Annales UMCS Informatica AI X, 2 (2010) 79-91
DOI: 10.2478/v10065-010-0055-3

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Level-oriented universal visual representation environment

Leszek Rybicki*, Marta Burzańska

Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Toruń, Poland.

Abstract – We propose a three-dimensional graphics engine targeted at simultaneous visualizing multiple data sets and simulations in progress using a number of different visualization methods. The user can navigate between different views in a way in which one would traverse a museum: by switching focus from one object to another or zooming out to include several objects at the same time. Related visual-izations are vertically organized into levels or floors, further enhancing the museum metaphor. Additional information and means of manipulating the visualized data or simulations are provided for the user in a form of a two-dimensional on-screen overlay and also with the use of various input devices, not only mouse or keyboard. L.O.U.V.R.E. proved to be a very efficient and useful tool when dealing with experiments on robotics simulations. This paper presents such usage, and al-so indicates other possible applications. We find that it fills a gap as an intuitive solution encompassing graphing, simulation and user interface at the same time. Its applications go far beyond computer science research into such fields as biology or physics.

1 Introduction

This paper describes the design and implementation of a programming library for visualizing multiple data sets and simulations in a unified way using a novel paradigm. The need for the library arose during the implementation of another project, which involved a simulated robot that learned to evaluate its current state, and generate a policy using that evaluation. In short, the robot associated a value with its position, one that can be roughly associated with the distance to task completion, if starting from that state. The policy of the robot could be regarded as a vector field: for every position in space the robot had a direction of movement,

*leszek.rybicki@gmail.com

which could be visualized as a vector. Also, since the robot was steered using three neural networks, the approximation error of the networks could be also plotted in the same reference frame, since all networks had the position in space as input or part of the input.

Displaying the progress for all tasks required viewing them in the same reference frame. Since the learning process was gradual, depended on a number of parameters and required restarting from time to time, it had to be continuously overviewed and optimized. A good visualization of the robot's internal world model and policy along with the simulation of the robot itself allowed for noticing which could still be optimized, where the networks had the greatest error levels, and how the generated policy depended on the gradually improving value function.

For investigating one plot or simulation we decided to make the simulated camera hover and orbit around a focus point. If the focus point were in the center of a plot, it would give the illusion of viewing an object from all sides. In this mode the other visualizations would be hidden from the user both for speeding up the display redrawing and allowing the user to focus on the single object.

As the need for more different visualizations arose, we added the ability to place the objects, or exhibits, on different levels. One level contained the visualized output of the net-works and network errors, while another two levels were occupied by the visualizations of the robot performing different tasks. This way the environment became a multistory building in which the user could move around shifting focus from one visualization to another, or comparing several visualizations at the same time, or switching to another set of plots on a different level in space. Using the application felt like visiting a virtual museum with different exhibits and navigating around them was easy and proved useful when the results of the experiment were presented to audience.

Since there was information that needed to be visible at all times and certain elements needed to be interactively modified during the training and simulation, we added a simple GUI on a semitransparent layer on the screen. The layer contained a progress gauge for the current learning phase, a number of flat mean square error plots, buttons for starting and stopping the simulation and a map of the robot's internal model of the task set, one which could be interacted with. For documenting the project, we created a print mode in which all elements were displayed in a way that made them look better in print and added screenshot capability, along with the saving of neural network data and the mean square error history. A mode for making a number of subsequent screenshots allowed to render short movie sequences presenting the learning process.

Since it soon became apparent that the potential of the graphical environment went far beyond the needs of the project, we decided to extract the part responsible for visualization, user interface and interaction and add features that could be useful in a more general scope of use: text rendering on flat surfaces in the threedimensional environment and in the overlay GUI as wellas parametric animations. The project was dubbed LOUVRE.

We envision the use of LOUVRE as a unified threedimensional user interface for running multiple applications. The current attempts at adding a third dimension to the user interface do so by placing flat windows in space (Project Looking Glass [1], Compiz Fusion) to ensure

compatibility with the existing applications. There are few interfaces that truly leverage the illusion of space in an effective way (BumpTop [2] deserves a mention). We believe that information displayed by applications can be effectively placed in space without any loss of clarity.

Further in the paper we make the distinction between different intended users of the project: the end user and the developer, sometimes identified with the researcher. The end user is any person viewing a simulation or application written using the LOUVRE library. We talk about the end user in terms of user interface, navigation and interaction. Another example of an end user would be a student or pupil using a LOUVRE-based application as a learning tool with visualizations of geometrical forms, chemical particles or physics simulations. The developer is a person creating the application, sometimes extending the built-in capabilities to better suit the needs of the project. If the project is a scientific one, we use the term researcher.

Fig. 1. The project logo as an exhibit.

The rest of the paper is organized to describe the LOUVRE project from different points of view: the following section contains technical information about the programming library, including hardware and software requirements. Section 3 describes how LOUVRE-based applications can be interacted with – the end user point of view. Section 4 contains basic infor-mation on how to use LOUVRE for scientific visualization and how to add new types of objects. Section 5 discusses the relation of LOUVRE to other similar tools. The paper is con-cluded by a discussion of the strengths and weaknesses of this form of visualization, as well as possible applications and future development.

2 Technical description

LOUVRE has been implemented in the D language [3], which is a relatively new pro-gramming language quickly expanding its programmer base, especially among game programmers, due to its numerous features and speed of execution. The same reasons appeal to researchers, who need fast platform-native executables written with a concise and legiblle code. The D language is currently the topic of two international conferences and is taught in universities, including one in Poland – Nicolaus Copernicus University. There are two D compilers: DMD, which is made by the main developer of D. Walter Bright and GDC, which is part of the GNU Compiler Collection and has been chosen for compiling LOUVRE due to its multi-platform compatibility and standards compliance. LOUVRE works with both standard libraries available for the D language: Tango and Phobos.

OpenGL rendering, input device support and True Type font rendering in LOUVRE has been achieved thanks to SDL, or rather the D port of SDL – Derelict. SDL provides an uniform programmer's interface on a multitude of platforms. For PNG texture loading and saving PNG-formatted screenshots we used the LodePNG library. SDL needs to be installed in the end-user's system for the LOUVRE-based applications to work.

LOUVRE has been compiled and tested on Linux, MacOSX and Windows.

Three-dimensional rendering and translucency effects used in LOUVRE require an accelerated graphics card, but since unnecessary objects are hidden when possible to avoid distracting the end-user, the rendering of the ones in view is quite fast, making LOUVRE less GPU and CPU-hungry than an average modern computer game. Especially the CPU savings (rendering plots does take some floating-point calculations) are important for the projects that require CPU hungry calculations or simulations and visualization at the same time. LOUVRE has been tested (actually, in most part, written) on a Linux machine with drivers that did not take full advantage of the graphics card. In one case the Linux machine was used for calculations only and another machine was used for visuals, using a local network and the X11 technology. This method, especially useful for presentation of running simulations, gives a noticeable boost in both calculation speed and quality of the visuals.

Fonts rendered in high resolution and large images require graphics adapters with support for big textures. In a given project, it is up to the developer to keep text and textures small and compatible with a client's graphics card, LOUVRE provides no detection or warnings at the moment.

3 User interface

LOUVRE is intended to be intuitive for the end user, while freeing the programmer from the burden associated with designing an interactive three dimensional display. Instead of leaving certain choices to the developer, we decided to limit them to keep the environment consistent and predictable. A good example of this is camera control, where pitch is limited, roll is not modifiable and yaw is done by orbiting around a focus point. Camera motion will be described in detail later.

3.1 Gallery Paradigm

The three-dimensional objects in LOUVRE are meant to be viewed like exhibits in a museum or gallery. The user can focus on a single exhibit, overview a number of corelated exhibits, move among them or travel to other floors to see other types of exhibits. We believe that users will find this method of organization to be quite intuitive. Rather than having separate windows, menus or other means of switching between exhibits, the user can employ an ability that's not only already known, but arguably natural: simply explore.

It should be kept in mind that LOUVRE has been designed for viewing scientific simulations and data. A typical exhibit is intended to be a plot visualizing a data set. LOUVRE has built-in capability to visualize data sets as volume plots, vector fields, landscapes and scatterplots, depending on data dimensionality and programmer's choice. The

appearance of plots is another example of a choice that was made for the programmer. Instead of having the researcher select color settings, line width and arrow shapes, we decided to create a number of pre-set modes for different purposes, for example a dark background with semi-transparent, pseudo color plots for displaying results on a projector screen and a bright background, high-contrast wireframe plots for printing.

Yet LOUVRE was never intended to be just an engine for displaying static plots. It is optimized from the bottom-up to visualize data as calculations progress. The plots are rendered in real time using data that can change from frame to frame. Plots and other exhibits can change using a system of parametric animations that are triggered during the user's interaction. In addition, LOUVRE can visualize custom simulations, like the aforementioned moving robot. Since animation, simulation, drawing and user interface are performed by separate threads, the interaction with LOUVRE remains smooth during calculation and data update.

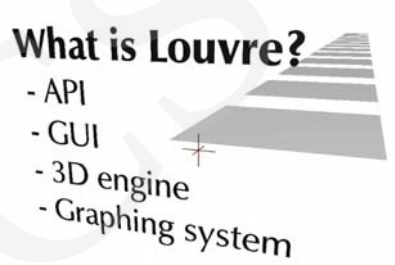

Fig. 2. A slide in the LOUVRE environment. The center point is marked and placeholders for neighboring exhibits are shown.

The end-user is also provided with a two dimensional overlay, which is meant for display elements that should always be visible. An example that keeps things within the gallery para-digm would be a map of the museum or an interactive guide. The overlay can contain buttons, progress gauges, flat plots and interactive maps. Since LOUVRE is designed to be customizable for the programmer, it's easy to design new control elements for particular projects.

3.2 Interactivity

The observer in the LOUVRE environment can move within a single floor using either the keyboard (the W, A, S and D keys) or by dragging the mouse (holding the right button) or or-bit around a point by dragging the mouse with the left button. Orbiting below a focused exhibit is not permitted (camera pitch is capped at a certain angle), but can be optionally permitted by the programmer. We propose a design guideline – if an exhibit requires viewing from below, place it above the ground.

The up and down arrow keys on the keyboard move the observer forward and backward respectively and the left and right arrow keys orbit left and right. While the observer is moving, the focus point is displayed to allow for targeting particular exhibits. The focus point is normally kept at some level above the ground, but can be moved up and down using the Page-Up and Page-Down keys. Moving the focus point up or down to another level causes the envi-ronment to display the exhibits on that level. Level height is 2 units in the LOUVRE system and the Page-Up, Down keys move the focus point by 0.5.

The observer is kept at a constant distance from the focus point, but can use the mouse wheel (or the Q and E keys) to zoom in and out. At a preset zoom level only the exhibit nearest to the focus point is displayed, allowing the observer to focus on that exhibit only, while orbiting

around it. If the observer decides to move while in a single exhibit mode, other exhibits are displayed as outlines on the floor to provide means of orientation. As the observer moves closer to another exhibit's outline, the exhibit becomes visible, while the previously focused one disappears. When the observer's camera is zoomed out to view several exhibits, still only the nearest exhibits are visible: exhibits further than 4 units from the focus point are not drawn to conserve CPU power. Also, the visible exhibits are covered by a gradually thickening fog, which can be optionally switched off. The parameters of the fog allow the observer to comfortably view several adjacent exhibits, while seeing some further ones darkened. Outlines of exhibits that are too far to be displayed are not even visible in the fog.

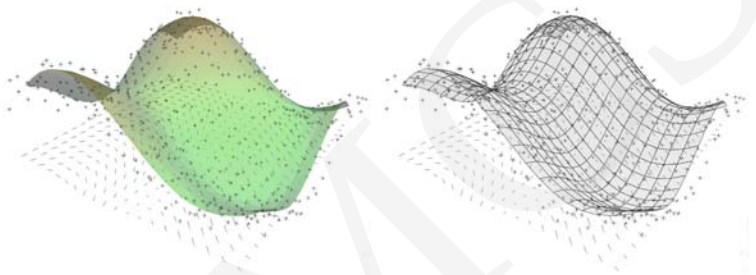

Fig. 3. Complex plot in colour and in wireframe mode. During normal execution the background for colour mode would be dark, here it is white for clarity.

Since the LOUVRE environment was designed for scientific data display, there is a built-in mode in which all objects are displayed in high contrast and with accented lines, a "publication-ready" mode. In this mode the fog and directional light are off, the two-dimensional overlay is hidden and all objects are drawn in shades of grey, where applicable. Landscape plots are displayed as wireframe to enhance readability on paper. Custommade objects should respect this mode. By default, when making a screenshot (the P key), LOUVRE switches to publication mode for the screenshot. Publication mode can be turned on and off with the I key. In the machine learning project, taking a screenshot was extended to also save data from the simulation as text files. This method of extending standard functionality is available to developers. LOUVRE can be used to make video sequences by taking screenshots of every rendered frame, but while making a video no changes are made to the display and no custom operations are permitted.

3.3 The overlay

For interactive visualizations we have designed a flat on-screen layer that displays semi translucent controls of various types. This layer is the primary recipient of mouse input in a short chain of response. As the user moves the mouse, overlay controls lighten up: depending on the type of control it can either become less translucent or display some additional data. A mouse action (click or drag) is sent to that control. If the control is designed to react to mouse clicks (like a button), the click is intercepted and the button's default action is executed. Even if an action has been executed, the button can still allow the click to be passed on to the gallery.

If a control does not implement an action (like a progress bar), the click is simply sent to the gallery.

An example of a custom control in the machine learning project was a map of the task set, which allowed to switch between tasks in the visualization: while all tasks were being trained, the user could select to visualize the progress of any of the tasks. The task map reacted to clicking, dragging and orbiting motion (the task map was actually three-dimensional itself) and the control showed additional information about the tasks when hovered over.

Since the overlay covers the whole screen, overlay objects are likely to cover objects in the three-dimensional gallery. Therefore they are designed to be translucent and as unobtrusive as possible: contain simple graphics rather than text, appear on sides of the screen, display only crucial information in their inactive state and more when hovered over.

4 API design

This section is meant to demonstrate how to use LOUVRE's to make visualization or presentation using built-in graphing capabilities and overlay components as well as extend LOUVRE.

4.1 Displaying data

Data for flat vector fields and level plots is kept in a Dataset2D structure, while data for three-dimensional vector fields and volume plots – in an analogous Dataset3D structure. Scatterplots, being simply sets of points, are treated separately, via a Scatterplot class. A Dataset, be it two- or three-dimensional, is a lattice of values. If the values are scalars, the data set will be visualized as a level plot or volume plot. If the values are vectors, the data will be visualized as a vector field.

To initialize a data set, use:

```
Dataset2D d2 = new Dataset2D(20,20,2);
Dataset3D d3 = new Dataset3D(30,30,30,1);
```

where the first set of parameters of the constructor are lattice dimensions and the last one is the dimension of the data.

In order to fill the data set object with values, use:

```
for(int y = 0; y<=d2.maxy; y++)
    for(int x=0; x<=d2.maxx; x++){
        float[] pos = d2.position(x,y);
        d2[x,y] = [pos[0]+0.05*sin(0.3*x),pos[1]+0.05*cos(0.4*x)];
}
```

The data in the Dataset(2/3)D object can be accessed in two ways: either using integer indices, which are interpreted as lattice points, or with floating point indices, which will be in-terpreted as a position in the plot, scaled to fit in a cube between -1 and 1. The easiest way to convert integer indices to the floating point position vector is with the position method, as

shown above. The position is required for the vector field: for generality, the vectors are stored in the absolute frame of reference.

Using floating point indices allows the programmer to disregard lattice dimensionality. The above could be also achieved with:

```
for(float y = -1.0; y<=1.0; y+=0.01)
    for(float x=-1.0; x<=1.0; x+=0.01)
        d2[x,y] = [x+0.05*sin(0.3*x),y+0.05*cos(0.4*x)];
```

Note that position calculation is no longer needed, since x and y are already in the absolute frame of reference. Also note that the d2[x,y] notation is the same for integer or floating point indices.

In case of data sets of scalar values, like d3 in our example, it is legal to use either of the forms:

```
d3[x,y,z] = 3.14f;
d3[x,y,z] = [3.14f];
```

but the value of d3[x,y,z] is an array of length 1.

Having filled the data set with values, we can create a plot object. Three-dimensional plots have to be put in exhibits. An exhibit can contain several subobjects called views. View objects need to implement a method called draw in which they, basically, draw themselves, but the scale and position in space of the drawing can be determined by the Exhibit object, the default being the frame of reference of the exhibit. Any plot is thus a view within an exhibit object, allowing plots to be arranged and displayed together in the same frame of reference or scaled and transformed within a view's own space.

For easy displaying of plots in the same frame of reference, we have designed the Multiplot object. The multiplot contains an array of data sets and scatterplots and displays all of them.

```
Multiplot mp = new Mulitplot();
mp.views ~= new PlotLegend("x","y","value");
mp.datasets ~= d2;
mp.datasets3d ~= d3;
mp.scatterplots ~= new Scatterplot();
scatterplot.points = [[0.0,0.2,0.5], [0.1,0.1,0.4], [0.2,-0.1,0.2]];
```

The = operator is an append operator, standard to the D language. What happens in the above lines is that we append data sets to the arrays of the mulitplot mp. Since the Multiplot is an Exhibit object, we can add views to it, in this case – a simple legend that au-tomatically labels the x, y and z axes. In order to place the multiplot exhibit in the LOUVRE space, we use a similar notation:

```
gallery.floor[0]~=mp;
```

The gallery contains one floor by default. The mulitplot will appear in the center of the floor and the camera will be looking at it when the application is initialized.

4.2 Overlay controls

Adding GUI controls to the overlay is a three step process. First, the controls need to be in-itialized:

```
Button up = new Button(vec(-550,230),30.0,
  new Texture("textures/up.png"));
Button down = new Button(vec(-550,310),30.0,
  new Texture("textures/down.png"));
```

The above code creates two buttons, places them 550 pixels left from the center of the screen and 230 and 310 pixels down. The next parameter is the scale, in this case: 30 pixels. The parameter after that can be a texture object or a string constant.

The buttons are inactive and clicking them causes no action. Assigning actions to buttons is achieved by:

```
up.onClick = &display.camera.forward;
down.onClick = &display.camera.back;
```

Fig. 4. Presentation of the 2D Overlay: two buttons, a static label and a circular progress gauge to-gether with a slide exhibit.

The display.camera is the built-in camera object, forward and back are the methods of the object. These methods, or rather delegates to those methods (the & notation) are as-signed to the buttons and will be executed when the buttons are clicked.

All that is left is to add the buttons to the overlay interface:

```
display.controls ~= [up, down];
```

The coordinate system for controls (centered in the center of the screen) may not be traditional, as most graphical environments are centered in the top-left corner, but it is consistent with the coordinate system used to place the exhibits and local coordinate systems used when drawing exhibit contents.

4.3 Slides, labels and animations

Additional information about the exhibits can be presented in the form of billboard like objects. The billboards can contain lines of text and graphics organized in rows, columns and combinations thereof, allowing for easy creation of presentation slides that will reside in the three-dimensional environment alongside other exhibits.

```
Slide introduction =
 new Slide("What is Louvre?",
 [" - API", " - GUI", " - 3D engine", " - Graphing system"]);
```

The above line creates a Slide exhibit, which will contain a title and a bullet list in a column. The Slide constructor creates Label objects out of the string constants and determines their size. The title will be automatically resized to take up the whole exhibit width and the bullets in the list will be resized to have the same font size.

Fig. 5. A complex exhibit with an animated rotating object and a column of text labels.

A Label object is a view containing either a line of text or a graphic and can be placed together with other views in an exhibit.

With complex views and labels organized into rows and columns, the LOUVRE system is a prototype of a three-dimensional user interface.

Any object in LOUVRE can be animated using Animation objects. An animation object is based on another object's parameter, which would be changed with time. The simplest form of animation changes one parameter of an object from a start value to an end value during a given amount of time. More complex animations can change multiple parameters, modify them in a loop or according to a function. All animation objects are taken care of by a separate thread, which triggers animations, progresses those already running and stops them if their end conditions become true.

5 Other visualization projects

It is not easy to compare LOUVRE to other existing projects. It is a hybrid between a 3D graphics library, a plotting engine and presentation software, while not being a swiss army knife in either of those fields. Using the same software for supervision of experiments and presentation of results along with slides and informational graphics, shortens the work flow

and appeals to the imagination of people viewing the presentation. Since LOUVRE targeted at a specific user group – scientists and engineers, we have decided to compare it to some scientific visualization tools.

One of the most popular engines for visualization is VTK (Visualization Toolkit, [5]). VTK is is an open source engine for 3D graphics, image processing and visualization with parallel rendering support and ports for multiple programming languages, including C++ and Python. Certainly a more universal and powerful tool than LOUVRE with a widespread user base and a large amount of documentation, tutorials and expert advice. In fact, VTK was considered for rendering the visuals for the machine learning project. For some researchers and programmers, including the authors of this paper, using a tool this universal is less convenient than using a simpler tool. At the level of complexity of VTK, unless advanced functionality is needed, designing a visualization that conforms to the needs of the project can be very time con-suming. VTK workflow is very low level, allowing for various forms of visualization, but burdening the programmer with technical details and taking the focus away from the experimental data or simulation to be visualized. Another difference is that VTK is meant to be embedded in an external GUI, while LOUVRE has built-in GUI capabilities.

Another visualization package we investigated is AVS/Express [6]. It is a commercial software that requires minimal coding skills from the user. Also a powerful tool, that is much easier to learn than VTK. VTK and AVS have been compared by the authors of [7]. What needs to be pointed out is that unlike LOUVRE, AVS allows only for a single thread, event based execution paradigm. Also the user is confined within the range of premade modules. They provide an easy development with generic components but the user is blocked from the low-level control over visualization. What's more – although the process of creating a visualization is much quicker than in VTK, it still is a slow and absorbing task.

Other visualization software like IBM OpenDX (which is very similar to AVS/Express), pv3 or Visad have their advantages and disadvantages. Some, like pv3, have difficulties dealing with huge scientific data. Some have parallel processing and some do not. Some are easy to set up, some are very troublesome. The way we believe LOUVRE stands out is by being a complete package for rapid visualization development that comes with a predefined set of templates that are used to build bigger structures, but it is flexible enough so that new templates can be added by a single programmer.

6 Conclusions and future plans

The LOUVRE project is based on a novel paradigm, which we believe to be intuitive for developers and end-users alike. It gracefully combines a three-dimensional simulation world with a two-dimensional GUI, while not burdening the developer with too much choice. Default settings for all objects are usually what the developer needs and what a viewer would expect. The gui is not advanced and lacks basic features found in other GUIs, but is unobtrusive and servers its purpose – to enable interaction with the gallery or display additional in-formation on the screen. While anything that can be done in LOUVRE can also be done in a number

of other more universal tools, LOUVRE allows for rapid development of aesthetically looking and practical visualizations.

There are some weak points to LOUVRE. First, the dependence on the D compiler can be problematic to many programmers. It is worth noting here that one does not need to rewrite the existing simulation code to D, as linking with C code is easy. D is needed to design the simulation environment and, if necessary, extend the default objects. Another drawback lies in the gallery paradigm itself: LOUVRE fails for viewing flat images alongside three-dimensional plots or viewing a large number of visualizations at the same time. Two applica-tions for which a set of windows would be perfect. Also, side by side windows would display three-dimensional plots as having the same scale, allowing for comparison, while LOUVRE forces a perspective and dims faraway objects. Despite these drawbacks, we believe that there are many possible scenarios in which LOUVRE can be used.

The gallery paradigm proved useful for a machine learning task and LOUVRE has been tested as a tool for displaying slides, akin to presentation software. These are two quite diverse uses that pave the way to a multitude of fields in which LOUVRE can be employed. Education has been mentioned before. Exploring the gallery requires no prior instruction for a student and any description of the objects can easily be included with the objects themselves – traits of a good learning application, an alternative to static pictures in a book. Students could learn about geometry, chemistry, physics, geography and arts, to name just a few, by observing animations prepared by a teacher.

Since the three-dimensional environment is made to resemble a building with objects placed on a grid, LOUVRE can easily be used for making interactive floor plans of buildings with detailed information about different areas in the building. The overlay can be used to display a map of the floor or controls for finding one's way around in the building. Shopping malls, airports, hotels and other places in which visitors need to find their way to a certain room, store or office would benefit from an interactive, game like display.

In the future, we plan to extend LOUVRE by embedding a physics engine that would allow for more realistic animations and would simplify making simulations. We intend to enable the use of multi-touch devices for navigation and graphical user interface. The interoperability with more input-output file formats is a must, especially the ability to load and display mesh objects.

To further facilitate the development of applications and lessen the dependency on D we plan to make most common tasks scriptable, i.e. equip the LOUVRE system with an interpreter of a scripting language or link the library to a scripting language interpreter. The first option seems most tempting, although we acknowledge the practicality of making a Python module for creating and arranging visualizations. The GUI part of the project needs more polishing: the default controls are rather simplistic and placing them in pixelexact positions is a burden. This could also be scriptable, with an automatic control positioning system. Grouping controls into sidebars is a good idea for future development. Also, any custom drawing of objects needs to be implemented in pure OpenGL. There should be a higher-level library for drawing GUI elements and custom three-dimensional objects.

To conclude this paper, we wish to express hope that the LOUVRE project gets a positive reception from the community and, through collaboration, the gallery paradigm will be adopted for different applications and more functionality will be added.

7 Acknowledgements

The authors wish to gratefully acknowledge the help of the Laboratory of Cognition and Behavior of RIKEN Brain Science Institute in which the concept of LOUVRE was born and first used during an experiment, as well as tested for presentation of experiment results.

The research was in part funded by the Polish Minister of Scientific Research and Higher Education grant N N201 385234 (2008-2010).

References

[1] Makita Yuki, Kawakami Takuro, Quang Vu, Sasaki Hitoshi, Development of an information visualization tool using the project looking glass (IEIC Technical Report) (Institute of Electronics, Information and Communication Engineers) 106(43) (2006): 15–20, ISSN:0913-5685.

[2] Keepin' it real: pushing the desktop metaphor with physics, piles and the pen, Proc. of CHI – the ACM Conference on Human Factors in Computing Systems (2006).

[3] K. Bell, L. I. Igesund, S. Kelly, M. Parker, Learn to tango with D (Apress, 2008), ISBN: 1590599608.

[4] Zeleznik R. C., Forsberg A. S., Strauss P. S., Two pointer input for 3D interaction, Proc. of the 1997 Symposium on Interactive 3D Graphics (1997).

[5] Avila W.J., Hoffman L.S., Kitware W., Visualizing with VTK: a tutorial schroeder, Computer Graphics and Applications, IEEE 20(5) (2000): 20–27, ISSN: 0272-1716, CODEN: ICGADZ.

[6] http://www.avs.com.

[7] Wilde T., Kohl J.A., Flanery R.E., Immersive and 3d viewers for cumulvs: Vtk/cavetm and avs/express, Future Gener. Comput. Syst. 19(5) (2003): 701–719.