The GML canvas: Aiming at Ease of Use, Compactness and Flexibility in a Graphical Toolkit

François Bérard

Ref: TR-IMAG-CLIPS-IIHM_200601

HCI group Technical Report
CLIPS-IMAG
B.P. 53
38041 Grenoble Cedex 9
http://iihm.imag.fr/

We present the GML canvas: a graphical toolkit aimed at supporting the development of post-WIMP interactions such as deformed, zoomable, semi-transparent, or bi-manual interactions. Our primary objective is to make the *development* of such interaction fast and easy; it is not to provide a library of post-WIMP interactions. The toolkit is used from an interpreted environment and generates its output with hardware rendering. We explain the rationale behind the design of the API and we present it in details. The API is split into a set of low-level atomic services and a set of high-level scripts. The atomic services offer the fundamental building blocks for interactive graphics and the scripts assemble these building blocks to serve as examples and to provide general interaction services. A number of examples illustrate how complex interactions have been built with a few lines of code.

Categories and Subject Descriptors: D.2.2 [Tools and Techniques]: User interfaces; **I.3.4 [Graphics Utilities]**: Graphics packages

Keywords: graphical use interface toolkit, post-WIMP, bi-manual interaction, transparent tools.

Introduction

Advances in Graphical User Interfaces (GUI) have shown the need for new graphic services that are not provided by current software toolkits supporting the WIMP (Windows Icon Mouse Pointer) interaction style. These services include non-rectangular interaction objects [Callahan et al. 1988, Chatty et al. 2004, Shen et al. 2004], general geometric transformations such as zoom, rotation and scaling [Bederson & Meyer 1998, Vogel & Balakrishnan 2004], the blending of graphic elements such as transparency [Bier et al. 1993, Vogel & Balakrishnan 2004], managing multiple control points for multi-user and multi-hand interfaces [Beaudouin-Lafon & Lassen 2000, Bier et al. 1993], and the animation of any parameters such as shape, transformation, or blending [Bederson & Meyer 1998, Vogel & Balakrishnan 2004]. In response to these needs, several research efforts have produced a number of post-WIMP toolkits that cover some or all of these requirements. However, these are only the functional requirements for the toolkits. But a toolkit that aim to be useful and widely used must also satisfy a set a non-functional requirements such as ease of learning, coding efficiency, flexibility, and performance. Non-functional requirements are often more difficult to define and to satisfy than functional requirements. For example, it is relatively easy to identify that Zoomable Interfaces [Bederson et al. 2000] require the scaling of the whole scene, and Toolglasses [Bier et al. 1993] require the ability to draw with some transparency. Once identified, satisfying functional requirement only amounts to difficulties on a technical level. However, what defines the ease of use of a toolkit? How does one design an easy to learn toolkit that is also extensible or support efficient coding? Those are difficult problems with no definitive answer.

As researchers in novel post-WIMP graphical interactions, we haven't settled on a suitable toolkit that would cover both the functional and non-functional requirements. Some toolkits only partially cover the functional requirements because they are biased toward one particular interaction, some toolkits have a strong set of services but the effort to learn and operate them is too high because they rely on complex low-level programming environments, some toolkits have low performance and thus are not suitable for tightly coupled interaction, and some toolkits, based on graphical programming, are easy to learn and to use but the set of interactions that can be built is limited. As a consequence, no post-WIMP toolkit has been widely used outside of the laboratory where it was developed. Researchers in post-WIMP interaction often need to reproduce the same work and develop their own abstraction on top of low-level libraries such as OpenGL.

In this paper, we present our design of a graphical software toolkit called the GML canvas. While satisfying all the functional requirements listed above, we focused its design on ease of learning, support of efficient coding, and a flexibility and performance that does not limit the creativity of researchers in novel interaction. In the first section of this paper, we present our design strategy and compare it to other toolkits reported in the literature. In the next sections, we detail the Application Programming Interface (API) of the toolkit in order to show its ease of learning, compactness and flexibility. We follow by a brief overview of the implementation, then we report on initial formal and informal evaluations.

Design strategy

We describe our design strategy according to the themes introduced by the work of Myers et al. [2000] on software tools for User Interface (UI) development. Along the way, we refer to the previous work in the domain in order to point to similarities and differences with our design.

PARTS OF THE USER INTERFACE THAT ARE ADDRESSED

Myers et al. observe that the tools that were the most successful are the ones that "focus on a particular part of the user interface that was a significant problem". In other words, these tools do not try to cover a wide spectrum of functions, but their focus is on the initial problem that motivated their development. This guideline is central to our design. We analyse the difficulty of implementing novel interactions and we observe that it should be relatively easy if given the proper *graphical* tools to do it. Hence, we define our "significant problem" as the one of "managing and drawing interactive graphical elements on a 2D screen for the implementation of novel GUI". This means in particular that providing software components implementing novel interaction such as Toolglass [Bier et al. 1993] is *not a primary requirement* for the toolkit: the requirement is to provide the means to *make the development of such interactions fast and easy.* Our approach is similar to the one of the Ubit [Lecolinet 2003] and TkZinc [Chatty et al. 2004, Locoanet & Mertz] that focus on providing fundamental services for drawing and interacting with basic graphical objects. It differs from approaches where the toolkit is designed to support one or more specific novel interaction or to support a broader set of services than graphical rendering. CPN2000 for example [Beaudouin-Lafon & Lassen 2000] includes support for the document structure; Jazz [Bederson et al. 2000] was designed to support Zoomable User Interfaces, and DiamondSpin [Shen et al. 2004] was design for circular interfaces.

THRESHOLD AND CEILING

Myers et al. define the "threshold" of a toolkit as "how difficult it is to learn how to use the system", and the "ceiling" as "how much can be done using the system". Obviously, the ideal toolkit provides a low threshold and a high ceiling. In practice this ideal is difficult to achieve: Myers et al. observe that low threshold toolkits tends to be low ceiling and high ceiling toolkits tends to be high threshold.

Extensive toolkits. There are two different approaches to achieve a high ceiling in a toolkit. One approach is to try to be *extensive*: that is to provide as many services as necessary to cover a wide range of needs. But this approach implies a high threshold and a steep learning curve: the novice programmer has to search for the right service to solve his problem in a large API. Becoming fluent in the toolkit takes a significant amount of time because memorizing a large API is more difficult than memorizing a small one. Another major problem of extensive toolkits appears when the developer encounters a need that is not covered by the toolkit: the specialized services of the toolkit are of no help in covering the new need and the developer has to do all the work. Traditional GUI toolkit such as Java AWT[1] or Tk [Ousterhout 1994] are typical example of extensive toolkits.

Flexible toolkits. The other approach to a high ceiling is the one of *flexibility*, which leads to the *extensibility* of the toolkit. A flexible toolkit is one that adapts well to new programmer's needs that were not envisioned when the toolkit was designed. When a new need appears, the programmer is still able to benefit from the services of the toolkit to cover the need. In addition, if the programmer's work is reusable, he has been able to actually *extend* the toolkit for this particular need. Flexibility is often achieved by a set of fundamental services that cover smaller functionality than extensive toolkit services. As the set is smaller than with extensive toolkits, the initial amount of knowledge required to start using the toolkit (i.e. the "threshold") is lower. Bederson et al. [2000] describe the extensive and flexible approach by calling them "monolithic" and "minilithic". We use the term "atomic", introduced by Lecolinet [2003] to characterize the services of a flexible toolkit.

The drawback of the flexible approach is that, for standard needs, the programmer has to assemble a set of atomic services where an extensive toolkit would provide a ready to use service. In other words, programming tends to be *less efficient* for the implementation of standard interfaces. This is not a critical problem for toolkits aimed at novel interactions because these interactions are, by definition, non-standard. Nonetheless, to cope with this

1. http://java.sun.com/

problem, flexible toolkits often provide a set of composite services made of preassembled atomic services and covering more complex standard needs. Lecolinet calls this aggregation of atomic services a “molecular architecture” [Lecolinet 2003]. These aggregated services tend to enlarge the API of the toolkit and thus make it longer to learn as a whole. But the composite services need only to be studied when specific need arise, hence the toolkit doesn’t have to be learned as a whole. Moreover, *beginning* to use the toolkit only requires to learn the atomic set of services, hence the low threshold.

We chose a design that follows the flexible approach of Jazz [Bederson et al. 2000], Ubit [Lecolinet 2003] or TkZinc [Chatty et al. 2004]. However, our analysis of these toolkits leads us to design a set of even more atomic services. For example, the three toolkits offer more than five different graphical object type where we find that two seems to be enough. The three toolkits also provide specific services for geometrical transformations where we find it more suitable to consider them as simple options of graphical objects.

Level of abstraction. The level of abstraction of the development environment in which the toolkit is used has a great influence on its threshold and ceiling. High level environments such as graphical programming environments or interpreted languages tend to facilitate the design of a low threshold toolkit because the environment itself is low threshold: programmers can learn by experimenting directly and incrementally in the environment thanks to the immediate feedback. Furthermore, high level environments tend to hide the complexity of the implementation and let the programmer express its desired goals in a more direct manner than low-level languages. The programmer doesn’t have to worry about declarations and types, he simply writes how to connect the components that make the program [Ousterhout 1998]. This supports coding efficiency: a lot is done with a few lines of code. Low-level languages tend to have a higher threshold: they require a compilation phase and have more complex programming rules. This slows down the learning of the toolkit and thus makes them less suitable.

High-level environments present two main drawbacks: the performance is low (Tcl [Ousterhout 1994] is several order of magnitude less efficient than a compiled language) and extending a high level environment may require developing the extension in a different lower-level (compiled) language. Ousterhout [1998] notes that due to the huge improvements in microprocessor performance, the part of the processing dedicated to the high-level control of the application (written in an interpreted language) has become small and thus interpreted performance has become less of an issue. The problem of extensibility is a more difficult one. Adding new functionality to a high-level environment can be done in an easy way by developing new services in the high-level environment language (we call it *high-level extension)*, or in a more difficult way by extending the environment itself in the low-level language that was used to implement the environment (we call it *low-level extension*). The latter approach is required when performance is needed, or when access to the low-level system libraries is necessary. But this low-level extension introduces a big step in the threshold of the toolkit (called a “wall” in [Myers et al. 2000]) because programmers need to change their programming language and learn new low-level concepts in order to progress. In contrast, toolkits that are programmed in a low-level language are subject neither to the problem of performance nor to the problem of extensibility. Examples of post-WIMP toolkits used from a low-level languages include DiamondSpin [Shen et al. 2004], Jazz [Bederson et al. 2000], and Ubit [Lecolinet 2003].

Because ease of learning is one of our two main objectives, we provide our toolkit in a high-level environment. We do not retain the graphical programming approach of MagLite [Huot et al. 2004] because we consider too difficult to reproduce the level of flexibility that script programming offers and that is needed for general development. Our approach is closer to the one of Pad++ [Bederson & Meyer 1998] and TkZinc [Chatty et al. 2004, Locoanet & Mertz] that are post-WIMP toolkits programmed from the Tcl/Tk [Ousterhout 1994] interpreted environment. We keep in mind that the high-level approach present a problem of extensibility:

we try to limit the need of low-level extensions of the toolkit to a minimum by carefully choosing a set of atomic services that support most extensions of the toolkit through high-level extensions. We realise that even if high-level environment are “by nature” lower threshold than low-level languages, they are less common and less taught than Java for example. This, in practice, raises the threshold of Tcl. In order to counterbalance this problem, we are considering to provide Java to Tcl translation tutorials.

PERFORMANCE

Post-WIMP interactions tend to require graphical services that are computationally expensive such as drawing semi-transparent object of complex shapes and arbitrarily deforming them. The consequence is that the lag (response time) of post-WIMP GUIs may be significantly higher that the lag of classical GUIs.

Our main target users are researchers in the domain of post-WIMP interactions. They typically need to assess the usability of their novel interaction through user studies. When measuring usability, researchers need to be confident that what they observe is an effect of their novel interaction, not an effect of the underlying graphical toolkit. In other words, post-WIMP toolkit must not introduce usability degradation because of a high lag. MacKenzie & al. [MacKenzie & Ware 1993] show that the effect of lag on usability can be measured for a lag of 75 ms but not for a lag of 25 ms. We note that the lag of our toolkit should fall in this interval, which translates to a refresh rate in the range [13-40] Hz.

In order to satisfy this requirement, we design the toolkit architecture so that the low-level rendering is isolated from the rest of the code. This facilitates the use of dedicated hardware rendering libraries such as OpenGL and the optimization of the rendering code. Our approach is similar to the one of CPN2000 [Beaudouin-Lafon & Lassen 2000] that was built from the beginning on top of OpenGL. Ubit [Lecolinet 2003] and TkZinc [Locoanet & Mertz] were initially developed using X-Window as low-level rendering library. They now offer partial support of OpenGL rendering but many operations are not hardware accelerated. Jazz [Bederson et al. 2000] and DiamondSpin [Shen et al. 2004] are built on top of Java2D which is not hardware accelerated on most platforms.

CONCLUSION

In this section, we have presented and justified our strategy for the design of the GML canvas toolkit: we focus on providing the services that manage and draw interactive graphical elements on 2D screens for the implementation of novel GUI. The toolkit aims at generality and is not oriented towards the implementation on any particular interaction. The services are offered in the form of a set of atomic services which functions can be assembled to easily implement post-WIMP interactions. The atomic services are designed to sustain the flexibility of the toolkit in order to achieve a high ceiling, but we take great care to identify the smallest possible set of atomic services to limit the breadth of the toolkit API and lower its threshold. A high-level interpreted environment is chosen to contribute to the low threshold but also to support efficient coding.

We now detail the interface through which programmers use the services of the GML canvas.

Application Programming Interface

LEVERAGING THE POWER OF TCL AND TK

The general form of the GML canvas API is largely borrowed from Tcl/Tk canvas [Ousterhout 1994] API. We present its key aspects in order to justify its suitability for a toolkit that is easy to learn and supports efficient coding.

General form of the API. Graphical objects are called *items* and are instantiated by the canvas "create" command followed by the type of the object. The command returns a unique Id for the item. In the following examples ".c" is the name of an instance of a GML canvas.

```
set pID     [.c create polygon]
```

Attributes of items are set by the use of options that, as the name implies, are always optional. We define a reasonable default value for every possible attribute so that there is always a correct minimal form of every commands, such as the one in the example above. Defining an option has the form "-<option> <value>". Options can be set when instantiating the item, or at any time using the item's "itemconfigure" method. Option's values can be accessed with the "itemcget" method:

```
set pID     [.c create polygon -shape $theShape]
.c          itemconfigure $pID -color red
set shape   [.c itemcget $pID -shape]
```

This general form of access and modification of attributes contributes a lot to the ease of learning of the toolkit: there is no need to learn many different names of method for different attributes. Furthermore, the interpreted nature of the programming facilitates the discovery and memorization of the name of item types, methods, or options: great care is taken to insure that the programmer gets an informative answer when an incorrect or incomplete command is evaluated. For example, programmers can simply call the "itemconfigure" method with no arguments to get the list of all possible attributes of an object, as well as their default and current values. A call on the "blabla" method returns an error message that lists all the correct methods of an object. In addition, error messages and default values often provide enough information to guess the form of an option's value. The interpreter, the minimal API and the informative return messages permit the "on-line" experimentation of the toolkit, which dramatically accelerates learning and support efficient coding.

Option	Meaning	Accepted values
Polygon items		
-shape	Shape of the polygon	Even list of 2D coordinates, defines the polygon local coordinate system.
-fill, -outline	Color of the interior and of the outline	{} (no fill or no outline), a color name (e.g. "blue") or an RGB value.
-fillAlpha, -outlineAlpha	Transparency of the interior and of the outline	Floating point value between 0.0 (invisible) and 1.0 (fully opaque).
-outlineWidth	Width of the outline	Floating point value, 1.0 means a width of 1 pixel.
-outlineAAlias	Anti-aliasing of the outline	Boolean value, set to true if the outline is to be anti-aliased.
-texture	A color bitmap displayed in the polygon interior with per-pixel transparency.	The name of a texture object. Texture object load bitmap data from image files.
-textureAlpha	Uniform transparency of the texture	Floating point value between 0.0 (invisible) and 1.0 (fully opaque).
Text items		
-text	The text to display	{} or a non empty unicode string.
-font	The font used to display the text	Any font string accepted by Tk, such as "Arial 72 bold".
-anchor	Position of the item local coordinate system origin with respect to the text	One of {n, s, e, w, nw, ne, sw, se, c} where "n" means "North", etc. and "c" means "center".
-color	The color of the text letters	A color name or an RGB value.
-width	The width of the column in which to wrap the text lines	0 (no wrapping), or a size in pixel.
-alpha	Transparency of the text letters	Floating point value, 1.0 means a width of 1 pixel.
-justify	Alignment in the text column	One of {left, right, center}. Only meaningful if -width is not 0.

Table 1: Options controlling the appearance of items.

Tags. Another key element of the simplicity of the Tk canvas is the use of *tags*. A tag is a simple string that can be attached to items. The only constraint is that the string must not start with a digit so that it can't be confused with an item id. An item can have zero or more tags attached to it. The tags of an item are set and queried with the "-tags" option. Tags are used to indicate on which item or set of items a method applies: a tag can be used at any place where an item id is accepted. For example the following command destroys any item that has the "layer_2" tag in its tag list:

```
.c delete layer_2
```

Tags contribute a lot to the simplicity and the flexibility of the canvas:
- grouping items does not require a new construct in the toolkit: items in the same group are simply given a common, unique tag. Groups could be used for example to control the visibility of a set of items with a simple command such as:

```
.c itemconfigure aGroupName -visible 0
```

- behaviour is attributed to a set of items by simply giving them the behaviour's tag. For example, a programmer implements the drag and drop mechanism by using the "dnd" tag instead of a reference to an item. Then, items can be set to accept or refuse the drag and drop behaviour simply by giving them or removing from them the "dnd" tag.

GRAPHICAL ITEM TYPES AND APPEARANCES

The GML canvas only offers 2 item types: polygon and text. Their appearance is controlled by the set of options detailed in table 1.

Polygon items are made of a simple polygon contour. The toolkit will soon support shapes made of multiple polygons by the addition of the **+** and **–** symbols in the coordinate list: + precedes polygons that should be added to the final shape and - precedes polygons defining holes in the final shape. Polygons can be textured. Texture coordinates are mapped to the local bounding box of the polygon. This minimal implicit mapping is sufficient to display rotated and scaled bitmaps on the screen, but we plan to support an arbitrary mapping with a `-textureMapping` option that takes a value in the form of a list of quadruplet of coordinates. In each quadruplet, one couple is expressed in the texture coordinate system and the second couple expresses the mapping of the first couple in the polygon local coordinate system. This will allow the arbitrary deformation of bitmaps for advanced visualization techniques such as fisheye views [Leung & Applerley 1994].

We only provide polygon items where other toolkit usually provide different items representing different shapes such as rectangle and ovals. This is a deliberate choice to facilitate the learning of the API by keeping it minimal: there is only one API to learn for any shape. But ease of learning should not come at the expense of coding efficiency: the script library provides a set of predefined function returning coordinates for the `-shape` option (circles, rectangles, rounded rectangles, etc.).

GEOMETRICAL TRANSFORMATIONS

The ability to geometrically transform graphical objects in various ways is one of the most frequent needs of post-WIMP GUIs (this is a key requirement for Zoomable User Interfaces [Bederson & Meyer 1998], distorted visualizations [Leung & Applerley 1994], or circular interfaces [Shen et al. 2004] for example).

In the GML canvas every item has a transformation with respect to its parent (an other item, or the canvas window). Transformations and parents are set and queried with the `-transfo` and `-parent` options. The transformation of an item defines how to transform the local coordinate system of the parent into the local coordinate system of the item. Transformations default to the identity (i.e. no transformation) and parents default to the empty value (which means that the item's parent is the canvas window). The canvas API includes the `windowtolocal` and

localtowindow methods for the transformation of 2D coordinates from the canvas window coordinate system to an item local coordinate system, and conversely.

The fact that every item has a transformation and a parent simplifies the API by eliminating the need of specialized transformation and hierarchical nodes in the scene graph such as group items in TkZinc [Locoanet & Mertz] or ZNode in Jazz [Bederson et al. 2000]. An invisible polygon item is equivalent to a transformation node with no visible appearance but controlling the drawing of its children.

Transformations are represented as a list of five floating point values { tx ty r sx sy } where (tx, ty) is a 2D translation expressed in pixels, r is a rotation in the plane expressed in radians in the trigonometric circle, and (sx, sy) are scaling factors. We chose this simple expression of transformations because it is compact and thus fast to assemble and it is intuitive: all its parameters can be directly set or interpreted, which would not be the case with a more general 2 x 2 rotation, scale and shear matrix. The drawback of this simplicity is the limit on the range of transformations that it can express. Shearing or perspective projections are examples of transformations that can not be expressed with this simple form. However, we find that translation, rotation and scaling satisfies the needs of most post-WIMP GUIs that we reviewed in the literature. Furthermore, transformations are actually implemented as 4x4 linear transformation matrices supporting 3D projective transformations of points expressed in homogeneous coordinates. Hence, the implementation supports a wide range of transformations that could be easily exposed in the interface of the toolkit if the need arise. For example, we plan to introduce an interface that applies a perspective transformation to the whole canvas in order to support displays projected on non perpendicular surfaces.

We also plan the support of multiple views on the scene through *view items*. A polygon item is made a view item by setting its –view and –viewtransfo options. The –view option receives the Id of an item (the *observed item*) which is the root of the hierarchy of item that the programmer wants to display, or re-display, through the view item. The –viewtransfo defaults to the identity and defines the observed item coordinate system with respect to the view item coordinate system.

CLIPPING, DEPTH AND LAYERS

In standard GUI toolkit, a widget is always clipped by the boundaries of its parent widget and always drawn on top of its parent (otherwise it would not be visible). In the GML canvas, clipping, depth order and item hierarchy are all independent from each other. An item can be clipped by any other item using the –clipper option, which defaults to {} (the item is not clipped). Child items are not enforced to be drawn on top of their parents. The raise and lower methods of the canvas can move any item or set of items at any place in the depth order. However, in order to facilitate the reproduction of the standard behaviour, the raise and lower methods accept the –withhierarchy boolean option which defaults to true and re-order all the descendents of raised of lowered item.

By default, items are created on top of all other items in the default layer (number 0). The layer of an item can be set or queried with the –layer option which accepts a non negative integer. For any m, n, and o such that m < n < o, all items in the layer n are drawn on top of all items in layer m but below all items in layer o. Items can also be made completely invisible with the –visible boolean option set to false. Layers are used for drawing overlays, mouse pointers, or floating windows for example.

By default, mouse events are sent to the topmost item underneath the cursor position, unless its boolean –takeevent option is set to false (in that case, the next item in depth order is considered, etc.). But any item picking strategy can be implemented with the "find at <2D point>" canvas method which returns the full stack of items under any location on the canvas. This find method can also filter the stack of items it returns according to the layer number, the visibility, or the fact that the items handle events of carry a particular tag. This is done by adding

the `-layer`, `-visible`, `-takeevent`, or `-tag` options to the `find` method call. This flexibility is required to implement Toolglasses [Bier et al. 1993] for example because both the tool item and the modified item under the cursor need to be recovered by the programmer.

We take care to offer maximal flexibility in decoupling the properties of depth, hierarchy, clipping, visibility and event handling. But we make sure that default values lead to a standard behaviour so that option's values must be provided only when a non standard behaviour is required. For example, the default of the `-visible` option of the "`find at`" command is true: the command only reports the visible items that are underneath the cursor.

This concludes the description of the low level atomic services (implemented in C/C++) of the GML canvas. They are the building blocks on top of which it should be possible to build most novel graphical interaction. At the moment, the atomic API is made of only 21 methods in the canvas, 2 item types, and 23 item configuration options.

THE SCRIPT LIBRARY

As discussed earlier, the drawback of an atomic approach is that it is often necessary to compose a large number of atomic elements in order to implement the desired interaction. In addition to the GML canvas, we distribute a set of Tcl scripts in the *script library*. The library has three purposes: it accelerates the programmer's development by providing ready to use scripts for the most common novel interactions, it acts as a tutorial of the toolkit, it represents a starting point from which developers can easily extend the library.

Shapes. The script library includes a set of parametric functions that defines standard shapes like rectangles (rounded or not), circles, and lines. Some of these functions have as little as 7 lines of code and can be easily extended to create new shapes. Also, a script that creates new shapes interactively is written in a few lines of code. It is thus very fast to implement interactive free form items such as in the MagLite system [Huot et al. 2004].

Object programming. Out of the box, the Tcl language does not include object programming mechanism. We provide a script that adds these mechanisms (classes, attributes, methods, multiple inheritance). We favour our own implementation of object orientation over popular ones such as iTcl[1] in order to keep a lightweight syntax consistent with the toolkit. Methods and attributes are dynamically defined and added when needed much like Tcl procedures and variables are dynamically added to an interpreter (i.e. there is no requirement for a monolithic class definition before creating instances). Our implementation fits in a single Tcl script file and thus it doesn't need a recompile on different architectures. In addition, its script nature makes it easy for other programmers to extend (for example by adding traces on method calls). The drawback is performance, but script classes are only used at a very high control level and we find that performance is usually not an issue at this level. A compiled version could be implemented if performance becomes an issue.

Multi-pointer management. The library includes a `gmlPointerManager` class that extends the canvas event-binding mechanism to take multiple pointer into account: events receive a pointer Id in addition to other parameters (x, y and timestamp). Instances of this class also create a network socket that listens to pointer events (Motion, ButtonPress and ButtonRelease). The network pointer events are generated by external software components that connect to pointer devices. For example we provide a small program that capture mouse events on a second machine and sends them to the machine running the pointer manager. We also interface to computer vision systems doing multi-object tracking (for tangible interfaces) and multi-finger tracking (for bare-finger interaction).

1. http://incrtcl.sourceforge.net/itcl/

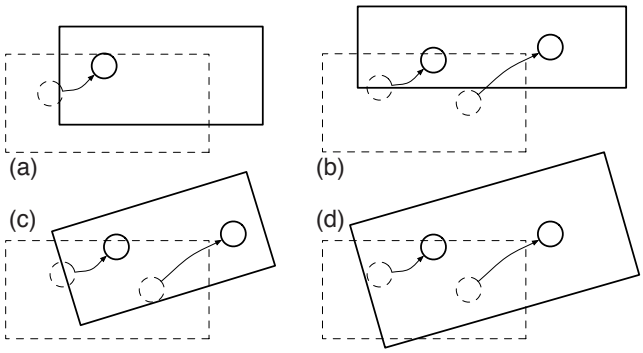

Figure 1. Deformations. Dashed lines represent the initial positions of the object (rectangle) and the control points (circles). Plain lines are final positions. a. translation. b. scaling. c. rotation. d. rotation and uniform scaling.

Deformations. We provide a unified way to control the deformation of an item with one or many control points. An object of class `gmlDeformation` manages a set of control points and a transformation in the form `{ tx ty r sx sy }` (as introduced in "Geometrical transformations"). Programmers control the deformation of an item in three successive steps:

1. The deformation is initialized by calling the `initDeform` method and passing it the initial transformation of the item, the name of the desired transformation (translation, rotation, rotation and scaling, etc.), and the initial position of the control points.
2. The positions of the control points are updated (typically in response to user action).
3. The transformation corresponding to the current position of the control points is retrieved by calling the `getCurrentTransfo` method and is used to reconfigure the `-transfo` option of the item.

We provide the following standard deformations (illustrated on figure 1): translation, scaling, rotation, rotation and uniform scaling. All but the translation require 2 control points, the translation requires only one. The control points may or may not be made visible and controllable *to the end-user*. For example, the standard scaling of a window by dragging its bottom right corner is implemented with the scaling deformation: the two control points are initialized by the programmer on the top left and the bottom right corners of the window. But only the bottom right control point is made visible and controllable to the user. This mechanism works independently of the initial transformation (i.e. if the window is initially rotated for example). A central rotation is implemented the same way by initializing the first control point on the centre of the window and giving control of the second control point to the user. Any new deformation type can be registered by providing its name and a function that computes new transformations from the initial transformation and two lists of initial and current positions of control points.

Animator. The `gmlAnimator` class facilitates the animation over time of any parameter. The programmer specifies an initial and final value (by default 0 and 1), a duration in milliseconds (by default 1000), and a callback script. The `gmlAnimator` then calls the callback regularly for the specified duration, passing it a new value of the parameter every time. For example, the following commands let an item fade in:

```
gmlAnimator anim -callback ".c itemconfigure someItem -fillAlpha"
anim start
```

The frequency of callback calls is 20 Hz by default, but another frequency or a particular number of calls can be requested. The evolution of the parameter follows a sigmoid function by default (which provides smooth acceleration and deceleration) but other functions can be requested such as the more classical linear function, or cyclic functions that never stop. The `gmlAnimator` is "fire and forget": once the beginning of the animation is requested the programmer does not need to care about the animation any more, unless he wants to stop it prematurely (which is often the case for cyclic functions). Animations are non blocking: the system continues to react during animation so callbacks must take care to check if the object

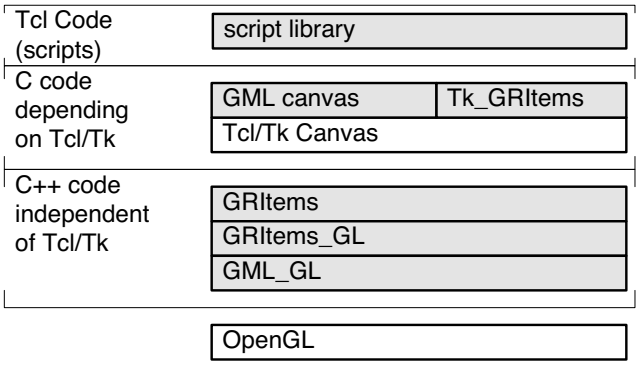

Figure 2. Layers of the GML canvas implementation. The white boxes are external libraries that our implementation rely on.

that they are modifying still exists at every step (it could have been destroyed in the meantime). Here, the introspective nature of an interpreted environment makes the task trivial.

The `gmlAnimateTransfo` callback is provided to handle the common case of animating the transformation of an item. The following command for example animates the changing of a viewpoint in a Zoomable User Interface:

```
gmlAnimator anim -callback  "gmlAnimateTransfo $rootItem \
                    [.c itemcget $rootItem -transfo] $finalTransfo"
anim start
```

At every step, the animator passes the value varying between 0 and 1 as the fourth parameter of `gmlAnimateTransfo`. This value is used to interpolate a new transformation between the initial one and the final one (2nd and 3rd parameter of the callback). This transformation is then set to the item passed as 1st parameter. In other cases, the animator callback could refer to a tag rather than an item id in order to animate a group of items.

Implementation overview

The GML canvas has been implemented on Mac OS X, X-Window (Linux) and MS Windows platforms. Sources and binaries are freely available for non-profit use[1].

We designed a layered architecture as illustrated in figure 2. The core of the toolkit is the GRItems layer (for GRaphical Items). It defines the base class that implements the standard behaviour of a graphical item: management of the item hierarchy, transformations, clipping, visibility, sensitivity to events, picking. The GRItems layer also contains the two specializations of the base class for polygon and text items and some utility classes such as the scene and display list classes.

The GRItems layer relies on the GRItems_GL layer for the rendering of the items with OpenGL. Isolating the low-level rendering code from the higher-level logic of the items has important benefits: the rendering code is more easily maintained, optimized, and shared between different item types. In addition, this architecture should facilitate the port of the toolkit on platforms that do not have an OpenGL implementation such as PDAs. The GRItems_GL layer in turn relies on the GML_GL layer for the creation and management of OpenGL drawing contexts in a platform independent way.

1. http://iihm.imag.fr/projects/gml/

This three layers constitute the low-level services that are independent from Tcl and thus do not require an interpreter. We take care to isolate all the functionality that does not rely on Tcl and Tk in order to make it available in other programming environments. In particular, this part of the toolkit has been made available in Java by researchers of our group through the Java Native Interface mechanism. Obviously, the Java version of the canvas does not benefit from the upper levels and lacks some of its key attributes: an interpreted environment, the Tk canvas tags, and the script library.

Tcl/Tk canvas extension. We modify the C code of the Tcl/Tk canvas to create the GML canvas that merges the simplicity of the Tk canvas API with the graphical power of the GRItems. The main modification concerns the Display function of the canvas in order to have it render in an OpenGL context. The C Tcl/Tk library already includes a modular API for the management of item types. We use it to replace the standard Tk canvas items by our polygon and text items. We thus keep the modifications of the Tk canvas to a minimum (represented by the "GML canvas" box in figure 2) and isolate item specific code ("Tk_GRItems" box).

Script library. Non atomic services are implemented in a library of scripts on top of the GML canvas. We have just began to enrich the script library. In addition to the services presented in the API section of the paper, we have began to implement some classes for the creation of standard widgets such as toplevel windows and buttons.

Evaluation

PERFORMANCE TEST

In order to evaluate how the GML canvas compares to other toolkits, we conducted a small performance experiment. We wrote a program that displays 200 complex polygons (stars made of 30 vertices) and rotates them. On the same machine (PowerPC G4 @ 1.7 Ghz, ATI Radeon 9600 Mobility) the resulting frame rate was 53 Hz when programmed with the GML canvas, 8.7 Hz with Java 2D, 6.5 Hz with TkZinc, and 2.4 Hz with the standard Tk canvas. This is obviously a very partial test that only gives a feel of the performance level, unfortunately we haven't had the time to carry on a full evaluation campaign.

REPORTS ON THE TOOLKIT USAGE

The toolkit has been used in different research projects while in various state of development. A first prototype was used to implement a multi-user photo layout program which purpose was to demonstrate a computer vision finger tracker. The programmer of the interaction was not directly involved in the development of the toolkit but was fluent in Tcl/Tk. He managed to rush this implementation for his paper submission in half a day (reference removed for anonymous review). The display was synchronized with the vision system and was able to keep up with the 30 Hz of the camera while showing 24 scaled photographs of about 800x600 definition on a 1024x768 display.

A more interesting case on a more advanced version of the toolkit is illustrated on figure 3. The system supports an interface that can be rotated, has animated pie menus [Callahan et al. 1988] and zoomable components. This system was entirely developed by a beginner developer that had to code everything from the atomic API because the script library didn't exist at the time. The length of development is hard to evaluate as the system is the result of many evolutions driven by user studies. But it is noticeable that a person with no particular previous knowledge

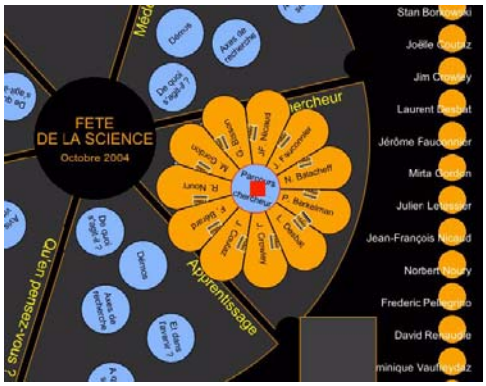

Figure 3. An example of a multi-user GUI built with the GML canvas. The big "pie slices" can controlled to rotate the interface. The first level of a hierarchical pie menu is opened. (names blurred for anonymous review purpose)

in the field was able to implement a post-WIMP interaction with minimal help. The system refresh rate remained above 40 Hz even when most of the screen needed redisplay.

The Java version of the toolkit was developed in order to support three researchers working on multi-surface interaction. Their initial prototype was build on top of the Java 2D toolkit but difficulties in implementing the deformations of windows and weak performances directed them towards our toolkit. Thanks to a suitable model of transformations and deformations, it only took them a few days to reimplement the drawing part of their demonstrator with our toolkit. The performance increased from a 5 Hz refresh rate to more than 20 Hz.

For demonstration purpose, we later decided to reimplement a multi-surface interaction system on top of the GML canvas (i.e. without Java). Only one week of development was necessary to achieve a system that allows any number of graphical displays (tested on 5) to be put together in order to built a large drawing surface. Here, the interpreted nature of Tcl was key to the rapidity of development: one interpreter running on one of the machines of the cluster acts as the *master interpreter*. Commands executed on this interpreter are actually captured and sent over an UDP multicast network packet so that any machine participating in the cluster can receive the command and *interpret* it. There is no increase of processing load when new machines join the cluster: the load is distributed to the new machines. The spatial topology of the different surfaces is trivially implemented on each surface with an invisible root item which transformation defines its position and orientation with respect to the other surfaces. Replicating the item hierarchy when a new surface joins was easy because of the introspective nature of the interpreter and the canvas (this is done in a single function with 36 lines of code).

An interesting case occurred when we attempted to implement the document passing interaction of the DiamondSpin toolkit [Shen et al. 2004]. With this interaction, users can move a document from one side of a circular table to the other side while the system will maintain a constant angle between the document and a radius from the centre of the table to the user's control point. This allows the user to pass a document to a facing person and re-orient it in a single one pointer gesture. We intended to implement a new deformation as defined in the paragraph "Deformations", but we quickly realized that this was simply the "rotation and uniform scaling" deformation where one of the control point is fixed at the centre of the table and the other is the user's control point. We simply added a circular parent item to the documents to reproduce the behaviour of the DiamondSpin table.

Conclusion and future work

We have presented the GML canvas: a multi-platform graphical toolkit facilitating the implementation of post-WIMP interactions. The toolkit offers the benefits of high rendering performance with the ease and efficiency of script programming. We propose a 2 layer Application Programming Interface with a low-level (compiled) atomic API and high-level (script) API. The fundamental services of the atomic API seem to offer great flexibility. The high-level API accelerates the development by providing ready to use composite services, and its script nature encourages the learning and extension of the toolkit. A number of systems implemented with the toolkit tend to confirm the benefits of our approach. However, more formal evaluations such as user (i.e. developer) studies or benchmarking will be necessary to assess the benefits of the toolkit.

Some of the features that we designed remain to be implemented: multi-polygon shapes, arbitrary texture mapping and multiple view on the scene. They are obviously the first candidate for the evolution of the toolkit. Additionally, while providing great flexibility, our input management services (item picking, multi-pointer event dispatch) remain at a low level of abstraction. We plan to build higher level input management in the script library using Hierarchical State Machines (HSM) such as proposed in [Blanch 2005]. Here again, it seems that the compactness of script programming makes it a natural environment for these high-level HSM descriptions.

References

Beaudouin-Lafon, M. & Lassen, H. M. [2000], *The architecture and implementation of CPN2000, a post-WIMP graphical application*. Proceedings of the 13th annual ACM symposium on User Interface Software and Technology (UIST), San Diego, California, USA, Nov. 06-08, 2000, pp. 181-190.

Bederson, B. B., Meyer, J. Implementing a Zooming User Interface: Experience Building Pad++. Software: Practice and Experience, 28(10), 1998, pp. 1101-1135.

Bederson, B. B. Meyer, J. Good, L. Jazz: an extensible zoomable user interface graphics toolkit in Java. Proceedings of the 13th annual ACM symposium on User Interface Software and Technology (UIST), San Diego, California, USA, Nov. 06-08, 2000, pp. 171-180.

Bier, E. A. Stone, M. C. Pier, K. Buxton, W. DeRose, T. D. Toolglass and magic lenses: the see-through interface. Proceedings of the 20th annual conference on Computer graphics and interactive techniques (SIGGRAPH), San Diego, California, USA, Jul. 27-31, 1993, pp. 73-80.

Blanch, R. Facilitating post-WIMP Interaction Programming using the Hierarchical State Machine Toolkit. Research report #1410, Laboratoire de Recherche en Informatique, Université Paris-Sud, France, April 2005.

Callahan, J. Hopkins, D. Weiser, M. Shneiderman, B. An empirical comparison of pie vs. linear menus. Proceedings of the SIGCHI conference on Human factors in computing systems, Washington, D.C., USA, May 1988, pp. 95-100.

Chatty, S. Sire, S. Vinot, J-L. Lecoanet, P. Lemort, A. Mertz, C. Revisiting visual interface programming: creating GUI tools for designers and programmers. Proceedings of the 17th annual ACM symposium on User Interface Software and Technology (UIST), Santa Fe, New Mexico, USA, Oct. 24-27, 2004, pp. 267-276.

Huot, S. Dumas, C. Dragicevic, P. Fekete, J-D. Hégron, G. The MaggLite Post-WIMP Toolkit: Draw It, Connect It and Run It. *Proceedings of the 17th annual ACM symposium on User Interface Software and Technology* (UIST), Santa Fe, New Mexico, USA, Oct. 24-27, 2004, pp. 257-266.

Lecoanet, P. Mertz, C. Zinc, an advanced scriptable Canvas. The pre 3.3 Reference Manual. Availiable at http://www.tkzinc.org/

Lecolinet, E. A molecular architecture for creating advanced GUIs. *Proceedings of the 16th annual ACM symposium on User Interface Software and Technology* (UIST), Vancouver, Canada, Nov. 02-05, 2003, pp. 135-144.

Leung, Y. K. and Apperley, M. D. A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM TransactIons on Computer-Human Interaction*, Vol. 1, No. 2, June 1994, pp. 126-160.

MacKenzie, I. S. Ware, C. Lag as a determinant of Human Performance in Interactive Systems. *Conference on Human Factors in Computing Systems* (INTERCHI), Amsterdam, The Netherlands, Apr. 24-29 1993, pp. 488-493.

Myers, B. Hudson, S. E. and Pausch, R. Past, Present and Future of User Interface Software Tools. *ACM Transactions of Computer Human-Interaction*, Vol. 7, No. 1, Mar. 2000, pp. 3-28.

Ousterhout, J. K. Tcl and the Tk Toolkit. *Addison-Wesley Professional*, 1st edition, March 31, 1994.

Ousterhout, J. K. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, Vol. 31, No. 3, Mar. 1998, pp. 23-30.

Shen, C. Vernier, F. D. Forlines. C. Ringel, M. DiamondSpin: an extensible toolkit for around-the-table interaction. *Proceedings of the SIGCHI conference on Human factors in computing systems*, Vienna, Austria April 24-29, 2004, pp. 167-174.

Vogel, D. Balakrishnan, R. Interactive public ambient displays: transitioning from implicit to explicit, public to personal, interaction with multiple users. *Proceedings of the 17th annual ACM symposium on User interface software and technology* (UIST), Santa Fe, New Mexico, USA, Oct. 24-27, 2004, pp. 137-146.