EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

Adaptive Infrastructure for Visual Computing

K.W. Brodlie, J. Brooke, M. Chen, D. Chisnall, C. Hughes, N.W. John, M.W. Jones, M. Riding, N. Roard, M. Turner and J.D.Wood†

Abstract

Recent hardware and software advances have demonstrated that it is now practicable to run large visual computing tasks over heterogeneous hardware with output on multiple types of display devices. As the complexity of the enabling infrastructure increases, then so too do the demands upon the programmer for task integration as well as the demands upon the users of the system. This places importance on system developers to create systems that reduce these demands. Such a goal is an important factor of autonomic computing, aspects of which we have used to influence our work. In this paper we develop a model of adaptive infrastructure for visual systems. We design and implement a simulation engine for visual tasks in order to allow a system to inspect and adapt itself to optimise usage of the underlying infrastructure. We present a formal abstract representation of the visualization pipeline, from which a user interface can be generated automatically, along with concrete pipelines for the visualization. By using this abstract representation it is possible for the system to adapt at run time. We demonstrate the need for, and the technical feasibility of, the system using several example applications.

1. Introduction

Although desktop graphical capabilities continually improve, visualization at interactive frame rates remains a problem for very large datasets or complex rendering algorithms. This is particularly evident in scientific visualization, (e.g., medical data or simulation of fluid dynamics), where high-performance computing facilities organised in a distributed infrastructure need to be used to achieve reasonable rendering times. Such distributed visualization systems are required to be increasingly flexible; they need to be able to **integrate heterogeneous hardware** (both for rendering and display), **span different networks**, **easily reuse existing software**, and **present user interfaces** appropriate to the task (both single user and collaborative use). Complex distributed software systems tend to be hard to administrate and debug, and tend to respond poorly to faults (hardware or software).

In recognition of the increasing complexity of general computing systems (not specifically visualization), IBM have suggested the Autonomic Computing approach [KC03] to enable self-management, which we have used to direct our system.

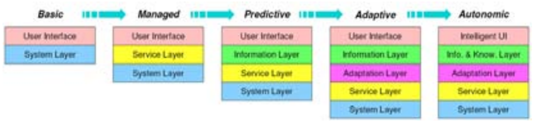


Figure 1: *The deployment model for developing a visual supercomputing infrastructure [BBC*05].*

We can think of a typical user scenario – a clinician will import a new data set into the system. The system will attempt to visualize the data based upon previous usage. A selection of images will be presented. The user will select the most appropriate, and then will describe their requirements for the system (e.g. real-time, critical, or lower quality and cheaper is preferable). The system determines the resources available to fulfil the requirement, and takes care of the visualization. The user may migrate their visualization from their desktop to their PDA, or may invite others to join in collaborative visualization. All changes are managed by the system. Also from the developer's point of view, their software can be integrated into this environment and take advantage of all these features. Our previous work [BBC*05] made an extensive study of the enabling technologies and the challenges that will be faced in implementing such a system. We proposed a model for the deployment of such a system

† K.W. Brodlie and J.D. Wood are with University of Leeds; J. Brooke, M. Turner and M. Riding are with University of Manchester; M. Chen, D. Chisnall, M.W. Jones and N. Roard are with University of Wales at Swansea; and C. Hughes and N.W. John are with University of Wales at Bangor

(figure 1) wherein we suggested that more intelligence about the system environment, user requirements and visualization can lead to a system that is able to analyse, predict and modify its own behaviour, and result in the above autonomic behaviour.

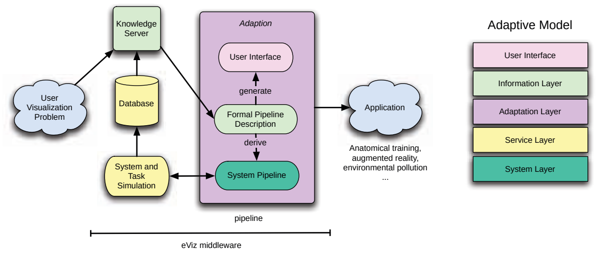


Figure 2: *Functional description of e-Viz.*

This paper presents a model of semantic data flow within a visualization system (called *e-Viz*) that will allow the system to self-configure, self-model and self-adapt its configuration subject to general goals from the user. The motivation is to simplify the user's experience of interacting with such complex visual systems, and simplify the integration cost for developers of visual software. We demonstrate the system using working prototypes, and show how easy it is to reuse and integrate existing software within the system. We present a system for the modelling, scheduling and managing of visualization computational tasks with run-time strategies for adaptability.

2. Related Work

Several visualization applications have been built using Grid services. The TeraGrid [Ter05], for example offers extremely large computational resources to scientists. Simulation and visualization is carried out on clusters and displayed locally. This is a large scale system, with large scale demands — for example integrating new software *'involves 6-10 weeks of work by 4-6 staff members'* [Ter05].

Shalf and Bethel [SB03] examine the impact of the Grid on visualization and compare pipelines running: entirely on a local PC; with part on a cluster; and with all (apart from display) on a cluster. They demonstrate that for small problem sizes the local PC will give the best performance, and thus argue that dynamic scheduling of the pipeline is required. Although their approach lacks adaptive or autonomic capability for task management, they do suggest the need for a simulation environment in the context of Grid research.

The Resource Aware Visualization Environment (RAVE) project [GAW04] implements visualization services using the Grid infrastructure. Render services are tested to check loading, and data is moved to and from render services to maintain good utilisation. RAVE supports heterogeneous render services, and display clients (including PDA).

The Grid Visualization Kernel (GVK) [HK03] supports the interconnection of the scientific visualization pipeline with Grid services. GVK is capable of dynamically changing the visualization pipeline without user knowledge, according to changing network conditions. It is implemented as modules for a number of visualization packages.

The real-time ray tracer (RTRT, or *-Ray) renders triangular meshes, volume data as isosurfaces and ray marching volume rendering [DPH*03]. Supervisors distribute tasks (image tiles) to worker nodes. They employ a hierarchical data structure for fast empty volume traversal, thus ensuring less disk thrashing. A 428MB data set of the Visible Human Female runs at 4fps on 6 nodes (each node is a dual processor 1.7GHz Xeon cpu).

Chromium [HHN*02] intercepts an application's OpenGL stream, and distributes it over a cluster of PCs with GPUs. Each GPU processes part of the image or object, and image fragments are composited to create the final image. It achieves volume rendering of 256MB of MRI data at nearly 10fps across 16 GPUs (64MB GeForce 3) [HHN*02]. Chromium relies on an OpenGL pipeline being available, and can distribute rendering using image-based or object-based approaches. A particular advantage of the system is that OpenGL applications can be integrated very easily. Integration of Chromium with the Globus Toolkit has also been demonstrated [FJ07].

The OpenRT engine from Wald et al. [WBDS03] also employs a cluster to distribute rendering. They use a demand driven tile based approach where each worker requests a tile from the supervisor. This helps with load balancing the rendering, and along with other acceleration methods (SIMD, etc.), their engine is able to provide real-time ray tracing.

The *Semotus Visum* (or remote visualization) framework described by Luke and Hansen [LH02] includes three strategies - image streaming, where all rendering is done remotely; geometry rendering, where the client has responsibility for rendering; and Ztex rendering which divides the load between client and server. Results show their framework is capable of delivering high frame rates and low latency for interaction.

Zhu et al. [ZWRI04] model the visualization pipeline as a sequence of modules, each with a computational cost, and the resources available as a set of nodes, each with a computational power. Connections between nodes are modelled in terms of bandwidth and delay. They use dynamic programming techniques to optimize the allocation of modules to nodes.

All of these systems are successful, and some show features such as dynamic load balancing (through workers requesting tiles from a supervisor), and the ability to work with heterogeneous clusters and/or diverse display clients. Our system demonstrates such features, and also enables a low integration overhead, a low configuration overhead (through the use of visualization-, user- and task- centred semantics) and by introducing introspection, adaption and self-modelling it is able to implement aspects of the autonomic model. In addition we seek to include adaptability into the user interface, to react to user behaviour. We note the earlier work of Domik and Gutkauf [DG94] in this area: they modelled the colour perception, mental rotation capability and fine motor coordination of users in order to improve the way visualizations are presented.

3. Adaptive Visualization

In [BBC*05] we proposed a deployment model for autonomic visual systems (figure 1). Basic systems provide a user interface to the visual task. Managed systems introduce a service layer (or middleware) (e.g., Grid) to manage the security, distribution, output destination and resources available to a task. Predictive systems add an information layer that can provide data about the performance of the system and quality of visualization. Adaptive systems will begin to use such data to alter their own state in order to achieve self-management, which leads on to a fully autonomic system, where a knowledge base is added to reason about the intelligence (both task side intelligence (e.g., this visualization method is the best for this kind of data), and user side intelligence (e.g., this user prefers this camera control widget)).

Figure 2 shows the functional description of *e-Viz*. We have placed the adaptive deployment model alongside, and have colour coded the principal components to match the model. The system and task simulation service layer (section 4) performs simulation using descriptions of the available hardware and proposed (or used) system pipeline. The ultimate aim of this service layer (called *SimuVis*) is to provide optimsed system pipelines for various types of visualization problems. All knowledge and information is stored in a database that can be queried using particular data types, in order to provide a valid pipeline (section 5). This pipeline is encoded using a formal pipeline description language (section 5.2), from which the user interface (section 6) and system pipeline (section 5.3) can be generated. During execution, the interface, formal description and system pipeline can be adapted to meet the user requirements (sections 6, 5.2 and 7).

4. Simulating Visualization Infrastructures

Visualization tasks are carried out on a variety of system architectures and some are executed on large infrastructures. In general, it is hard to anticipate the combined effects of many visualization systems when they interact with the underlying infrastructure, users, and one another. It is also difficult to create a variety of possible scenarios in which visualization algorithms are expected to function, and to recreate a particular condition for the purpose of testing and optimization. It is risky to install any visualization system that has not been through an adequate engineering process. It is costly to provide a replica of the live system in order to engineer and test visualization systems. One solution to such problems is simulation, which plays a vital part in a development process and enables us to determine how well a complex visualization would work without the risks and limits of testing it on a live system.

Many simulation systems have been built to assist in the engineering of computer systems. Many of these focus on emulating particular hardware and operating systems. In theory, it may be possible to build a testbed for visualization systems by running a mixture of existing simulators. In practice, such a testbed will be costly to build and maintain due to the complexity and incoherence associated with various software interfaces and data representations. In fact, such fine-grained simulation is not particularly necessary for the development of visualization systems.

4.1. SimuVis – A System for Visualization Simulation

The design of the *e-Viz* system was facilitated by the use of *SimuVis* to prototype some components. *SimuVis* is based on SimEAC [CC06], and allows the simulation of autonomic visualization systems. *SimuVis* was created as a design tool to allow rapid prototyping and evaluation of components of the *e-Viz* system.

SimuVis enables the designer of a visualization system to create an underlying infrastructure to be simulated by specifying a collection of hardware attributes, and it provides a fine degree of control over operating system and task simulation, hence allowing new algorithms to be prototyped, tested and optimized on the virtual system infrastructure. *SimuVis* supports modeling and simulation of a variety of system architectures, including large scale networked systems, in a relatively abstract manner. It provides a user interface for configuring a virtual system infrastructure, and for specifying the statistical and stochastic behavior of the system (e.g., system failures) and that of visualization tasks (e.g., dynamic job loads). *SimuVis* allows the designer to conduct experimentation on different virtual architectures without the risks of bringing down a service system, and enables scientific evaluation of typical system attributes such as scalability.

The specification of a visualization system to be simulated is divided into three layers. The bottom layer describes the capabilities of the hardware, including connections between individual hardware nodes. The top layer specifies various visualization tasks. The middle layer contains resource managers, which allocate access to hardware resources in the bottom layer to tasks in the top layer. The main technical modules of *SimuVis* include:

- *Abstract Modeling* — An XML schema is provided for specifying various hardware components and their connectivity, such as *Processing Units*, *Storage Units*, *Inter-*

connect and *Routers*. The schema also provides references to resource managers and visualization tasks.

- *Algorithmic Specification* — An API in Objective-C is provided for coding resource managers and visualization tasks, each written as a class conforming to a formal protocol.
- *Built-in Models* — To assist users with limited programming experience, a number of built-in classes have been provided, for modeling of resource managers and visualization tasks with some commonly required functionality. A GUI, also written in Objective-C, allows the built-in classes and user-defined classes to be selected for simulation, and customized with initial parameters.
- *Simulation Engine* — The simulation engine runs using discrete time intervals, known internally as ticks. The size of each tick is controlled by the user — shorter ticks produce a more accurate simulation at the expense of requiring more processor run time for the simulator. Most interaction between components in the system takes place through an abstracted message passing system.

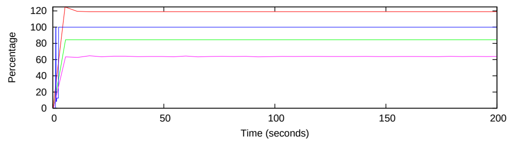
4.2. Simulating *e-Viz* Components: The Pollution Simulator

We applied *SimuVis* to the pollution simulator to be detailed in 7.1. The pollution simulator is an integrated simulation and visualization application implemented in *e-Viz* and consists of three components, namely the simulator, renderer, and client. Each component was implemented as a visualization task in *SimuVis*. The objective of the simulation was to determine how the system would function in different network configurations.

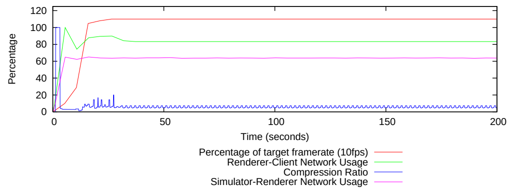
We first conducted a primary simulation running in near-optimal conditions. Figure 3 (a) shows some results collected from this simulation. Each component of the system was run on a different (simulated) machine, connected with a 100Mbit network connection. It can be seen from this graph that the application is able to meet the target framerate without saturating either network; something we already knew experimentally.

This test was used to calibrate the simulation model. Instrumentation was added to the application to determine the CPU load, simulation and rendering times, and network usage in this configuration. Once these values had been collected, they were applied to the simulation, and validated by modifying other values such as the granularity of the simulation.

Based on the calibrated simulation model, we evaluated the performance of the pollution simulator working under different conditions by modifying various parts of the specification, resulting in a performance profile for the pollution simulator. Figure 3 (b) shows the result of one such simulation. In this example, the renderer and the client are separated by a comparatively low-bandwidth Internet link.



(a) Calibration run



(b) With Internet link

Figure 3: *Simulation of Pollution Demonstrator.*

In this case, there is not enough bandwidth available to deliver the required frame rate. To overcome this problem, we added a new autonomic feature to the simulated model. The simulated renderer task detects a long delay between sending a frame and receiving an acknowledgement, and increases the amount of compression applied (indicated by the blue line on the graph). This autonomic feature is adaptive, and the compression ratio stabilizes relatively quickly to a value which allows both the desired framerate and an effectively utilized network. This analysis will drive the next revision of the pollution simulator.

5. The *e-Viz* run-time system

5.1. *e-Viz* Pipelines

The *e-Viz* system offers the user a selection of pre-defined visualization services, implemented using a range of either standalone visualization applications such as Visual Molecular Dynamics (VMD) and the Real-Time Ray Tracer (RTRT), or configurations of generic visualization systems, such as the Visualization ToolKit (VTK). We refer to a running instance of a visualization service as an *e-Viz* pipeline.

The process of initialising an *e-Viz* pipeline closely follows the gViz reference model described by Brodlie et al [BDG*04]. In the gViz model three tiers or layers of abstraction are defined; conceptual, logical and physical. The conceptual layer describes a visualization independently of the hardware and software used to run it. The logical layer additionally includes details of the implementing software, and the physical layer adds a description of the underlying hardware resources.

When creating a visualization, the *e-Viz* user is provided with a list of candidate visualizations described at the conceptual level, but the process of defining the visualization at the logical and physical levels is left to the *e-Viz* system. The user has only to choose the desired type of visualization and not how it is implemented or where it is run. We have developed an *e-Viz* remote rendering library that enables multiple servers to deliver frames to the same client window, and so can switch between alternate frame streams at runtime. Several codecs are supported, each offering different trade-offs between compression ratio, image quality, and the time taken to encode and decode. Meta-data is gathered relating to the time taken to deliver frames so that the most appropriate codec for the current environment can be selected (see also section 4.2). Servers can be added and removed at any point in the life-time of a session, meaning that an *e-Viz* visualization can seamlessly migrate from server to server.

The list of candidate visualization pipelines is identified through the use of a relational database. In the database, each conceptual pipeline is described in terms of the format and entity type of the input data. Each pipeline is then related to implementation software instances, yielding a logical pipeline. Finally each logical pipeline is related to the hardware instances capable of running the software that the individual user has access to. This gives us a description of the pipeline at the physical layer. In this way, the *e-Viz* system can inform the user which of its visualization services are available for use with a particular dataset.

This system is implemented by means of a client side wizard style launcher application, which identifies the user and his/her data type and format, and then queries the centralised *e-Viz* database through a web-service interface. The database returns a list of candidate visualization services to the user, providing a thumbnail preview image of the type of visual output that might be expected from each pipeline for a standard dataset. This database is our first step in an evolution towards the vision of a knowledge server that will 'learn' over a period of time. The user then selects the desired pipeline, and the *e-Viz* system begins the task of allocating resources and running the job.

In our implementation we used PostgreSQL for the database, WSRF-Lite for the web service interface, gSOAP to interface with the web service from the client, and QT to build the launcher application (see also [RWB*05]). We have used the *E-notation* classification scheme identified in [BCE*92] as the basis of our technique to relate input data to visualization techniques. In this scheme, the underlying phenomenon being visualized is described in terms of the dimension of the domain and the type and number of associated data fields. Visualization techniques are then classified according to type of entity that they are suitable for use with.

5.2. Formal Pipeline Description Language

At the heart of the *e-Viz* system is a formal description of the conceptual dataflow pipeline, with RDF annotations to indicate the transformation to logical and physical versions. We build on the work of the UK e-Science project, gViz, which proposed an XML language, called skML, to describe dataflow pipelines [BDG*04, DS05]. skML sees a pipeline, or `map`, as a set of `links` and `modules`, each link joining a pair of modules, from an `out-port` on one to an `in-port` on another. Modules have a `param` element, which is used to associate relevant parameters with the module. In the snippet below, skML is used to describe an isosurface rendering pipeline.

```
<skml>
  <map  id="isosurface">
    <module  id="DR" name="ReadData"
                                 out-
port="Output">
     <param name="Filename">Dglazing.dat</param>
    </module>
    <module id="IS" name="IsoSurface"
           in-port="DataIn" out-
port="Geometry">
        <param name="Threshold">65.0</param>
    </module>
        ....
    <link  id="DRtoIS">
       <module ref="DR" out-port="Output" />
       <module ref="IS" in-port="DataIn" />
    </link>
        ....
  </map>
</skml>
```

skML does more than just capture a snapshot of the system (i.e. describe the current state) - it provides a mechanism to alter the visualization pipeline that is being used. This is achieved by the provision of an action attribute to the module element which allows it to be created, destroyed or modified, and the link element which allows for connection and disconnection. These five actions in general allow for the modification of the pipeline at the module level (add a new module, make a new connection between modules etc.)

Thus we have used skML to provide a formal, non-proprietary foundation for the *e-Viz*system. Indeed we take it further than its original intent, by using it to automatically generate visualization user interfaces. This requires some small extensions to the specification, so that it can drive the selection of appropriate widgets, and so that it can support the sort of adaptivity we need. Thus we extend skML in the following ways:

- We add a `type` attribute to the parameter element of skML so that an appropriate widget can automatically be selected and presented in the user interface.
- We also add a `widgetType` attribute to the parameter element, allowing a user-specified choice of widget to be included rather than the automatically selected one.
- We add an `action` attribute to the parameter element, allowing dynamic changes to the way parameters are presented in the user interface - for example, this allows the parameter to be hidden in the interface, or visible but inactive.

The use of skML to drive the user interface generation is described in more detail in section 6.

5.3. Visualization Task Management

The first task in an *e-Viz* session is to describe the visualization that is to be performed using the launcher application (see section 5.1). Once the visualization technique and implementing hardware and software have been identified, we can create a description of the pipeline in skML. The system must then arrange for the execution of the server-side software, taking account of any server specific considerations such as batch queues and firewalls. When a visualiztion involves the use of external time-managed HPC facilities, Grid software such as the Globus Toolkit [Fos05] provides the necessary middleware functionality. In situations where a visualization is implemented using locally administered networks and machines, process management is taken care of using software agents as described by Roard and Jones [RJ06].

Visualizations and user interfaces can be reconfigured as they run in order to satisfy user defined metrics (such as performance requirements and cost functions), either autonomously by monitoring agents, or on the request of the visualization itself.

5.3.1. Self-optimizing

The system implementation of the adaptive features is greatly simplified through the use of software agents. Using relatively simple individual software agents, a more complex system can be built up. We can regard visualization software as an agent that transforms an environment into an image. This gives a simple pipeline (see figure 4 where an appropriate user interface (section 6) can produce the environment (as skML). The glue code (skeleton application as mentioned above) accepts the skML environment parameters and sets the local environment within the visualization software, and transmits the image(s) back. Progressive rendering, for example, has the same domain as the simple visualization producers – it consumes an environment, and produces an image. Internally it sends the environment to (e.g.) 3 visualization agents, but altering the image size parameter within the environment. The highest resolution, most recent image is the output sent for rendering. If a new environment is consumed, the *stop* functions are called on the higher quality agents as their images are no longer needed, and they are required to start rendering high quality images for a new view.

If the visualization producer exposes the camera environment to the *e-Viz* system, then a tile based distributed rendering can be achieved. A visualization producer agent is started on each available hardware client (up to a user or system limit). A dispatcher which has the same domain (consumes an environment and produces an image, figure 4) will send a modified environment to each visualization producer (the

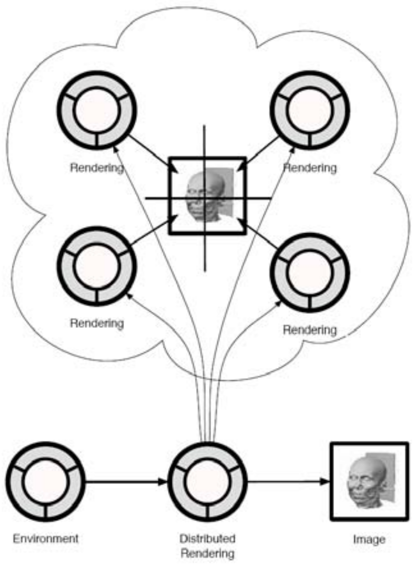


Figure 4: *Tile-based distributed rendering node dropped in to the pipeline.*

camera position, 'look at' and image size will be modifed). The dispatcher gathers the resulting image tiles, and composites them before issuing the output image.

By monitoring frame-rate, *e-Viz* can decide whether progressive rendering or tile based distribution is required to maintain performance. If required, and hardware is available, then extra threads are introduced to carry out the tile rendering, or lower quality rendering in the case of progressive rendering.

5.3.2. Self-healing

The *e-Viz* system offers a degree of self healing by monitoring active visualization components through agents using a *ping* function. If one component is lost due to a software failure or network problems, the monitoring agents forward the information to the associated pipeline, which can then switch to other similar running components (if they exist) or request new ones. Details about the monitoring and restart process are given in Roard and Jones [RJ06]. If no redundancy strategy is in place for a pipeline, a failure will be translated into a service interruption on the client side, but a replacement component will be requested and the visualization will thus restart automatically after a few seconds. This mechanism is transparent for the user.

5.3.3. Self-protection

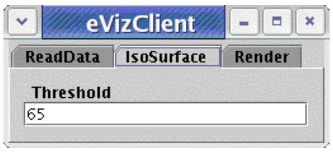
Self-protection is provided through redundancy. For example, with tile based rendering, a pool of rendering agents is used to render the tiles. If an agent fails, the tile is not returned to the dispatcher, and so will be rendered by another agent. As agents fail, performance is degraded, but as long as there is at least one agent, the visualization will not stop, as images will continue to be produced, albeit at reduced framerates. Any failing agents are restarted (self-healing), and so after a delay for initialization (reading the data), the performance will increase again.

6. User Interface Layer

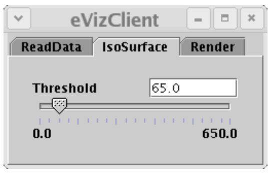
In the *e-Viz* system the visualization application is dynamically created based upon the problem posed by the user and the hardware and software systems available. In such a system it is impossible to always know in advance from what components it will be composed nor where they may be executing. Yet despite this a user will expect a consistent and appropriate user interface to be presented that is interactive and dynamically updated to reflect the changing state of the application.

6.1. *e-Viz* Client

e-Viz provides a user interface to running applications through the *e-Viz* client. This is a Java application which parses the skML description file and automatically generates a user interface that fits with the dynamically created application. It uses the gViz library [BDG*04], initially designed to support computational steering of simulations but now generalised in *e-Viz* to support general user interactions, to connect the user interface to the remotely executing components of the visualization pipeline. A tabbed interface is created where each component of the pipeline is represented as a separate tab and widgets representing its parameters are placed on its tab. Figure 5(a) shows the user interface created from the skML in section 5.2.



(a)



(b)

Figure 5: *Two images showing the same tab of the user interface, (a) is using orginal skML from section 5.2 (b) is generated from the extended skML.*

As can be seen from figure 5(a), only text boxes can be created from this description. With the simple extension of adding a type attribute to the parameter element of the skML more appropriate widgets may be chosen. In this example a `type="double"` is added for Threshold parameter so the text typein box can be replaced with a slider that generates a floating point number (figure 5(b)).

Whilst most elements of a user interface are for input from the user, some are for output only, e.g., the `Current Time param` element from the simulation in section 7.1. Thus the `param` element has an interaction attribute whose values can be `steer` or `view` and the *e-Viz* client will react accordingly, setting view parameters to be non-editable.

6.2. Reactive User Interfaces

A novel contribution is the ability for the user interface to be dynamically modified according to a range of external factors. This is seen as fundamental for the *e-Viz* approach of an intelligent visualization environment. The dynamic interface is achieved by a further extension of skML, allowing run-time modification of the dataflow pipeline and module parameters. skML contains the basic mechanism for changing the pipeline at the map and module level, but is not efficient for representing changes to parameters or even capable of some subtle modifications to the user interface.

The interface can react according to a range of influences:

User Behaviour: It is useful to reduce the cognitive load on users by dynamically modifying the list of options shown to users based on their past behaviour. For example if a user never modifies the decimate parameter for a marching cubes module then its widget can be hidden from the control panel.

```
<map id="MCMap" style="left:10;top:10;
                              color:#d4d4d4">
  <module id="2" name="MarchingCubes"
                             action="modify">
     <param name="Decimate" action="hide" />
  </module>
</map>
```

Here we have added an `action` attribute to the `param` element which allows modifications at the parameter level.

Application Behaviour: In computational steering, for example, there are typically parameters associated with the setting up of the numerical solution, that must not be changed during the course of a run. An example might be the grid size over which the solution is computed; or some parameter defining the problem, such as temperature under which a simulation is carried out. In contrast, other parameters (such as frequency of output of results) are changeable at any time. Thus the interface needs to react in response to information from the simulation, concerning its present state: for example, is it at its initial step or midway through?

```
<map id="PollSim" style="left:10;top:10;
                                color:#d4d4d4">
  <module id="2" name="Simulation"
                               action="modify">
```

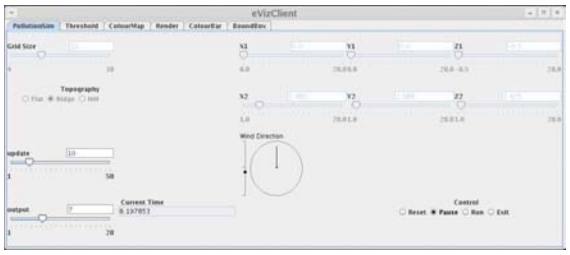


Figure 6: *Personalised Interface.*

```
    <param name="Grid Size" ac-
tion="disable" />
  </module>
</map>
```

Here we have set the `action` attribute to be `disable` to allow the *e-Viz* client to display this widget, but indicate that it is not currently active.

6.3. Personalised Interfaces

It is important to be able to adapt the user interface to the experience and preferences of individual users. We address this in two ways in the *e-Viz* system. Firstly, the user can create a `.evizrc` file in which they can record preferences for default settings. This can also record recently unused widgets so that a compact version of the interface can be displayed as an option (as discussed above). Secondly, the user can provide their own widgets which will then replace the widgets automatically selected by the system. An example of this is illustrated in figure 6, which shows a further version of the interface for the pollution demonstrator described in section 7.1. In figure 6, the original sliders that provide wind direction control are replaced by a simple 2D compass type widget plus an elevation control.

6.4. Communication Layer

The *e-Viz* client must be capable of operating in conjunction with a wide variety of visualization and simulation software. Thus it contains, in addition to the user interface presentation, a communication layer which allows it to connect to external software. The architecture is shown in figure 7. The client defaults to using the built-in gViz library for parameter communications, but other mechanisms can also be used. An interface is provided that allows other users to write plugins to support their own protocols, converting this to the eViz parameter descritpion before passing it to the client. These plugins are loaded at run time using directions passed in the skML RDF description of their module(s). In practice we have built applications where the communication is by the gViz protocol itself (see section 7.1), by the RealityGrid steering library protocol [PHPP04] (where *e-Viz* was

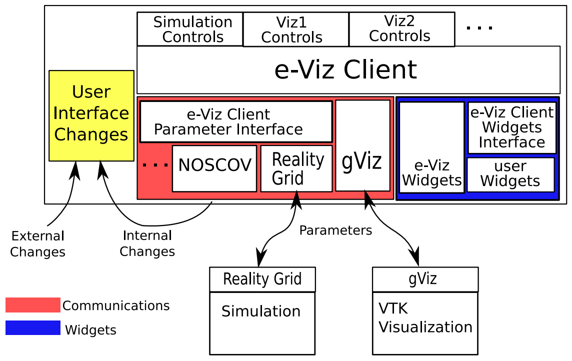


Figure 7: *User Interface Architecture.*

used to visualize live data output from a molecular dynamics code) and by a protocol used to communicate between the *e-Viz* client and a set of visualization web services in the NOSCOV system [WBHW06]. Moreover the architecture allows us to support different protocols within the same running system, using say, gViz connection to visualization software and RealityGrid connection to simulation software. This is a further contribution to minimizing the integration effort to incorporate *e-Viz* into an existing system.

The architecture diagram also highlights the two ways in which the user interface can dynamically change. This can either be in response to a user interaction (for example, moving a slider) or in response to a message from an application (for example, changing a parameter from being active to inactive), or from messages from the *e-Viz* system.

7. Application

7.1. Environmental Pollution

One of the motivating scenarios for the gViz project [BDG*04] was that of an environmental pollution disaster. In this scenario an accidental release of a chemical has taken place and it is necessary to compute in faster than real time the concentrations of the pollutant in the environment to inform decisions with respect to evacuation of population centres. The pollutant is moved under the action of the wind which may change as time progresses and alter the levels of concentration in different locations. This scenario is modelled using a PDE based numerical simulation generating data that is visualized and presented to the user in real time. The user is able to change the direction of the wind while the simulation is running and see the effects of these changes as they are computed. In gViz a number of different visualization systems were used as clients, each one requiring a hand coded user interface to be created.

Figure 8 shows the automatically generated user interface for the same pollution application. In the *e-Viz* framework the system is responsible, using high level directives from the user, for generating an appropriate visualization pipeline and representing this using skML. Using this description the various components that make up this pipeline can be launched and connected. This skML description is passed to the *e-Viz* client which parses it to dynamically create an appropriate user interface. For each parameter it selects an appropriate widget; the selection is initially based on its type but can be refined on the basis of special widgetTypes specified within the skML, or by local user preferences, or by the nature of the client device. In the application shown in figure 8 the widget types have been selected based on their data type. The *e-Viz* client uses the gViz library to connect itself to the individual components of the pipeline to send and receive parameter changes.

In the environmental pollution system's initial design, the simulation was embedded as a module in IRIS Explorer with the data passing directly down the map for visualization. With this tight coupling, the user interface could be directly manipulated by the simulation module's code allowing it to disable widgets for parameters that were not steerable at certain times. The next step in the evolution of the demonstrator moved it into the more generic steering framework of gViz. This allowed the simulation to be run on a remote high performance machine independent of any visualization system but removed its ability to manipulate the user interface. By contrast, in the *e-Viz* version, skML snippets, (section 6.2) are used by the simulation component to influence the state of its user interface to change the state of its initialisation parameters from active to disabled while running and back to active upon reset.

8. Discussion

By exploring the Autonomic Computing approach, we have demonstrated how it could impact on the way that visualization services are implemented and presented to users and developers. We have taken our proposal of a deployment model for a visual supercomputing infrastructure [BBC*05], and have carried out a thorough analysis of how it could be achieved with a functional implementation. We presented the implementation of the infrastructure in figure 2 and section 3. Important aspects of the infrastructure include:

- a task simulation engine (provided by *SimuVis*) for evaluating and optimizing visualization task configuration within our environment;
- a knowledge server which can map user requirements and data to formal visualization and system pipelines;
- the specification of an abstract pipeline language;
- semantics for ease integration of existing software;
- software agents for monitoring and handling visual tasks;
- reactive user interfaces with dynamic modification based upon the formal abstract pipeline language;
- and run-time strategies for adaptability.

Using such an infrastructure, we have been able to make reasoned decisions about a visualization task based upon — cost, speed, quality and reliability, and thus how to schedule it over available resources. Other example applications have already been shown to benefit from this approach [HJ07].

Such scheduling decisions, task simulation, low software integration costs, and adaptability are going to be increasingly important as we enter a period where increases in processing power come through increasing the number of cores rather than clock speed. The size of scientific data sets can also be predicted to grow and become even more reliant on visualization for efficient analysis. The infrastructure we present here, *e-Viz*, provides many of the solutions that we will need.

References

[BBC*05] Brodlie K., Brooke J., Chen M., Chisnall D., Fewings A., Hughes C., John N. W., Jones M. W., Riding M., Roard N.: Visual supercomputing – technologies, applications and challenges. *Computer Graphics Forum 24*, 2 (2005), 217–245.

[BCE*92] Brodlie K. W., Carpenter L. A., Earnshaw R. A., Gallop J. R., Hubbold R. J., Mumford A. M., Osland C. D., Quarendon P. (Eds.): *Visualization Techniques*. Springer-Verlag (Berlin), 1992, ch. 3, pp. 37–85.

[BDG*04] Brodlie K., Duce D., Gallop J., Sagar M., Walton J., Wood J.: Visualization in grid computing environments. In *Proceedings of IEEE Visualization 2004* (2004), pp. 155–162.

[CC06] Chisnall D., Chen M.: The making of SimEAC. In *International Conference on Autonomic Computing* (2006), pp. 301–302.

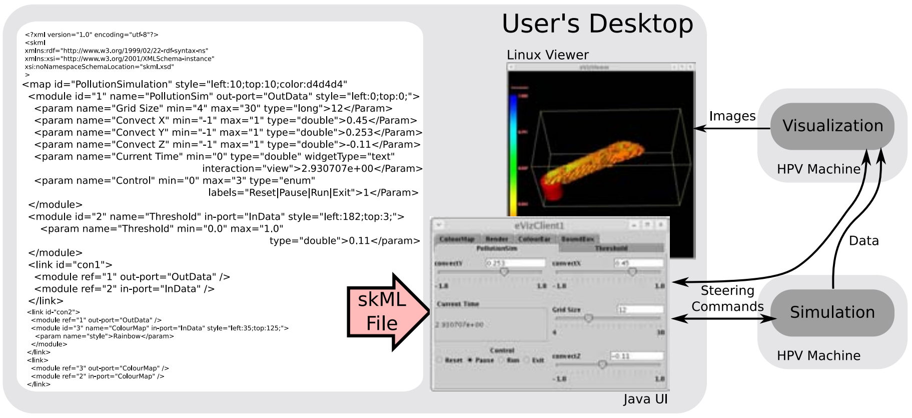
[DG94] Domik G. O., Gutkauf B.: User modeling for adaptive visualization systems. In *Proceedings of the conference on Visualization '94* (1994), IEEE Computer Society Press, pp. 217–223.

[DPH*03] DeMarle D., Parker S., Hartner M., Gribble C., Hansen C.: Distributed interactive ray tracing for large volume visualization. In *IEEE Symposium on Parallel Visualization and Graphics* (October 2003), pp. 87–94.

[DS05] Duce D. A., Sagar M.: skML: A markup language for distributed collaborative visualization. In *Proceedings of Theory and Practice of Computer Graphics 2005* (2005), Lever L., McDerby M., (Eds.), Eurographics Association, pp. 171–178.

[FJ07] Fewings A. J., John N. W.: Distributed graphics pipelines on the grid. *IEEE Distributed Systems Online 8*, 1 (2007).

[Fos05] Foster I.: Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of the IFIP International Conference on Network and Parallel Computing* (2005), pp. 2–13.



```
<?xml version="1.0" encoding="utf-8"?>
<skml
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="skml.xsd"
>
<map id="PollutionSimulation" style="left:10;top:10;color:d4d4d4"
 <module id="1" name="PollutionSim" out-port="OutData" style="left:0;top:0;">
  <param name="Grid Size" min="4" max="30" type="long">12</Param>
  <param name="Convect X" min="-1" max="1" type="double">0.45</Param>
  <param name="Convect Y" min="-1" max="1" type="double">0.253</Param>
  <param name="Convect Z" min="-1" max="1" type="double">-0.11</Param>
  <param name="Current Time" min="0" type="double" widgetType="text"
                                   interaction="view">2.930707e+00</Param>
  <param name="Control" min="0" max="3" type="enum"
                                    labels="Reset|Pause|Run|Exit">1</Param>
 </module>
 <module id="2" name="Threshold" in-port="InData" style="left:182;top:3;">
   <param name="Threshold" min="0.0" max="1.0"
                                              type="double">0.11</param>
 </module>
 <link id="con1">
  <module ref="1" out-port="OutData" />
  <module ref="2" in-port="InData" />
 </link>
 <link id-"con2">
  <module ref="1" out-port="OutData" />
  <module id="3" name="ColourMap" in-port="InData" style="left:35;top:125;">
   <param name="style">Rainbow</param>
  </module>
 </link>
 <link>
  <module ref="3" out-port="ColourMap" />
  <module ref="2" in-port="ColourMap" />
 </link>
```

Figure 8: *The skML description is processed by the e-Viz Client to generate the shown user interface. The widgets shown have been selected based on their type in the skML description, except for the Current Time which has been refined by the use of a widgetType=text attribute and it is non-editable as it is a view only parameter. The e-Viz Client connects to remote components using the gViz library enabling the simulation to be steered according to the wind direction set by the user and the images to be delivered to the viewer using the e-Viz Viewer.*

[GAW04] GRIMSTEAD I. J., AVIS N. J., WALKER D. W.: Automatic distribution of rendering workloads in a grid enabled collaborative visualization environment. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (2004), p. 1.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02* (2002), pp. 693–702.

[HJ07] HUGHES C. J., JOHN N. W.: A flexible approach to high performance visualization enabled augmented reality. In *Proc. of Theory and Practice of Computer Graphics 2007* (2007). To Appear.

[HK03] HEINZLREITER P., KRANZLMÜLLER D.: Visualization services on the grid: The grid visualization kernel. *Parallel Processing Letters 13*, 2 (2003), 135–148.

[KC03] KEPHART J. O., CHESS D. M.: The vision of autonomic computing. *Computer 36*, 1 (2003), 41–50.

[LH02] LUKE E. J., HANSEN C. D.: Semotus Visum: a flexible remote visualization framework. In *VIS '02: Proceedings of the conference on Visualization '02* (2002), pp. 61–68.

[PHPP04] PICKLES S., HAINES R., PINNING R., PORTER A.: Practical tools for computational steering. In *Proceedings of UK e-Science All Hands Meeting* (2004).

[RJ06] ROARD N., JONES M. W.: Agents based visualization and strategies. In *Full Papers Proceedings of WSCG 2006* (2006), pp. 63–70.

[RWB*05] RIDING M., WOOD J., BRODLIE K., BROOKE J., CHEN M., CHISNALL D., HUGHES C., JOHN N., JONES M., ROARD N.: e-Viz : Towards an integrated framework for high performance visualization. In *Proceedings of UK e-Science All Hands Meeting* (2005).

[SB03] SHALF J., BETHEL E. W.: The grid and future visualization system architectures. *IEEE Comput. Graph. Appl. 23*, 2 (2003), 6–9.

[Ter05] TERAGRID:. `http://www.teragrid.org/`, 2005.

[WBDS03] WALD I., BENTHIN C., DIETRICH A., SLUSALLEK P.: Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. *Lecture Notes on Computer Science 2790* (2003), 499–508. (Proceedings of EuroPar 2003).

[WBHW06] WANG H., BRODLIE K., HANDLEY J., WOOD J.: Service-oriented approach to collaborative visualization. In *Proceedings of UK e-Science All Hands Meeting* (2006).

[ZWRI04] ZHU M., WU Q., RAO N. S. V., IYENGAR S. S.: Adaptive visualization pipeline decomposition and mapping onto computer networks. In *3rd International Conference on Image and Graphics* (2004), pp. 402–405.