

# Table of Contents

<b>1. Introduction .....</b>	<b>5</b>
<b>2. Related Work.....</b>	<b>6</b>
<b>2.1. Video Annotation Data Models .....</b>	<b>6</b>
<b>2.2. Video Annotation Systems .....</b>	<b>7</b>
<b>2.3. Closed-Loop Video Performance Analysis Systems.....</b>	<b>10</b>
<b>3. XGTF: The ViPER Data Model.....</b>	<b>10</b>
<b>4. User Interaction Design.....</b>	<b>12</b>
<b>4.1. Introduction to ViPER-GT's User Interface.....</b>	<b>12</b>
<b>4.2. Modality .....</b>	<b>12</b>
<b>4.3. The Video Canvas.....</b>	<b>14</b>
4.3.1. Design .....	14
4.3.2. Implementation.....	14
<b>4.4. The Score.....</b>	<b>15</b>
4.4.1. Design .....	15
4.4.2. Implementation.....	16
<b>4.5. The Spreadsheet.....</b>	<b>16</b>
4.5.1. Design .....	16
4.5.2. Implementation.....	17
<b>4.6. The Schema Editor .....</b>	<b>17</b>
4.6.1. Design .....	17
4.6.2. Implementation.....	18
<b>4.7. The Remote Toolbar .....</b>	<b>18</b>
4.7.1. Design .....	18

4.7.2.	Implementation.....	19
<b>4.8.</b>	<b>Notes about Miscellaneous User Functions .....</b>	<b>19</b>
4.8.1.	Handling Media Source File Selection .....	19
4.8.2.	Managing Undo and Redo .....	20
4.8.3.	Adding Scripts .....	20
4.8.4.	Handling Files .....	20
4.8.5.	Interpolation and Propagation .....	21
<b>5.</b>	<b>Use Cases .....</b>	<b>21</b>
<b>5.1.</b>	<b>Text Detection .....</b>	<b>21</b>
<b>5.2.</b>	<b>Person Tracking.....</b>	<b>22</b>
<b>5.3.</b>	<b>Event Detection on the Tarmac.....</b>	<b>23</b>
<b>6.</b>	<b>Extending ViPER-GT .....</b>	<b>25</b>
<b>6.1.</b>	<b>The ViPER File Format, XGTF .....</b>	<b>25</b>
6.1.1.	The FILE Information Descriptor .....	25
<b>6.2.</b>	<b>Accessing the Data through the ViPER API .....</b>	<b>25</b>
6.2.1.	Creating an Instance of ViperData .....	26
6.2.2.	Reading from and Saving from XGTF .....	26
6.2.3.	Transactions.....	27
<b>6.3.</b>	<b>Adding New Data Types .....</b>	<b>27</b>
<b>6.4.</b>	<b>Adding New Interaction Methods: Wizards and Widgets .....</b>	<b>28</b>
6.4.1.	An Example: Labeled Text Zones.....	28
6.4.1.1.	The Zone Editor Panel .....	29
6.4.1.2.	Adding Color to the Types .....	29
6.4.1.3.	Displaying Connections.....	30

<b>7. Future Work.....</b>	<b>30</b>
<b>8. Appendix: The Limn3 Application Loader.....</b>	<b>31</b>
<b>8.1. System Management .....</b>	<b>32</b>
8.1.1. Bootstrapping .....	32
8.1.2. The Triplestore .....	34
8.1.3. Declaring a Bean .....	34
8.1.4. Declaring a Menu Item or Group .....	34
<b>9. Appendix: Glossary.....</b>	<b>36</b>
<b>10. Bibliography.....</b>	<b>37</b>

# 1. Introduction

Video understanding algorithms attempt to extract information automatically—allowing a machine to annotate a video quickly, a task often prohibitively time consuming for humans. To evaluate a video analysis algorithm, or compare a set of algorithms, it is necessary to define a performance evaluation methodology. As many books and papers describe different evaluation methods, we decided to develop a general framework for to support many types of evaluation. A central idea common in information extraction evaluation is a comparison between the extracted data and some ideal version of ‘Truth’. [7]

In some subfields of computer vision, like document processing, it is possible to generate test data automatically from the source. Metadata is rarely constructed alongside videos, leaving annotation and metadata to be added later. In order to ensure researchers may repeat and verify evaluations, it is important to distribute trustworthy ground truth metadata. Metadata browsers and editors allow qualitative verification of the ground truth. ViPER-GT is a tool for creating and editing video metadata.<sup>1</sup>

The major design goal of the ‘ViPER Ground Truth Authoring System’ is a general-purpose and unbiased annotation system that supports creating data useful for evaluation. Video information extraction is a large field, from medium-level vision tasks, like object detection, to high-level media understanding, like event analysis. The annotation tool can be configured for supporting such diverse tasks as text detection, faces identification, detecting personal interactions, finding object exchanges or labeling terrorist activities. The goal of ‘General-Purpose Annotation’ really means general enough to cover all considered uses. The first use cases for the ViPER Toolkit came directly from ongoing research in LAMP. This included a text detection and extraction system, face detection and identification based on landmark points, and a rule based event detection system. This paper goes through the major aspects of ViPER-GT’s design and implementation, using these use cases and goals as considerations and examples.

The goal of creating a general-purpose annotation system led ViPER-GT to avoid case-specific optimization and computer vision assistance. These are two of the most requested features. Case specific optimization is possible through plug-ins and scripts. Similarly, machine vision can be added as extensions. For more information, see the Extending ViPER-GT section.

With these goals in mind, the original authors of ViPER-GT, Felix Suhenko and Charles Lin, developed a complete annotation system. Written for Java 1.1, then 1.2, it supported loading

---

<sup>1</sup>*ViPER-PE, a companion program to ViPER-GT, performs comparison and performance evaluation of video metadata. It is a command line tool; its complex configuration system seems to have prevented its adoption as a standard tool by the community.*

sequences of images and a large number of the features described in this paper. However, it was implemented without a solid separation of components, and extending it to support new data types, views and modes of interaction proved difficult. ViPER-GT 4.0 is a complete change in architecture, adding a unified data model and support for such features as Undo, loading multiple videos, and support for scripts and plug-ins.

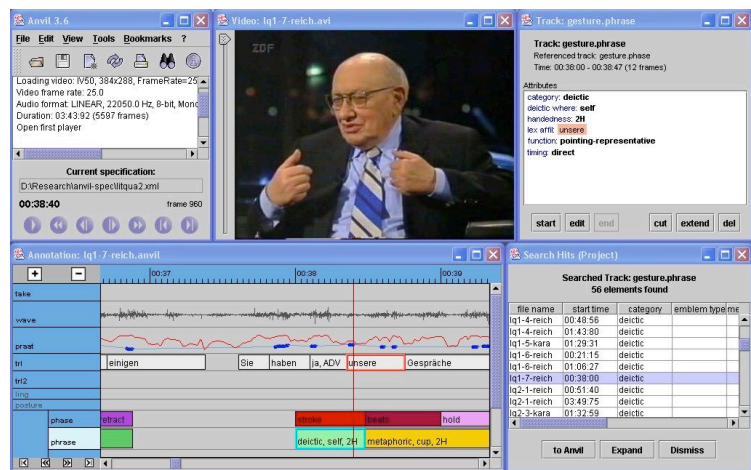
## 2. Related Work

The ViPER project does not exist in a vacuum. Whenever possible, it looks to the existence of standards in the field and other tools for support. This could mean adding import and export filters, or simply incorporating features expected in annotation systems. This section examines some tools that offer video annotation functionality that is similar, or less similar, to ViPER-GT.

### 2.1. Video Annotation Data Models

The ViPER toolkit uses a simple data format, XGTF, to describe video content, described in the appropriate section below. Some systems present even more simplified, and usually task specific, data models, such as KLV. Others provide more general data models, such as the various available OWL media description ontologies. Orthogonal to data model is file storage and arrangement, which may be stored in separate, data oriented files, like RDF+XML, or time-oriented in a data stream with the video, like MPEG-7.

The simpler, domain specific approaches have their benefits. KLV (Key/Length/Value) is easy to parse, and its closed world of committee-defined annotations make complete implementations within the grasp of most developers. ViPER, whose primary users participate in Video TREC and ARDA's VACE program, could benefit from such machinery in its definitions. However, ViPER is almost as simple, and its extensibility allows different groups to design their own extensions to agreed-upon schemata. KLV provides a similar mechanism: systems that do not recognize a key simply pass over the value. Since each file contains its schema, XGTF provides allows systems to display and edit unrecognized keys that have known types. [5]



**Figure 0: Multimodal annotation with Anvil. Here, the interface is used to mark up gestures.**

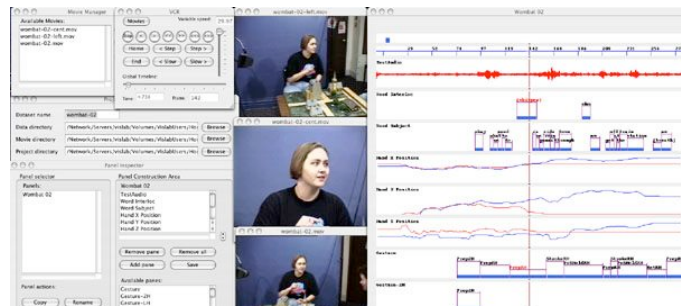
The W3's OWL standard provides a standard for Description Logics that is useful for generic data storage and reasoning. Unlike most Description Logic formats, OWL unifies the A-Box, which contains class definitions, and the T-Box, which contains instances. For information extraction evaluation, the elements of the A-Box, such as possible event and object types, are usually fixed. In event detection evaluation, the specific type of event being found is important, not the general event class that was detected; the DL inference features of OWL are not essential. Recently, some progress has been made in evaluation based on multi-class or hierarchical set counting. These methods define correctness as how the collection of sets the item is found with within, per its hierarchy. Since it is more accepted, and provides all the features required, OWL should eventually replace the XGTF, possibly through an XGTF ontology. [15]

MPEG-7 provides a third path, including both a schema language and a comprehensive committee-defined set of default schemata. The schema language is an extension to the W3's XML schema definition language. This means any valid XML document may be interpreted as an MPEG-7 document. J. Hunter has been doing some work to convert MPEG-7 data into OWL; this may give MPEG-7 the logical grounding required to make general-purpose annotation possible. For now, MPEG-7 tools will continue to support fixed schemata, with full schema support still far off. [1, 2]

## 2.2. Video Annotation Systems

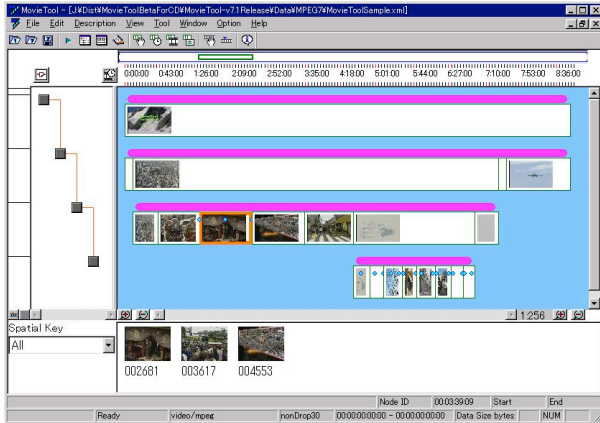
Anvil presents one of the most complete video annotation user interfaces. Anvil is useful for event and other time-oriented annotations; its original purpose was to add gesture annotations to output from the speech-annotation toolkit Praat. With its focused development goals, the software provides a tight design to an active user base. It includes its own scripting language, which uses sockets to support generic extensions. A third party plug-in provides spatial annotation. Anvil has a well-developed music score visualization, which presents mixed time series data elegantly. In addition, it supports audio annotation, mostly to edit existing Praat files. [6, 16]

PhotoStuff is a proof-of-concept system for annotating still images off the web. It is very generic and produces valid OWL data. Unfortunately, it does not present a very effective annotation interface. To annotate an image, the user draws shapes on the image then attaches the shapes to OWL resources. The system generates a custom form for describing the resource, using its declared class information from the ontology. [8]



**Figure 0: MacVisSTA is a multimodal annotation tool for Mac OS X**

VideoAnnEx, IBM Research's tool for annotating shots and segments of video, provides



**Figure 0: Ricoh MovieTool**

annotation of video with MPEG-7 and free text. It has tools for automatically segmenting video, picking key frames, region annotation of key frames, and annotation learning, which uses content analysis to predict shot keywords from existing keywords in the video. Although a closed-source windows application, it is one of the few free video annotation applications to use MPEG-7 and automatic, content-based annotation assistance. [9]

Heavily influenced by Anvil, NOMOS, a tool out of Stanford, uses a generic ontology like PhotoStuff and a timeline interface like

Anvil. It is still new, but has a more advanced integration of OWL's description logic than PhotoStuff. It supports multimodal annotation with a score view, but it is relatively new and still far from achieving Anvil's user experience. [10]

Vannotea supports collaborative editing of video metadata. It provides similar annotation power to VideoAnnEx, but it has a more user-centered interface that displays more of the video at once and more information about it. [11]

VisSTA, developed by Francis Quek and others at UIC, Wright State and VT, provides access multiple, synchronized video streams. Like Anvil, its focus is on multimodal annotation of human communication; here, the focus is on meeting data filmed from multiple viewpoints. The recent editions of the software are available for Mac OS X, called MacVisSTA. It is open source and could prove a useful code base for any number of video annotation and browsing systems. [12]



Ricoh's Movie Tool is a hierarchical scene description editor based on MPEG-7. A movie can be divided into labeled titles, chapters, scenes and shots. It is unavailable online, and development seems to have ceased. VideoAnnEx could adopt its simple interface to MPEG-7.

<b>Toolkit</b>	<b>License / Availability</b>	<b>Language</b>	<b>Platform</b>	<b>Supported Media</b>	<b>File Format</b>	<b>Spatial?</b>
<b>ViPER-GT</b>	GPL	Java	Java	MPEG-1,2, JMF, QuickTime	Custom XML	Yes
<b>Anvil</b>	Educational	C	Cross-Platform	AVI/MOV	Custom	3 <sup>rd</sup> Party
<b>PhotoStuff</b>	MPL	Java	Java	Still Images	OWL	Yes
<b>VideoAnnEx</b>	Educational	C	Windows	Win Media	MPEG-7 (free text)	On keyframes
<b>NOMOS</b>	Freeware	Java	Java	JMF	OWL	Not yet
<b>Vannotea</b>	Unavailable			MPEG-1,2; still images	RDF (Annotea)	Limited
<b>MacVisSTA</b>	GPL	Objective-C	Mac OS X	QuickTime	?	Limited
<b>Ricoh MovieTool</b>	Unavailable			MPEG	MPEG-7, (scene annotation)	?

**Figure 1: Comparison of Annotation Tools**

[13]

ViPER-GT is the only video tool to focus on spatial annotation. While some provide points or bounding boxes, or, in the case of PhotoStuff, arbitrary polygons, ViPER-GT is alone in its attempt to provide advanced marking up of spatial data on every frame. This can be an error-prone and time-consuming task; the other tools are missing features equivalent to 'Display with Respect To,' 'Display When Valid,' item locking, and interpolation.

Several of the tools provide better event editing than ViPER-GT. In addition, ViPER-GT does not support audio annotation; in fact, the audio stream is not decoded. In general, older and more specialized toolkits offer the most mature and complete set of timesaving tools, while the newer or more general annotation software presents information in a disorganized manner that often makes simple annotation difficult.

### 2.3. Closed-Loop Video Performance Analysis Systems

It has generally fallen out of fashion to present closed-loop media annotation and evaluation systems. Conferences like VideoTREC enforce separate evaluation, while the rise of interoperable standards, like MPEG, OWL and even simple XML, make such systems less necessary. Though a few closed-loop systems still exist for specialized vision research projects, there do not appear to be any undergoing current development. Even Informedia seems stagnant. [18] The ViPER toolkit presents a data format, XGTF, that, while not standard, is easily adaptable to existing standards, to avoid lock-in. However, these systems often offered better browsing and data maintenance features, as well as providing a uniform way of executing a set of tests and comparing different algorithms or parameters, but these features are better left to specialized software.

### 3. XGTF: The ViPER Data Model

The ViPER data model, which is stored in XGTF files, is essentially a temporally qualified relational model. An XGTF file consists of a set of ‘Descriptors’, objects temporally qualified for a specified media file, and the schema that describes them. Each Descriptor has a set of ‘Attributes’, which take on concrete data or links to other descriptors. These correspond to rows (Descriptors) and columns (Attributes) in an SQL database. Usually, a Descriptor describes a unique object or event, with its Attributes describing aspects of it.

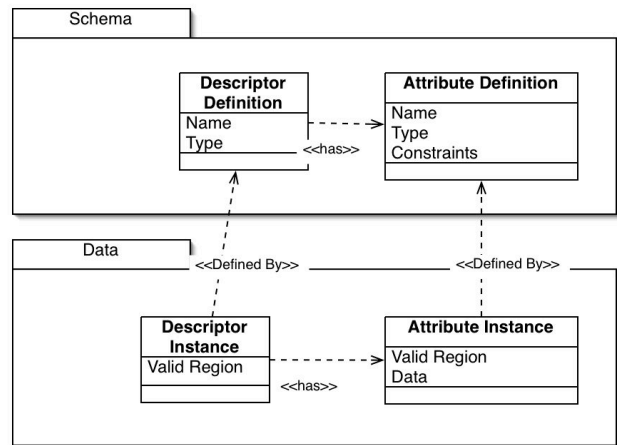


Figure 1: ViPER Data Model

The schema describes a variety of constraints on the Descriptors and their Attributes. Each Descriptors class can be specified as ‘Object’ (no restrictions), ‘Content’ (only one at a time), or ‘File’ (only one per file). For example, a user could define a ‘Person’ Object Descriptor class, a ‘Cut’ Content Descriptor class, and an ‘Author Information’ File Descriptor class. Each Attribute definition includes the data type, such as String, Bounding Box or Enumeration. Additionally, the schema labels Attributes either static (one value per instance) or dynamic (temporally qualified) and specifies a default value.

Data types included with the initial system include a String type, numeric types,

Relations and several spatial types (Bounding Boxes, Points, etc.). The current selection of field types reflects the ViPER toolkit’s main use: evaluating algorithms for detection, recognition and

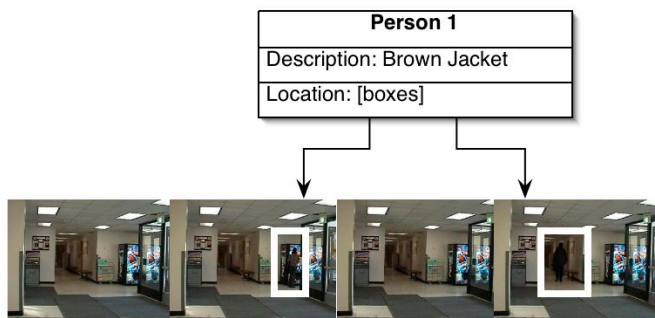


Figure 2: Person 1

tracking. Later additions came upon user request: an Enumeration type, an Omega shape for quickly annotating the head and shoulder of people in a video, and a Text Line type, which adds offsets for word breaks to the included Oriented Box type. While each additional shape could have been expressed using the base viper data set, some domain knowledge allows a more appropriate tool. For example, a String or Integer could easily replace the Enumeration, but the Enumeration allows for better display in the timeline and a pull-down in the spreadsheet view.

The relational model is familiar to most developers, simple to parse and store, and maps simply to most standard evaluation metrics. It is easy to explain, unlike MPEG-7, which requires several 300-page documents.<sup>2</sup> Most importantly, it is usually a simple task to write a script to extract any relevant data description languages. The ViPER toolkit web page links to XSLT scripts to extract all data from Informedia II files and from the 2000 Video TREC MPEG-7 subset, and the ViPER-Full distribution includes several short Perl scripts that convert from task-specific annotation into ViPER, and back again.

The ViPER API provides programmatic access to ViPER data files. There is currently only a Java implementation. It presents a DOM-inspired view of the data, with support for querying based on time and content, transactional editing and change event notification. For more information, refer to the Extending ViPER-GT section below, or the generated JavaDoc and example applications listed on the ViPER toolkit web page. [19]

---

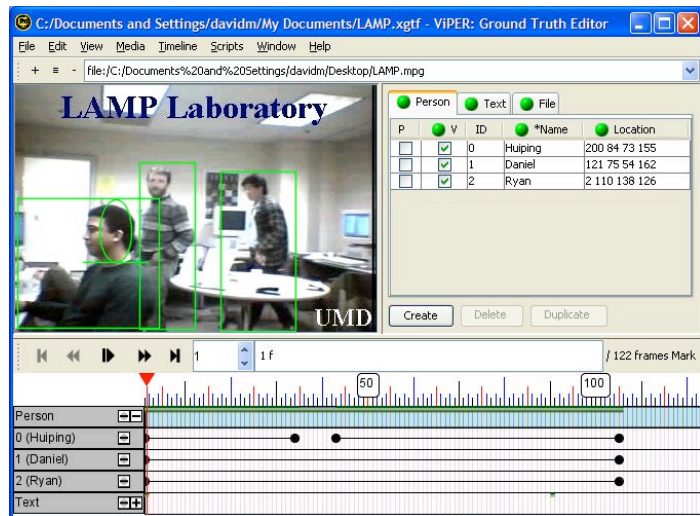
<sup>2</sup> Most current MPEG-7 tools put many interesting annotations into ‘Free Text’ data blocks.

## 4. User Interaction Design

### 4.1. Introduction to ViPER-GT's User Interface

Video annotation is often a repetitive, stressful task with little cognitive involvement. To reduce repetition and increase engagement, the data model should have a complete and consistent visual representation, and, where possible, be directly editable. Additionally, ViPER-GT supports such standard user interface features as unlimited undo and automatic saving. ViPER-GT presents the user with powerful controls customizable to most tasks and generic enough to be immediately usable. This includes views and data management principles that let the user focus on the data that is relevant to his or her goals.

The interface presents three main panels, with several supporting panels. The *Video Canvas* contains the currently selected video frame and any spatial annotations. The *Score* presents a timeline summary of the annotation, while also providing basic time-oriented editing abilities for describing events or scenes. The *Spreadsheet* presents a data-oriented view, allowing access to non-spatial data. The *Remote Toolbar* is for controlling video playback. Secondary panels include the *Undo History*, the *Schema Editor* and the *Media File Editor*. These panels are small, or hidden and only used at certain key times (building the schema, adding media files, correcting complicated mistakes).



**Figure 1: Editing in ViPER-GT. The Video Canvas is on the top left, the Spreadsheet on the top right, the Remote is in the middle and the Score is at the bottom.**

### 4.2. Modality

The user interface presents a unified interaction model; each view is a separate JavaBean, but they act in unison. Each view reflects the current the user state, including what objects the user is currently interacting with, selected objects, any items that are marked as 'locked', etc; the mediator maintains the user's mode, and each view remains aligned with the user's intension by following the mode state. For example, each view displays selected items prominently, hides hidden items and makes animates changes to data. To accomplish this, they each plug into the same model bean that synchronizes user interaction, the *ViperViewMediator*. The Mediator bean has methods to access all of these modes; this makes the class quite complex. Keeping these in the Mediator keeps each view from having to know about each other.

It is easy to overwhelm the user with information. In a long video, the views may fill up with irrelevant descriptor information, hiding relevant data and slowing the interface. Data is

often correlated spatially and across frames, so the interface should support techniques that allow redundant information to be entered quickly or only once. Making sure that modalities act across components increases their benefit and makes it much easier for the user to manage state, hiding the data that should be hidden and linking the data across frames that should be linked by the interpolation manager.



**Figure 1: Editing spatial annotations with the Video Canvas**

The most important element of user state is the current user selection. Closely related is the hover item, which acts as an estimate of the item the user is thinking about selecting. In addition, individual data items take on a variety of modes, including preferences for interpolation, propagation, visibility, lock state, if they represent a key segment of the video, or if all interpolation and display calculations should take the value as the reference point.

These different modalities use a generic *Selection* interface available from the Mediator bean. For example, to set the user selection, a view accesses the Mediator's *getUserSelection* method; to hide, lock, or display and unlock a value, a view accesses the Hidden/Locked/Visible tri-state selection via the Mediator's *getHidden* method. On the *Selection* interface, the method *isSelected* allows each component to determine if their data items are in the given mode. Alternatively, the *getPrimarySelection* and *getAllSelected* list the selected elements. Specific modes implement additional fields, like *Hider*'s *getHiding* methods.

ViPER-GT's user selection mode has two orthogonal values: the time selection and the data element selection. The user can only select a single time value. Functionality that would normally use a range selection must use other features; the 'Time Marker' function or the 'Display When Valid' flag, described in the *Score* section below, both provide means of selecting or labeling multiple points or ranges of time. The data selection may contain an attribute instance, a descriptor instance or a descriptor class. Multiple selections are partially supported, but most of the views only operate on the last item added to the selection.

The Propagate mode indicates which data elements should be copied when the user changes frames. This should interact well with the Display with Respect To mode – meaning, values will propagate in the transformed space, not in the original video space.

While the Mediator addresses the internal means of handling modality, it does nothing to make sure the user is always quite aware of their mode. It would be bad to have the user think items were gone when they are hidden, or the interface is unresponsive when items are locked or broken when an item is set as 'Display with Respect To.' Each modality should have the ability to modify, display and reset their items' state, making the features more discoverable and usable. Each view must not only play by the rules of each mode, they must actively support them when possible, allowing the user to edit the mode set as is logical. The Mediator provides is no programmatic enforcement of the lock set, and each view must check this before modifying data.

This has led to a few bugs introduced when locked data is still editable by a new UI addition, such as a plug-in or a new hotkey binding. Similarly, for ‘Display with Respect To,’ the Interpolator object must know about every spatial data type to compute the required transforms. The design description of each view describes how each component handles the different modes, and the thinking behind them.

### **4.3. The Video Canvas**

#### **4.3.1. Design**

The Video Canvas displays spatial annotations on the current video frame. It supports zooming and ‘Display with Respect To’, a playback mode that keeps a selected item fixed while moving the display around it. In addition, it supports live editing; for example, a user can drag a box to follow a person walking through a scene. The goal of this view is to display the video and any annotations in a method oriented to editing and adding new items.

Zooming and panning can be accessed with the scroll-wheel; dragging the mouse results in a pan action, while spinning the scroll wheel zooms. The left mouse button is used for selection and editing, while the right button is reserved for invoking context-sensitive menus. A single left-click selects an item, if it is far away from the current selection (or if no item is selected), and ctrl-click cycles through possible selections. The interface stores a View History, acting like the Undo History, so the user can go back to a previous selection and view. Piccolo allows continuous zooming, so the view history only stores views that are static for more than two seconds.

To add data, the user must first select an empty Attribute in the spreadsheet (an Attribute with a NULL value), then draw the spatial data item. Each item has a different drawing technique: points are placed with single clicks, circles draw with a drag from the center to the edge, and so on. Information about how to draw each shape can be found in the User’s Manual. Where possible, shape editing follows similar conventions to existing drawing applications, like PowerPoint and Illustrator. [19]

#### **4.3.2. Implementation**

The Video Canvas presents information from the ViPER API and the Mediator’s mode to the user using a Piccolo canvas and a set of video decoders through the DataPlayer object. The major part of the framework is in the edu.umd.cfar.lamp.viper.gui.canvas package, but specialized Piccolo nodes and listeners are in packages by data type.

Piccolo, a Zoomable User Interface toolkit developed by Ben Bederson at the HCIL at UMD, comes from a long line of zoomable toolkits. [17] It offers two main features: a data structure framework for manipulating two-dimensional data, the Piccolo scene-graph, and a visualization and editing framework using Java’s Graphics2D (or Window’s GDI+) framework. Each data type has an associated subclass of PNode, and the Viper Data Piccolo Layer (*ViperDataPLayer*) loads and attaches them as required.

Video decoders that support accurate random access, especially for MPEG-1 and MPEG-2, are hard to find. After giving up on a heavily modified version of an existing open source MPEG-1 decoder, Jonathann Schneier, an undergraduate at the University, wrote one from scratch. When VACE settled on MPEG-2 for their second round of evaluation, ViPER-GT incorporated a version of VirtualDub's MPEG-2 frame-accurate decoder modified to support Java. Both the pure Java and VirtualDub decoders support serialization and deserialization of the index, a requirement for reasonable start times. Unified access to these decoders, and several others, is found in the 'edu.umd.cfar.lamp.viper.gui.players' package with the *DataPlayer*'s *getPlayer* static method. This method tries each decoder in an order determined by the file extension, failing through to the decoder that finally displays the item, or the *NotFoundPlayer*.

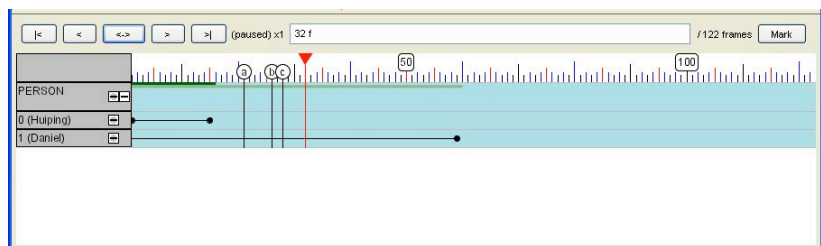
Piccolo determines display order not by an explicit Z-value, but instead by an in-order traversal in the scene graph, where later elements are rendered above the previous elements. To make sure the appropriate elements are visible, the *ViperDataPLayer* has separate sub-layers for the video, the locked data, the unselected data, and the selected data. The viper data layer object listens for changes to the mode and data, and modifies each layer independently to avoid too much work. The renderers for each spatial data type are loaded in the system.n3 file using Limn3's OWL preference manager.

Adding 'Display with Respect To' support was not difficult. Piccolo can support arbitrary affine transformation (e.g. keep this box level, or keep this box square). The Video Canvas adds a translation matrix to keep the centroid of the DWRT box fixed in the view. This is implemented by adding a Piccolo view transform to the *ViperDataPLayer*, which is updated as the frame or data changes. Unfortunately, it causes a race condition during live editing, so the DWRT shape is locked during playback for direct (mouse-on-canvas) editing. Propagate is still supported, however.

## 4.4. The Score

### 4.4.1. Design

The Score displays an overview of the annotations and, if enabled, an overview of the video itself with the optional filmstrip line. HCIL's 'LifeLines' project heavily influenced the design. Similar widgets are found in most of the applications mentioned in Related Work, OntoLog,



**Figure 1: The Remote and Score**

Macromedia's Flash editor and Apple's Final Cut software. These programs all display time series data, with each line representing another aspect of the data. The Score also allows the user to directly manipulate the data, and add bookmarks for navigation and interpolation support.

The Score displays data as a tree with at most four layers: the optional filmstrip, a summary of each Descriptor class, a line displaying each Descriptor's valid region and where auto-interpolate is active, and optional lines for each attribute. At the left, the labels include the descriptor name and id, as well as a pair of buttons. These are the tree expand/contract icon and the 'Display When Valid' button. Clicking 'Display When Valid' restricts playback to when any Descriptor is valid (for filmstrip), any Descriptor of the selected type is valid (for the Descriptor class), that Descriptor is valid (for the Descriptor instances) or where the Attribute is non-null (for Attribute lines).

Currently, only the Descriptor valid line is editable. The user may drag the endpoints, which changes the valid region. The end points snap to any marks that are set. Holding shift ignored the snap regions.

The play head is a special mark, displayed in red and with a triangle icon rather than the normal black marks with circular, lettered icons. Dragging the playhead immediately changes the frame but releasing it while the mouse is not inside the Score resets the mark to its previous position (before the user started dragging). Dragging the playhead also pauses playback while the user is dragging the mark. To add a new bookmark, the user clicks the 'Mark' button. Each bookmark has a pop-up menu that includes such items as 'Move To' and 'Delete'.

A future version should support arbitrary arrangement of Descriptor instance lines through a dynamic query interface, which would better support event annotation. In addition, the current interface can be cluttered for long sequences, or if there are many ephemeral items. A method for hiding/shrinking lines, or packing multiple lines into a single horizontal region, would help the user manage information.

#### **4.4.2. Implementation**

The Chronicle widget, which is the basis for the Score, is stored in a separate source folder with a different license (LGPL) than the rest of the project, to encourage generality and third-party adoption. It is divided into a data model and a display package. The basic data model is a *ChronicleDataModel*, which contains *TimeLine* objects. Each *TimeLine* object has a valid range, a title, and a list of child *TimeLines*, with each *TimeLine* acting as a node in a graph. The display of each line is determined with the *TimeLineDisplayFactory*, which converts from *TimeLine* to display *PNode*, such as lines-with-dots and rounded-rectangles.

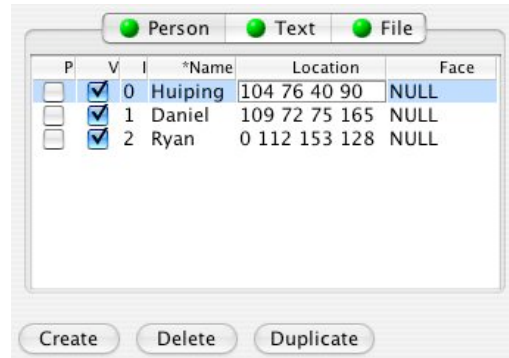
The Chronicle package includes a few default data implementations, including a Tree graph, which ViPER-GT subclasses. The tree model includes the expand/contract emblems. The current implementation often chokes on longer videos, since this may involve the display of tens of thousands of PNodes.

### **4.5. The Spreadsheet**

#### **4.5.1. Design**



The Spreadsheet View displays the metadata associated with the current frame in an excel-like table view, with a sheet for each Descriptor class, a row for each Descriptor instance and a column for each Attribute. An extra column is used for displaying the Descriptors that are in the Propagate mode. Another column displays the current value of the validity bit for each Descriptor. Descriptors labeled invalid on the selected frame may be hidden with a menu option (View » Hide Invalid Descriptors). Selecting a value, in the Score or Video Canvas, scrolls to display that value in the Spreadsheet. Each header shows a Show/Locked/Hidden tri-state toggle, allowing the user to lock or hide attribute types or descriptor types. Each column is also sortable, and the columns may be rearranged or removed. Since they are constrained to be limited in number or time, all CONTENT descriptors and all FILE descriptors are combined on to single sheets. In addition, FILE instances cannot be marked as invalid or contain dynamic data, so they do not need the V (Valid) or P (Propagate) columns.



**Figure 1: The Spreadsheet view on Mac OS X**

#### 4.5.2. Implementation

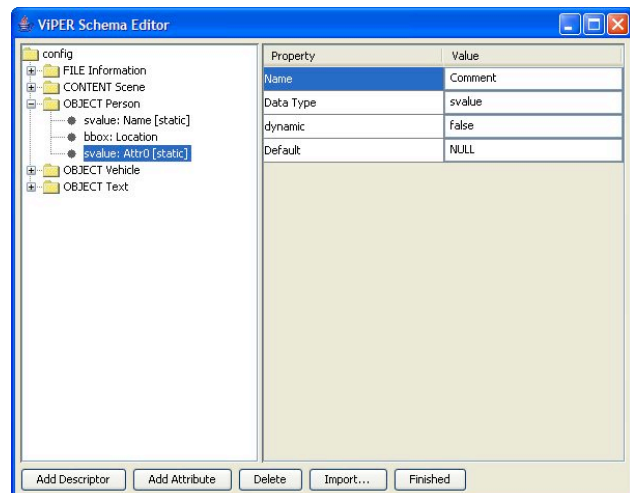
The Spreadsheet uses an extended JTable object that has a more intelligent event listener and a few helper methods for sorting columns and cell/row selection. The panel which contains the TabbedPane is implemented in the class TablePanel in the edu.umd.cfar.lamp.viper.gui.table package.

To implement editing, each Attribute type implements its own TableCellEditor. To be more helpful, each implementation performs data checking and appropriately inform the user when bad data is being entered, either by displaying a red cell or text, or, more often, simply ignoring bad data with a beep and flash.

### 4.6. The Schema Editor

#### 4.6.1. Design

One of the most important features added in the ViPER-GT 4.0 redesign is the ability to edit the schema. The Schema Editor uses a two-pane structure/ details view, with the user adding and removing descriptor and attributes in the structure editing tree view. Descriptor classes and Attribute definitions are displayed in the left-hand pane, while the details of the selected node are displayed on the right. Buttons modify the structure, while



**Figure 1: The Schema Editor**

the details are in a directly-editable properties sheet.

The Add Descriptor button does just that, while the Add Attribute adds an Attribute to the currently selected Descriptor class, or the Descriptor class of the currently selected Attribute definition. The Remove button removes the currently selected item, either Descriptor or Attribute. These buttons work on live data and support undo. Deleting a Descriptor class removes all instances of that Descriptor class, and changing its name affects all instances of it.

The properties sheet supports Descriptors, the default Attribute constraints and extended Attribute definitions. That is, changing the attribute type to 'Enumeration' adds another property, 'Acceptable Values', where the user enters a list of values for the enumeration. Similarly, a 'Text Line' data type has a 'Value Link' field, which connects the Text Line shape object to a String valued Attribute in the same Descriptor class.

Much of the schema editor functionality should be added to the other views. For example, changing the name of a Descriptor class should be an option when right-clicking the tabs in the Spreadsheet view. Also, a '+' button next to the tabs in the spreadsheet view should add a new Descriptor class. This would be much more discoverable than expecting the user to find the Schema Editor tucked away in the Windows menu.

#### **4.6.2. Implementation**

The configuration editor is designed to reach a good equilibrium of usability and ease of implementation. While a UML-type editor would be preferable, a tree/details view editor achieves much of the necessary functionality. When a good open-source UML editor for Java appears, ViPER-GT 5.0 can integrate it. For now, the interface supports two main items, the tree view in a JTree with a ViperTreeModel data model, and the details pane using a PropertiesPanel from the Limn3 accessories package.

The *PropertiesPanel* can use standard JavaBeans properties, but here we pass it a delegate object, an AttrConfigExtensions bean, so that properties can be added or removed on the fly. This is what lets the panel expand when the user changes the data type to Enumeration or Text Line.

### **4.7. The Remote Toolbar**

#### **4.7.1. Design**

The Remote Toolbar is a small but important component of the interface. It controls the currently selected frame, and allows the user to play or pause the video. In addition, it supports setting the 'Loop' option, which restarts the video when the end is reached, and has buttons for advancing to the end or start of the next or previous section and controlling speed of playback. Since playback is integral and controlling it so fundamental, the Remote functionality is exposed as a set of hotkeys.

The Remote features a ‘Play’ button prominently, and information about the current time code or frame number and playback speed. When playing, the Play button becomes a Pause button. The speed is specified in a numeric spinner, and can either be modified directly or with the ‘Accelerate’ and ‘Decelerate’ buttons. While playback is paused, these buttons act as frame steppers. Holding ‘Shift’ increases the power of the buttons: accelerating or decelerating by greater amounts, or stepping by five frames. The ‘Chapter Advance’ and ‘Chapter Retreat’ buttons go back or forward to the next valid region of the current ‘Display when Valid’ item. Each button displays only an icon, but the tool tip displays the descriptive name, followed by the hotkey in parentheses. Buttons appropriately deactivate when pressing them will not change the state of playback.

#### **4.7.2. Implementation**

The Remote uses a Java Timer to control playback, ticking thirty times a second, checking to make sure a frame is not currently being decoded, and, if not, starting a decode of the frame that should be decoded at the current time. This results in a good approximation of real-time playback, with frames dropped as the interface cannot keep up, but not doing any extra work decoding frames that won’t be displayed at all. The interface is exposed as a simple Java toolbar, using standard Swing elements.

### **4.8. Notes about Miscellaneous User Functions**

#### **4.8.1. Handling Media Source File Selection**

When starting a new ViPER-GT session, the user can either first define a schema or open a media file. The most obvious way to add a file is the ‘Media’ menu’s ‘Add Media File’ option. Another method is the ‘+’ button on the Media Selection Toolbar. This is the only enabled button available when the interface loads, and located at the top-left of the interface. Next to the ‘Add Media’ button are the disabled ‘Replace’ and ‘Remove’ media buttons and the empty ‘Media Selection’ menu. With a media file loaded, the media selection window displays the URI of the currently selected file. For most uses of the program, this is all the functionality needed for managing media. However, it becomes more complex as files are distributed to different users and a ViPER data file describes many source media files.

The link to the media in the data is specified as a URI. This means that the ViPER-GT expects to find the video in a specific folder on the user’s disk. However, it is rare that it will be the same folder when the file is passed off to another user. When a data file is first loaded, and the media file is unable to be located, ViPER-GT begins searching for the file on the hard disk. This can take some time, and may return the wrong file that has the same name, or no file if the media file has a different name. The user is given an option to manually select the media file, as the search process continues in the background. To correct a mistaken choice, or to explicitly change what file is loaded for some other reason, the ‘Replace Media’ button, represented with an equivalency symbol, allows a user to replace the currently loaded file without modifying the metadata.

However, handling large numbers of files can become unwieldy with only a single combo box. The ‘Media Files’ window provides a list of the files described in the current metadata file, and lets the user delete multiple files at once. More importantly, it lets the user delete unselected media files, so the user does not have to load unwanted videos before removing them. It also lets the user sort by their FILE metadata or their local file name, which can be handy for finding the source of problems while editing a file created on another computer.

#### **4.8.2. Managing Undo and Redo**

Another feature new in ViPER-GT version 4 is Undo support. The actual implementation uses the event transaction support in the ViPER metadata API. However, simply storing undo objects forever will result in memory issues. To manage this, the undo history only keeps a fixed number of items, or as many as required to make it to the last save point. This policy is fixed; the user cannot explicitly remove items from the history. However, the user can see a list of the event queue, allowing quick switching between versions of metadata with a double-click on the event of choice.

To support internationalization, the undo events each have an associated URI; the Limn3 loader binds the text description to the Undo item. These descriptions can include information about the event. Since many actions, such as the Interpolate command, can cause a large number of updates to the data structure, and because they are often equivalent to user actions, each recorded transaction generates a single undo event. User loaded scripts must be careful to define their own undo description bindings.

#### **4.8.3. Adding Scripts**

Given ViPER-GT’s goal of being general purpose, to support advanced or case-specific annotation, the end user must be able to add case specific editing features. Editing the Limn3 configuration gives the skilled Java developer many options for the configuration of the program, but does not support casual scripting. Limn3 supports generic scripting with Groovy, a dynamic scripting language that compiles to Java byte code.

This works well, and allows the user to add functionality without learning Limn3’s API, but still requires understanding of ViPER-GT’s implementation. ViPER-GT extends Limn3’s scripting to support a very specific type of scripting: performing an arbitrary transformation on the annotation data. A user can link to any program that takes a ViPER data file on the input stream and writes replacement ViPER data on the output. This allows users to integrate their own machine vision routines into ViPER-GT, or modify the data file to conform to their needs. The extended Script bean loads any executable file in the `~/vipер/scripts` directory as a possible transformation command.

#### **4.8.4. Handling Files**

Saving metadata to a file is available with a standard hotkey (Ctrl+S), or through the File menu. The file name is displayed in the title bar; brackets indicate a save is needed (on Mac OS

X, the ‘Unsaved Changes’ dot is supported with the MRJ Adapter). In the File menu, the import and export options address the problems introduced when ViPER upgraded to a new data format. They also let a user combine multiple files or save pieces of data by media file or frame range.

As important as explicit saving is auto-save, a feature that helps users who save rarely. A timer in the background saves the data to a separate, emergency file once every minute or after ten Undo events are created. When the user restarts the application after a crash, a dialog appears asking if the user wishes to reload the found temporary file.

#### **4.8.5. Interpolation and Propagation**

Since ViPER-GT supports editing large files, and objects often move in smooth paths, interpolation functionality greatly enhances the ability of users to mark up long sequences. ViPER-GT version 3 supported explicit interpolation through a Range Slider, connecting all objects selected for Propagation between the start and end frame with the linear interpolation of their attributes. Version 4 replaced the range slider with the Mark objects in the Score view. This allows multipoint interpolation, and right-click menus on items allows for per-attribute interpolation. However, the interpolation must be explicitly re-invoked to make corrections.

The Interpolation dialog allows the user to select mode of interpolation, allowing the changes to modify only the selected attribute in the selected range. Most importantly, it allows the user to quickly re-run previous Interpolate commands.

## **5. Use Cases**

While designing the fourth version of ViPER-GT, three use cases were used as examples: a text use case, an object tracking use case, and an event detection case. Each one exercises a different aspect of ViPER’s capabilities. The Text Detection case is very specific, and, for the most part, frame oriented. The Object Detection case involves tracking objects across occlusion and over time. The Event Detection case does not use spatial data much, instead demonstrating ViPER-GT’s strengths, and weaknesses, at marking up event data in video.

### **5.1. Text Detection**

Text detection is a problem of finding text, characters overlaid on the video or in the scene, in a noisy environment. At the most granular, a text detector finds frames or video clips that contain text, while at the finest it detects pixels which belong to characters. For the purposes of this use case, we are going to assume that the goal of the text detection is to feed OCR software. Current OCR software usually expects its input to be grayscale text documents, with the text already rectified. Our text detection evaluation will we will test for the location of words and characters in the document, or for the actual OCR results. ViPER-GT’s role in the evaluation is marking up the location of text in the video; this will be used as the *target data* for the evaluation of different Text Detection heuristics.

The text may be evaluated at the block, line, word or character level. Since text in video is rarely more than a few lines long, it would be prudent to evaluate at the line level. This may result in improper matches, as current text detection often use simple models for text which often combine lines or split them at spaces, since split or combined characters are often useless to OCR. With enough time and money to spend on annotation, character evaluation will give the best results.

To better support differentiation of the algorithms, the ground truth is rated by quality from 0, illegible, to 5, well defined and clear. It is difficult to define what an OCR algorithm will find legible, so running multiple evaluations accepting [5], [4-5], ... [0-5] as input will give a reasonable curve displaying how a system breaks down. Comparisons between programs at any one level of text quality would not be prudent. The value of any legible strings will be recorded; illegible strings are given a null value. This also allows goal-based evaluation, comparing the ground truth strings to actual OCR output.

In addition to specifying content and legibility, it is necessary to divide text on objects in the scene from text in overlay graphics, like pop-up text on CNN. This classification is subjective; text in animations may be considered graphic, and animations may appear over a scene. This can be accomplished with an Enumeration that can take the values SCENE or GRAPHIC.

A user first defines the schema; the schema editor's import button can load the agreed upon template schema. The 'Load Directory as File' option allows the user to work with a sequence of key frame JPEGs as if they were a single video. The New Descriptor (Ctrl+N) and Next Attribute (ctrl-}) hotkeys allow the user to create a descriptor and cycle through its attributes quickly. This use case emphasizes ViPER-GT's per-frame editing, including the ability to create and display descriptors with spatial and non-spatial elements.

## **5.2. Person Tracking**

A variety of systems require person tracking, from location monitoring to video summarization. For our purposes, this use case focuses on surveillance situations. This includes both tracking people as they move around and tracking individuals in groups, at different times, different angles or in different lighting conditions. Annotation of these tracks must happen on the frame level; even with support from multiframe tools, this can be a very time-consuming task. ViPER makes it attempts to make the annotation of a single person in the video take as close to real time as possible.

As discussed above, there will be one descriptor for each person that a human editor can recognize as the same person in the video. Unfortunately, identification is not guaranteed, as even humans cannot tell when two people facing away from the camera are the same person.

The most difficult part is selecting a visual representation for a person. Possibilities include:

- bounding box around all visible parts
- bounding boxes around torso, head, arms, and legs
- centroid
- three ellipses (head, torso, legs)
- other methods not covered by the included attribute data types

One possibility, and the one we have chosen in the past, is to place a box around the torso and another around all visible extremities. This allows the area between the torso to be regarded as important and the rest of the area to be regarded as good to have, but not required. This use case selects the Torso + Extremity Box.

This use case emphasizes ViPER-GT's handling spatial data over time. An expert user should take advantage of the Interpolate, Display When Valid and Display with Respect To features.

### **5.3. Event Detection on the Tarmac**

LAMP developed software to detect events from a surveillance camera trained on an airport runway. Events found on this sequence include groups debarking from planes, refueling planes, taking luggage out of aircraft, and, occasionally, people walking out of the standard path. The interesting events range from the arrival of planes to finding people walking astray. While it might be useful to mark up the tracks of the people (too small to identify) and aircraft, the video sequences are long (hours) and the tracking results not of interest except to infer the events.

There are two possibilities for marking up events in ViPER-GT: a generic Event descriptor type, and specifying a different Descriptor type for each event type. For this use case, an Event Descriptor is defined with a Type Enumeration attribute and several Relation Attributes named 'Participant 1', 'Participant 2', 'Vehicle' and 'Object'.

This use case focuses on non-spatial data over extended periods of time, exercising the Score view and the tools ability to reliably access large media files.





## 6. Extending ViPER-GT

### 6.1. The ViPER File Format, XGTF

A ViPER data XML file contains two major sections, for schema and data. The schema section contains a list of <config> elements, each describing a Descriptor class and its Attributes. The data section is divided by source media file, with each <sourcefile> element containing a list of <descriptor> elements. In the older plain text format, there was just one data section, with a one-to-one mapping from markup file to media file; the source file element allows one markup file to describe any number of videos. This represents one of the main reasons for developing a new file format for version 4; the unified XML file is easier to distribute, and its explicit links to the source media make it easier to work with, despite changes in the file name. However, it makes life harder when the media file is moved or renamed, as the existence of a link will lead users to expect the system to just work. For a detailed description, refer to The Guide to Authoring Ground Truth with ViPER-GT. [19]

#### 6.1.1. The FILE Information Descriptor

It is recommended that all ground truth contain a FILE Information Descriptor. ViPER-GT automatically adds one when a media file loads. This Descriptor includes number of frames, frame rate, image size, and other information essential for evaluation or display in the absence of the source media file.

Attribute Name	Attribute Type	Meaning
NUMFRAMES	Integer	Total number of frames the video file (or collection of images) contains.
SOURCEFILES	String	A list of (number, filename) pairs. The number is the frame offset; the file is the media source name.
SOURCEDIR	Enumeration	Type of the media file, either SEQUENCE for a video or FRAMES for a set of frames (e.g. a .info file).
H-FRAME-SIZE	Integer	Horizontal width of video, in pixels.
V-FRAME-SIZE	Integer	Vertical height of video, in pixels.

Specifics of the XML Format

### 6.2. Accessing the Data through the ViPER API

The ViPER API offers a set of Java interfaces and classes for manipulating data in the ViPER XML format. It offers a hierarchical view of the data, allowing manipulation of the configuration and the instance data. The main package of interest is `viper.api`, which contains the `ViperData` and supporting interfaces. The implementation is found in the `viper.api.impl` package, with current Attribute data types in `viper.api.datatypes`. This section assumes an understanding of the ViPER data model, described above and in the ViPER-GT manual.

### 6.2.1. Creating an Instance of ViperData

Here is a simple Java example of accessing ViPER data:

```
ViperData data = new ViperDataImpl();
Config fileInfo = data.createConfig(Config.FILE, "Information");
fileInfo.createAttrConfig("Framerate", "fvalue", false,
    new Integer(29.997), null);
Sourcefile f1 = data.createSourcefile("sample.mpg");
Descriptor first = f1.createDescriptor(fileInfo);
first.getAttribute("Framerate").setAttrValue (new Integer(25));
System.out.println ("Framerate is: " +
    first.getAttribute("Framerate").getAttrValue());
```

The above code creates a new instance of a ViperData object. The second line creates the configuration for a File descriptor with the name “Information.” As described above, a Descriptor with this name and type is important for evaluation. It then adds a single “fvalue”, or Float, Attribute to the Descriptor, and then creates an instance of it. Note that a Descriptor must be attached to a media source file, although that file may have no name (pass the creation method *null*).

As you can see, to access the attribute value, you first have to get an attribute data object, instead of acting directly on the descriptor via *setAttributeValue* and *getAttributeValue* methods. The attribute objects may save on lookup, but helper methods may save on code. It should be noted that Attributes for dynamic values have a different set of access methods: *getValueAtInstant* and *setValueAtSpan*. Also, there are methods to iterate over the non-null values in an interval. Accessing a dynamic attribute with the static methods, or the static attributes with the dynamic methods, will raise an *IllegalStateException*.

### 6.2.2. Reading from and Saving from XGTF

Here is a sample of code that reads ViPER data from an XGTF file:

```
FileInputStream inFile = new FileInputStream ("example.xgtf");
Element documentEl = DocumentBuilderFactory.newInstance()
    .newDocumentBuilder()
    .parse(inFile)
    .getDocumentElement();
inFile.close();
ViperData data = ViperParser.parseDoc (documentEl);

PrintWriter pw = new PrintWriter(new OutputStreamWriter(
    new FileOutputStream("out.xgtf"), "UTF-8"));
XMLSerializer.toWriter(data, pw);
pw.close();
```

The above code opens the file “example.xgtf” for parsing, parses it into the XML DOM using the javax.xml package, extracts the ViperData using viper.api.impl.ViperParser and saves it to a new file. The Xerces parser from xml.apache.org will also work.

### 6.2.3. Transactions

Transaction support is useful for combining multiple events as a single action. It also has the benefit of automatic rollback, in event of a runtime exception. A *doTransaction* method takes a Runnable object and executes it within the context of a transaction. The following Groovy script snippet shows how to add transaction support to a ViPER-GT script.

```
void run(AppLoader application) {
    mediator = application.getBean("#mediator")
    sf = mediator->currFile
    if (sf != null) {
        success = false;
        trans = sf.begin(myIdentifier);
        try {
            // Modify sf here
            success = true;
        } finally {
            if (success) {
                trans.commit();
            } else {
                trans.rollback();
            }
        }
    }
}
```

### 6.3. Adding New Data Types

The most common user request is to add a data type to ViPER-GT. For each data type, an AttributeDataWrapper handles parsing and serialization, Java JTable cell renderer and editors are used to display the item in the Spreadsheet and on the ‘Default’ item in the schema editor, CanvasEditor and CanvasCreator event handlers and an Attributable PNode add support for the Video Canvas view, and an attribute data renderer added to the RendererFactory lets the data type have a custom display in the Score view. To add a new type, these classes can be constructed in any Java IDE, and then added to the classpath and the Limn3 system.n3 file must be modified to load them.

The first extension is reading in the data to the API. This involves adding a new data wrapper. Some examples are in the viper.api.datatypes package. For a simple example, see the viper.api.datatypes.Bvalue, while the viper.api.datatypes.Polygon is a more complex example. The attribute wrapper must implement AttrValueParser to support serialization and deserialization, and DefaultedAttrValueWrapper to support taking default values. (To specify

parameters for the data in the schema section, implement `ExtendedAttrValueParser` instead of `AttrValueParser`.)

An *AttrValueWrapper* is a utility object that enforces typing on the attributes. It can also convert between an internal and external format. For example, an Enumeration can convert between a String and an Integer. This allows space savings and possibly allows the use of `==` instead of `.equals()`. The interface *AttrValueParser* extends this to support serialization to an XML DOM object. Note that the serialization functions use the encoded format of the data. To add the data type to the list of allowed data types, modify the `n3` file to invoke the *ViperDataFactoryImpl*'s *addType* method.

While the parser does not require it, any new data types must have an XML Schema definition somewhere for the ViPER XGTF file to be valid XML. Writing the XML Schema encourages good design, and gives other developers who wish to use the data type a reference. In the ViPER XML Schema, each data element must extend the *viper:null* root type. As far as design of the data type itself, types that more of their structure in XML are more accessible to XML tools that use XPath and XQuery. For example, bounding boxes include `x`, `y`, `width` and `height` as individual XML attributes, instead of saving them in a single 'value' attribute.

To test the wrapper object, the Java command line tool `viper.api.impl.ViperParser.FileTranscriberTest` parses a file and serializes it again. Comparing the output and input and checking the error stream will demonstrate any major errors the file format and the wrapper object.

To add support to the Spreadsheet view, each data type implements a *TableCellEditor* object. The Spreadsheet view uses the `Limn3` properties object to get the *TableCellEditor* for each attribute. Similarly, to add support for a spatial attribute to the Video Canvas, you have to implement two event handlers (one for creation, another for editing) and create a `Piccolo` node class for it. These are specified in the `n3` file with the *gt:visualCreator*, *gt:visualEditor* and *gt:visualNode* properties.

## 6.4. Adding New Interaction Methods: Wizards and Widgets

In developing large amounts of ground truth for specific applications, it is often cost-effective to develop custom panels, or even data types, to extend ViPER-GT. This is in line with the finding in the Related Work section that users prefer special-purpose tools. For example, it might be useful to build a face annotation wizard to select face points, or a zone tool that allows the user to quickly label zones of text.

### 6.4.1. An Example: Labeled Text Zones

One common form of ground truth is labeled text zones in a document. For normal, scanner based OCR, this ground truth is often generated directly from source. However, no such possibility is available to test camera based OCR, new script identification algorithms and any system that works on older data. This example extension uses a simple schema, with a single

OBJECT Descriptor type called ZONE, with three attributes: an Enumeration indicating the zone TYPE, a Bounding Box indicating its LOCATION, and a Relation LINK indicating its place in a directed graph of other ZONE descriptors.<sup>3</sup>

#### 6.4.1.1. The Zone Editor Panel

First, a panel specifically for ZONE objects acts both as the main UI for this mode of editing, but also to act behind the scenes with the Limn3 framework as a plug-in. There are two zone editor panels in the example code: *SimpleZoneEditor* and *FinalZoneEditor*. This document introduces the *SimpleZoneEditor*; for more detail, look at the example package documentation.

*SimpleZoneEditor* extends *JPanel* to provide the Zone editor functionality; it presents a button selecting the type of zone the user wishes to draw. Clicking the same button twice exits zone editor mode, so the user is able to access the standard ViPER-GT functionality. The panel also features a checkbox allowing the user to quickly hide all other spatial data. It requires editing the *gt-config.n3* configuration file to load the plug-in.

For ease of the example, the *SimpleZoneEditor* uses *BoxLayout*. It features a single check-box which hides the other shapes, and a set of buttons that is derived at run-time from the list of values the ZONE Type attribute may take. In order to do its work, the checkbox uses the Mediator's hiding mode. The buttons are somewhat more complicated.

In ViPER-GT, a user must first create a Descriptor instance, usually by clicking the 'Create' button on the spreadsheet, before configuring its Attributes. To support immediate creation, the *SimpleZoneEditor* introduces a hack in the zone type buttons' action listeners. When the user first clicks a ZONE button, its action listener creates a ZONE Descriptor instance of that Type, and sets the user's selection to the Location attribute of the object. This way, the Video Canvas view knows that a click will mean the start of a box creation event for that Descriptor. Clicking on another ZONE Type button changes the type of the unplaced ZONE, and clicking the same button deletes it. When a ZONE is placed on the video frame view, a *ViperMediatorChangeListener* will know to create a new ZONE.

#### 6.4.1.2. Adding Color to the Types

To add color to the boxes, the sample code includes a replacement for the existing Piccolo node object with a new one. The association of attribute type to Piccolo node is found in the *gt-config.n3* file. The new node class overrides the *get\_x\_DisplayProperties* methods. These methods tell the node how to render its lines while under highlights and other positions. To determine the type of the box, it climbs the ViPER API. It first invokes *getAttribute* to gain access to the viper data, then *getParent* to get a reference to the containing object. If the

---

<sup>3</sup> The code described in this example is available in the full source download from SourceForge.net in the *edu.umd.cfar.lamp.examples.zones* package.

Descriptor is not an instance *Zone*, it returns the *super* method value. If it is, the custom renderer uses the *Zone Type Attribute*'s value to select a different Paint color.

#### 6.4.1.3. Displaying Connections

The replacement *Piccolo* node has a subnode in its *Piccolo* hierarchy that is a pointer to the next item. It appears at first as a single handle at the bottom of the box. Dragging the handle displays an arrowhead, which allows connection to another *Zone*. This lets the user connect a zone to its successor, or back to itself to indicate no connection.

## 7. Future Work

There is a variety of improvements to be made to the system. To decide what direction to take, a user study or detailed expert review should rank the importance of each improvement. In addition to features required for parity with the tools described in *Related Work*, such as support for *OWL* and audio annotation, each component should be checked over carefully and the whole program profiled for memory leaks and performance problems.

For example, *Limn3* supports serialization of entire programs to save on start time and maintain user state across sessions. It can save two instances of the state: *afterStart.dat* and *last.dat*. It checks against the modification time on the *n3* files in its path and loads the most recent file. To support this, all of the views loaded with *ViPER-GT* must be serializable. To avoid bit rot, the user interface should include a 'Reset' button, which removes the serialized state files and reloads the application from the *n3*.

Another large barrier to adoption is the difficulty in installing the software. Many major Java desktop applications now support *WebStart*, a standard method for updating software and quickly putting it on users' desktops; with the click of a link, a user can install an entire application. To support this, *Limn3* must be modified to search for *n3* files in more standard places. This will likely take place in the context of a general code audit and upgrade to Java 5. In addition, updating to Java 5 will lighten the *ViPER* download, as it will require fewer auxiliary libraries; Java 5 includes some of the features provided by the *XML* and concurrency libraries included with the *ViPER-GT* download.

The *Interpolation* tool has come a long way since it was first introduced. It now supports marking regions as 'interpolated', and allows a user to quickly 'update interpolated regions'. However, this process should be transparent, like animation tweening in *Macromedia's Flash* developer application. Here, the values on the tweened frames will be updated automatically whenever the keyframes are modified. This will likely involve moving interpolation support from a client library in *ViPER-GT* to a standard function of the *ViPER* API itself, adding a value check to the 'getAttributeValue' methods themselves. Since events are generated on the attribute level, this will not cause a problem with existing event handlers.

A list of additional modifications to the *ViPER* interface is on the *SourceForge* web page. All these changes should improve the experience, quality and efficiency of markup. Improving

the user experience overall, including things like caching the user state across sessions and using WebStart, will help adoption of the program. ViPER-GT is singular in its ability to annotate video; improvements to its interface and core functionality will assist it in becoming a standard utility for video annotation and evaluation.

Even more important than core annotation functionality is extensibility. Many researchers do not have time to focus on developing a user interface for their computer vision routines. ViPER-GT's generality required the development of a variety of features useful to the vision community at large, including a suite of frame accurate Java video decoders, a simple metadata file format and an application for quickly browsing generated annotations. The most important feature to be added to ViPER-GT is to make it as available to developers as possible and easy to extend. A humane interface is necessary for ViPER-GT to continue as a simple annotation program developed by a closed lab, but a humane API is required for ViPER-GT's future as an open source project. ViPER-GT has the potential to become the basis for a plethora of computer vision research tools; with the development of a vibrant developer community, like that around ANVIL and Piccolo, it could help make vision algorithms research more user focused and improve communication across the entire community.

## 8. Appendix: The Limn3 Application Loader

ViPER-GT version 4 uses a custom Java application launcher, the Limn3 Application Loader. Limn3 reads Java application descriptions from a text file and builds the user interface on the fly. It addresses a variety of concerns with IBM's Bean Markup Language format (primarily its 'retirement'). [14] The application description is stored as OWL data in an N3 file. Limn3 includes a variety of helpful objects for dealing with Java Swing applications, including OWL ontologies for menus, Undo histories, icons and a JavaBean ontology. This means that it is possible to rearrange the menus, and even add or remove components in the display, by editing a text file. Also, the use of OWL instead of XML lends itself to greater extensibility, better error messages and a more readable file format.

This Appendix should give the reader an understanding of how Limn3 loads an application. Debugging a Limn3 configuration will be difficult, if not impossible, without an understanding of how it loads and initializes itself.

Short Name	Schema URI	Description of Elements
limn3	<a href="http://vipertools.sourceforge.net/owl/apploader#">http://vipertools.sourceforge.net/owl/apploader#</a>	Provides the core set of Limn3 OWL classes and properties, as well as declaring the Core javabean, which is an instance of AppLoader. This schema provides most of the functionality found in Bean Markup Language
option	<a href="http://vipertools.sourceforge.net/owl/apploader#">http://vipertools.sourceforge.net/owl/apploader#</a>	Binds command line arguments, Java properties and

	toolkit.sourceforge.net/ow/apploader/prefs#	system environment variables to bean properties or generic RDF triples.
menu	http://viper-toolkit.sourceforge.net/ow/apploader/menu#	Used for specifying menu items and their actions.
mru	http://viper-toolkit.sourceforge.net/ow/apploader/mru#	Keeps a 'Most Recently Used' file list.
undo	http://viper-toolkit.sourceforge.net/ow/apploader/undo#	Binds Undoable events to their descriptions.
props	http://viper-toolkit.sourceforge.net/ow/apploader/properties#	Useful for describing beans in detail, beyond that found with JavaBeans introspection. This is mostly used for generating property sheets, such as the one in the Schema Editor.
hotkeys	http://viper-toolkit.sourceforge.net/ow/apploader/hotkeys#	Binds hotkeys to action objects.

**Figure 2: Limn3 schemata**

## 8.1. System Management

This section focuses on how Limn3 loads components and responds to changes in the preference model. At the heart of the system is an RDF triplestore, which keeps the state of the program description, and a core bean, which provides the menu panel and a place to attach other GUI elements; a *PrefsManager* object maintains the triplestore, and an *AppLoader* class implements the core bean.

### 8.1.1. Bootstrapping

Limn3 first creates a *PrefsManager* object, then constructs an *AppLoader* bean. The *PrefsManager* handles loading n3 files and modifying the in-memory application description. It first loads a 'system model,' the initial set of triples. In ViPER-GT, this is the `gt-config.n3` file. The *OptionsManager* object converts command line arguments and environment variables to OWL data or runs various commands specified in the preference model when certain variables are set. The *AppLoader* is a subtype of *JFrame*, and includes a place to add other components and menu items; it has the Limn3 URI `limn3:Core`. In its constructor, it loads and initializes the application beans, sets up the menu and initializes itself as a Java Swing application.

The first step is loading the System model. The system model is either loaded from the file found at the "limn3.prefs" property (passed to the JVM with `-Dlimn3.prefs="file name"`), or from the file found at the location "CONFIG/system.n3" or "CONFIG/system.xrdf" (useful for packaging jar applications, which do not support arguments to the JVM in the manifest).



After the system model is loaded, the options are activated. The options are specified as *option:PreferenceFlag* or *option:PreferenceTrigger* resources. These present a unified method for defining the command line options, preferences and system variable handling, as well as automatically generating the usage statement (along with the *limn3:Core* *limn3:shellCommand* "appName <options>" . triple). Triggers represent key/value pairs, and flags represent toggles. A preference object can have a short, standard or long name. The short name is a single character abbreviation preceded by a single dash on the CLI, specified by *limn3:abbr*. The long name is a standard name that is preceded by two dashes on the command line, specified by an *rdfs:label*. The long name, which is used for java system properties passed via the -D option to the java virtual machine, is specified with the *limn3:longName* property.

After the preference triggers and flags are parsed, the *OptionsManager* puts them into action. First, it sorts through the system properties for the longNames. Then it parses the command line in order. For each trigger or flag parsed, the system invokes the functor specified with the *option:invokes* property. The functor class must implement the *TriggerHandler* interface. The current system doesn't implement concatenating short names, like tar does, but there is nothing really preventing this, and it might be useful for some applications (find, tar, and other cli-oriented apps are possibilities).

There are two implementations of *TriggerHandler* included with the *AppLoader*. These are *PrintUsage* and *PropsToPrefs*. *PrintUsage* does exactly that, printing out a usage statement to the command line and exiting without an error code. The *PropsToPrefs* handler converts the value of the preference trigger into a triple, inserting the reified value of the *option:inserts* into the system model; if it is a flag, it inserts it unmodified, but a trigger will add any missing objects in the inserted triples as a string literal.

With the model complete, and assuming no options have resulted in an exit or other change in the flow of the program, the Core bean is loaded. The Core then loads and initializes the other application beans. Note that beans are specified as members of the class of *limn3:ApplicationBean*. These beans are global to the whole application, and are around for the lifetime of the application. *limn3:TemporaryBeans* are generated as required, usually when passed to another bean as a property value. *limn3:requires* links are used to determine in what order to load and initialize the beans. To support cycles, these are separate stages. The load phase places a new instance of each bean class into a lookup table, as per the *limn3:className* property. The initialize phase examines each *limn3:setProperty* statement in the triplestore, and sets the properties to the appropriate value, which can either be a literal, another bean, or a property of another bean.

Finally, before the main frame is displayed, the menu is configured. The menu setup is much simpler than the bean setup. It simply attaches menu items to menus (or the Core bean), associating *ActionListeners* to the items. The *ActionListeners* must be retrieved from loaded application beans.

### 8.1.2. The Triplestore

The RDF triplestore is held by a *PrefsManager*. In the current incarnation, it is a Jena2 RDF Model with extensions to handle change events. The preference manager bean also offers some helper methods, and gives a tiered system of maintaining the triples, while providing a unified store. The first tier is the System tier, loaded when the system starts from the *limn3.prefs* file. The User tier is next, loaded from the *user.n3* file in the user's application directory (*~/appname/user.n3*). Next is the temporary tier, which is not serialized.

Although the preference manager provides access to the unified model, direct changes to it will be ignored. Instead of accessing the Jena Model directly, all changes must go through the *changeModel* methods on the *PrefsManager*. These methods take a set of triples to remove and a set of triples to add, in the form of Jena models. Changes to the User model will be persistent across runs of the application, but not changes to the System or Temporary triple spaces.

### 8.1.3. Declaring a Bean

There are two types of beans, application beans and temporary beans. Application beans are loaded at the start of the application, may be identified by URI, and are not removed until the application closes. Temporary beans must be referenced indirectly through properties of other beans or through the *limn3:Parent* link, and are created when needed. The initialization procedure is similar for both: all bean properties are set, and then the visual representation is dealt with.

A *limn3:setProperty* predicate identifies that a property of the bean should be initialized to a given value. The target must be an *limn3:BeanProperty* resource with *limn3:propertyName* and *limn3:propertyValue* links. The *limn3:propertyName*'s object must be the name of a property. If there is no *props:interfacier* specified for the bean, then this label will be used to find the JavaBean style property setter method on the object. The *limn3:propertyValue* can be either a directly encoded value, an XSD typed literal, a String, an *ApplicationBean*'s URI, the value *limn3:Parent*, another bean property with a *limn3:propertyOf*, or some RDF list of several of these for multiparameter property set methods, as in *setSize* on a *Dimension* object. The *limn3:Parent* resource is refers for the bean that the setter is called on, and is mostly useful for configuring temporary beans, which are anonymous (without a fixed URI).

After the properties are loaded, beans with a graphical representation are displayed if requested. Implementations of *JFrame* are displayed with the *rdfs:label* as their window title, while *JComponents* are not made visible unless attached to an existing *JFrame* or *JComponent* bean. For example, in ViPER-GT, the schema editor and action history are both loaded as a components, but made available through menu items.

### 8.1.4. Declaring a Menu Item or Group

A menu item is like a bean, but is not implemented as such. The menu system has its own menu namespace and schema. A Menu item may be attached to another Menu, to a menu Group,

or to a bean with a window as a visual representation. A menu attached to a window is instantiated within a *JMenuBar*. A Menu attached to another menu uses the standard *JMenu* class. The Groups do not show up as actual menus; these are used to provide *JMenuSeperator*-defined groupings inside a menu. A future system could replace them with sub-menus when the list becomes too long. The priorities are like Python array indexes: zero is the first element, -1 is the last element. A downside to this rule-like approach is that it can be difficult to debug when menu items are out of order.

For example:

```
:viewMenu
  a menu:Menu ;
  rdfs:label "View" ;           # the default text string
  rdfs:label "Regarde"@fr" ;   # a localization
  menu:mnemonic "V" ;         # Hitting Alt+V will get this menu
                              # to come up
  menu:attachment lal:Core ;   # The root window of any Limn3 app
  rdfs:comment "View menu; controls the user's perspective" ;
                              # the comment may be used for the tooltip
  menu:priority "2" . # will come after items 0 (File), 1 (Edit)

:zoomGroup a menu:Group ;     # a group for zoom commands
  menu:attachment :viewMenu ; # within the view menu
  rdfs:label "Zoom" ;         # this could be used later
  menu:priority "-1" .       # will be displayed last
  # groups are smart enough to not render a separator if
  # none is required - e.g. only one group exists, or
  # the group is empty.

:zoomToActual a menu:Item ;   # a menu item
  menu:attachment :zoomGroup ; # within the zoom group
  rdfs:label "Actual Size" ;  # the menu item text
  menu:generates :zoomActualSizeActionate ;
                              # the event generated. This is a bean!
  menu:mnemonic "A" ;        # for B.L.T.-ing
  menu:priority "-20" .      # means the item will be near the
                              # end of the group, but before
                              # any items or sub-groups
                              # with a priority -1 through -19
```

## 9. Appendix: Glossary

*Attribute:* An element of data associated with a descriptor. This is equivalent to a field in a database or a property of an object. Attributes can be specified as ‘static’ or ‘dynamic’, indicating if they can take on different values over the course of a video. Attributes are typed.

*Candidate Data:* Also known as result data, this is the set of descriptors generated by some algorithm that will be compared to the target data in ViPER-PE’s terminology.

*Descriptor:* The main unit of data in ViPER, a descriptor is equivalent to a row in a database, or an object or record in a programming language.

*Detection:* An object is classified as Detected if one of its type is found on the frame. For example, if you want to retrieve all the frames containing faces, you may find that Detection is the only required depth of analysis.

*GTF:* Ground truth file. This is a file containing ViPER metadata, and need not refer to truth but sometimes refers to software generated or otherwise untrustworthy metadata.

*The Remote:* The panel for controlling playback speed and position.

*The Score:* A time line display of the data in a viper metadata file.

*The Spreadsheet:* An excel/lotus view which displays the value of all descriptors on the current frame.

*Target Data:* Also known as truth data, this is the set of descriptors that represent the true content of the media file in ViPER-PE’s terminology.

*The Video Canvas:* The view of the current frame and any spatial markup.

*Visibility Ball:* The colored ball in each tab of the data panel, or at the top of each column. Green indicates that the shapes are visible in the canvas, red means that the shapes are to be invisible and unable to be selected, while yellow indicates the data is locked.

## 10. Bibliography

- [1] J. Hunter. Adding Multimedia to the Semantic Web - Building an MPEG-7 Ontology. In *Proceedings of the First Semantic Web Working Symposium (SWWS)*, Stanford, USA, 261-281, 2001.
- [2] Chang S.-F., Sikora T. and Puri A. Overview of the MPEG-7 Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 6, pp. 688-695, June 2001.
- [3] P. Cohen. *Empirical Methods for Artificial Intelligence*. 1995. 0262032252. MIT Press.
- [4] R. M. Haralick. Performance Assessment of Near-Perfect Machines. In *MVA*, vol. 2, no. 1, pp. 1-16, 1989.
- [5] H. Karacaoğlu. File Exchange Formats and Metadata. <http://www.broadcastpapers.com/>.
- [6] M. Kipp. Gesture Generation by Imitation - From Human Behavior to Computer Character Animation. Boca Raton, Florida: Dissertation.com. 2004
- [7] D. Doermann and D. Mihalcik. Tools and Techniques for Video Performance Evaluation. In *15<sup>th</sup> International Conference on Pattern Recognition (ICPR '00)*, vol. 4, pp 41-67. 2000.
- [8] C. Halaschek-Wiener, J. Golbeck, A. Schain, M. Gropve, B. Parsia and J. Hendler. PhotoStuff – An Image Annotation Tool for the Semantic Web. In *4<sup>th</sup> International Semantic Web Conference – Poster Paper*. 2005.
- [9] C.Y. Lin, B.L. Tseng, and J.R. Smith. Video personalization and summarization system. In 2002 IEEE Workshop on Multimedia Signal Processing. 9-11 Dec 2002. pp. 424-427.
- [10] A. Gruenstein and J. Niekrasz. NOMOS User Manual. 2005. <http://godel.stanford.edu/twiki/bin/view/Public/NomosManual>
- [11] R. Schroeter, J. Hunter and D. Kosovic. Vannotea – A Collaborative Video Indexing, Annotation and Discussion System for Broadband Networks. *K-Cap 2003 Workshop on Knowledge Markup and Semantic Annotation*. Florida, Oct. 2003.
- [12] F. Quek, Shi Y., C. Kirbas and Wu S. VisSTA: A Tool for Analyzing Multimodal Discourse Data. In *Seventh International Conference on Spoken Language Processing*. 2002. Denver, CO.
- [13] Ricoh MovieTool. <http://www.ricoh.co.jp/src/multimedia/MovieTool/>
- [14] S. Weerawarana, F. Curbera, M. J. Duftler, D. A. Epstein, and J. Kesselman. Bean Markup Language: A Composition Language for JavaBeans Components. In *Proceedings from the 6th USENIX Conference on Object-Oriented Technologies and Systems*. Jan. 2001.
- [15] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. A *W3C Recommendation*, 10 February 2004.
- [16] P. Boersma and D. Weenink. Praat: doing phonetics by computer. <http://www.fon.hum.uva.nl/praat/>
- [17] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering*, 30 (8), pp. 535-546. 2004.
- [18] M. Christel. Lessons Learned from the Informedia Project: Development and Evaluation of Digital Video Library Interfaces. UNC SILS Interaction Design Lab Symposium on Understanding Video, University of North Carolina at Chapel Hill, October 17-18, 2002.
- [19] D. Mihalcik. ViPER Toolkit Home Page. <http://viper-toolkit.sourceforge.net/>