Agents Based Visualization and Strategies

Nicolas Roard
Swansea University
csnicolas@swansea.ac.uk

Mark W. Jones
Swansea University
m.w.jones@swansea.ac.uk

ABSTRACT

This paper describes a flexible visualization architecture based on software agents, which enables the abstraction and reuse of rendering strategies. Using a reification of the rendering environment, the system is able to add new rendering strategies (such as distributed rendering or progressive rendering) to an existing pipeline, without any modification of the other components (controls components, display components, rendering algorithms, *etc.*). The ability of changing strategies on the fly leads to a better adaptability to runtime constraints. The system uses an agent-based graphic pipeline, where each agent/component can be located on different computers; communications between agents use XML/RPC and data stream in order to easily integrate existing code in the system. Agents can add specific behavior to graphic pipelines, such as saving environments to reuse them, adapt information and knowledge from another pipeline, and generally modify and improve the entire system. Various visualization and control clients exist, enabling collaboration between platforms such as PDAs, Windows, Linux, MacOS X, and Web (using Java applets).

Keywords Distributed Visualization, Software Agents, Volume Rendering

1 INTRODUCTION

Although individual graphical capacities continually improve on workstation or desktop computers, visualization at interactive frame rates is still a problem with very large datasets or complex rendering algorithms. This is particularly evident in scientific visualization, such as medical data or simulation of fluid dynamics; high-performance computing facilities organized in a distributed infrastructure need to be used to achieve reasonable rendering times in those cases [HEvLRS03, BBC+05].

Such distributed visualization systems are required to be more and more flexible; they need to be able to integrate heterogeneous hardware (both for rendering and display), span through different networks or the internet, and easily reuse existing software. They also need to allow complex user interactions like collaboration [MF00], and easy customization to answer specific needs.

Complex distributed software systems tend to be hard to administrate, and tend to respond poorly to faults (hardware or software). The Autonomous Computing grand challenge initiated by IBM [KC03, IBM] aims to build software systems that are as autonomous as possible to simplify implementation and administration.

WSCG 2006 conference proceedings, ISBN 80-86943-03-08
WSCG'2006, January 30 – February 3, 2006
Plzen, Czech Republic.

Our system is a reflective middleware [KCBC02] based on agents, which we used to implement a distributed graphic pipeline. We worked on a Volume Rendering pipeline, as Volume Datasets by nature tend to be both large and slow to render, and are thus good candidates for testing a distributed visualization system.

We will first review related systems and present the general architecture of our system. We will then present some examples showing the reflective nature of the system, first by talking about the Visualization Strategy pattern, then by introducing added behavior to generic pipelines.

2 RELATED WORK

Sharing computing resources is an old idea in computer science, but the advance of Internet and Web Services now permit the building of interoperable, cross-platform distributed services. The Grid Initiative [FK99, FKT01] and projects like the Globus Toolkit [FK97, Fos05] provide toolkits and infrastructure to deploy such grid computing systems.

Visualization systems using the grid have their own set of requirements [SWB03, BBC+05], and are an ongoing research subject (see [ADK+99, BDG+04, KHRV04, GAW05, RWB+05] for some of the existing visualization systems using the grid). These systems implement a distributed graphic pipeline following the established modular dataflow model [UFJK+89, Dye90, Cam95] using grid technologies.

Multi-Agent Systems (MAS) are a particular architecture of distributed systems. They tend to be more flexible and reliable than traditional distributed systems, as Software Agents can work and react at a local level, providing a decentralized intelligence. As such, they

seem an interesting research idea to obtain autonomic systems, and more reliable and flexible distributed systems [JJZ+04, ZM05, GFB05, MGR+99].

Some research projects follow an interesting approach, combining an agent system with a distributed rendering system [HEvLRS03, RKACM03].We followed a similar direction, although our orientation is slightly different than those systems. We wanted a flexible research tool, with reflective capabilities [KCBC02, ASA01], to experiment how we can use thoses capabilities to build intelligent applications.

3 GOALS AND PHILOSOPHY

Our principal goal was to obtain a flexible system for experimenting with ideas and architectures. We believe that using loosely coupled, dynamic systems and languages, leads to simpler and more powerful systems, which are easier to prototype.

The design thus focused on building a layered system (Figure 1), with each layer kept simple.

7	Autonomic System
6	Intelligent Applications
5	Reflective System
4	Graphic Pipeline
3	Agents
2	Distributed System
1	Processes

Figure 1: System layers

Remote Processes (1) interact in a Distributed System (2) by sending messages. On top of this Distributed System we build an Agent infrastructure (3) – in fact we consider everything to be an agent in our system, which gives the next layers a range of useful functionalities (creation, discovery, *etc*.). Using Agents, we then implemented a graphic pipeline (4). Agents are aware of their environment and can modify it, in essence creating a Reflective System (5). The agents environment comprises not only the graphic pipeline, but also the complete system. We then take advantage of this Reflective System and the autonomic nature of Agents to build Intelligent Applications (6). Our final objective is to build an Autonomic Visualization System based on the existing layers (7).

Our current interest and what we describe in this paper is in exploring the Intelligent Applications layer – how can we take advantage of the Reflective nature of the system to build applications or interesting architectures.

4 SYSTEM ARCHITECTURE

4.1 Core System

The core system is a very simple distributed architecture, composed of a naming service and agents, where agents communicate using XML/RPC [Win99]. Each agent is associated to a quintuplet *{location, port, name, type, status}*. Ports can be shared among agents.

Using XML/RPC gives us a simple solution for sending remote messages, which uses very common standards (TCP/IP, XML, HTTP). An immediate benefit of this simplicity is the fact that most mainstream computing languages have a working implementation of XML/RPC (*e.g*. [Mül03, ASF05]). It is therefore a simple task to transform existing code into a component of our system.

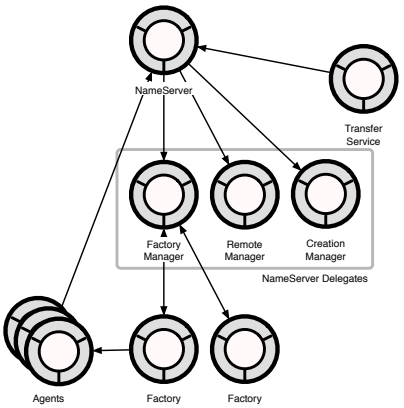

Figure 2: Architecture overview

An agent can be duplicated easily or moved to another machine, which gives us a very flexible network topology. Furthermore, the system is very dynamic and reflexive, as agents can modify the system at runtime, request information or meta-information about the system or the graphic pipeline, and add new information to the system.

Figure 2 shows an overview of the basic architecture. On each machine we have a NameServer which keeps track of all the objects on the local machine and of remote NameServers. A Transfer Service agent can be used by agents to move data from one machine to another. Factories (Agents that create other agents) are handled via a Factory Manager. The Remote Manager agent is used to manage a network of machines and forward requests. The Creation Manager is in charge of managing the population of agents on the available machines. The following sections detail each component.

4.2 Naming Service

A new agent which wants to register with the system contacts the local Naming Service (using a default port). The Naming Service then returns an available local port to the agent. The agent can then initialize and finish its registration.

4.2.1 Extending the Naming Service

Additional agents can register to the local Naming Service as delegates, providing a Naming Service themselves. Then, in case of an unsuccessful request to the local Naming Service, these agents are called and can answer the request (see Sections 4.4.1 and 4.5). In our current systems those delegates are the Factory Manager, the Remote Manager and the Creation Manager.

4.3 Transfer Service

The Transfer service is an agent providing simple transfer methods, able to send data/binaries from a machine to another. It is needed by the Replication mechanism as well as Data management.

4.4 Agents

Agents are autonomous software that can interact with their environment. In our system, we also need to be able to replicate agents easily and create them on demand.

4.4.1 Agents Creation: Factories

Factories create new agents of a certain type on demand. Factories are implemented as normal agents answering a certain protocol (a protocol beeing a set of messages) with the type FACTORY. A Factory Manager agent registers as a delegate to the NameServer. Each Factory is automatically registered to it.

When an agent is requested, the NameServer searches to see if an available instance exists in the system. If not, the request is delegated to the Factory Manager, which in turn calls the corresponding Factory and asks it to create a new agent. This new instance is then registered to the NameServer and returned.

A Factory cannot create an agent if the system is overloaded (in which case other machines can possibly handle the creation).

4.4.2 Replication

The process of replication of an agent is twofold; first, we consider the agent binary (or sourcecode if the agent is a script). Second, we also need to replicate the agent's state if we want to move an agent rather than create an agent of the same type. Replication helps with the autonomic aspect of the system – i.e. self repair.

In our current system the agent's state is determined by the pipeline and the environment (see Section 4.6), so we do not need to manage the serialization of the agent's state. We only need to register the new agent to the pipeline, and restart it. The start method in the agent will use the pipeline to initialize itself.

4.5 Managing Multiple Machines

Using Factories and Replication, we can have a remote creation process to use multiple machines in our architecture. The Creation Manager agent use the Transfer & Replication services to create new Factories on available machines, depending on their state (*e.g.* CPU load, disk space).

The naming mechanism is extended via a NameServer delegate (similar to the Factory Manager), called the Remote Manager. Machines are organized in a hierarchical way (Figure 3) where each child machine registers to the Remote Manager.

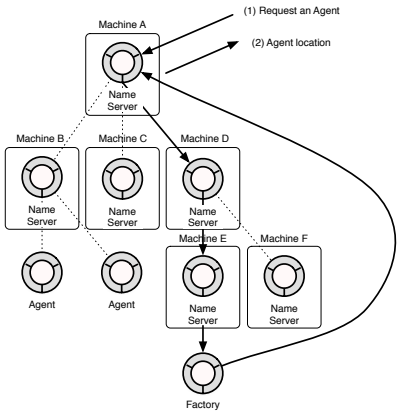

Figure 3: Remote creation

When the NameServer gets a request for an available agent, and none is found locally, it forwards the request to its delegates. If a delegate doesn't return an agent's address, the next delegate is called. Figure 3 shows this mechanism.

For example, if the Factory Manager doesn't return an agent (because it doesn't have a corresponding factory, or because the maximum number of agents is reached, or the CPU load is too high), the Remote Manager is called and try to answer the request. The Remote Manager maintains a list of the machine's children, and a cache of their factories types. It then forwards the request to a child having the corresponding factory. If the Remote Manager itself can't handle the request (*e.g.* no corresponding factory) the Creation Manager is called and using the Transfer & Replication service can create new Factories.

4.6 Visualization Architecture

We implemented a distributed graphic pipeline using our system, where each pipeline component is an agent. Figure 4 shows the general structure of a pipeline.

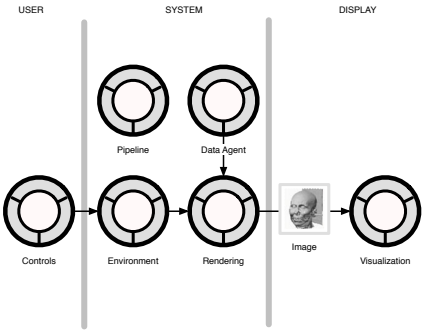

Figure 4: Visualization System Architecture

A Rendering agent gets its data information from a Data agent, and uses a "rendering environment" given by the Environment agent to render an image. The image is then sent to a Visualization client.

The Environment contains all the information for the rendering – quality, specific settings, camera position,

etc. – and can be modified by other agents, like a Control agent to move the camera.

The Visualization and the Controls are in general simple clients to the system, not real agents, and thus can be completely externalized, which permit for example to run them through a web page (using applets), or in a client application (Section 4.8).

Other agents in the pipeline need to be fully integrated (shown on Figure 4 by the gray outline) in the system. Agents in a pipeline are registered in a Pipeline agent and can be reached and manipulated by other agents through it.

Agents communicate by sending XML/RPC messages (synchronous and asynchronous) and direct socket transfer for transmitting data. The overhead of XML/RPC is not a problem in general, specifically by using asynchronous messages whenever it's possible[1] and limiting the number of necessary messages in the architecture.

4.7 Data Management

Data needs to be duplicated and moved along with the rendering components of the pipeline. Data agents are in charge of that task, and use the transfer facilities to move or duplicate a dataset to another machine. In the future we envision intelligent agents and data placements using planning agents, but for the moment the data management simply transfer data from the original pipeline to cache it locally.

4.8 Clients and User Interface

As explained in Section 4.1, XML/RPC permitted a quick development of the system and an easy integration of existing components. A good example is how we were able to program visualization and control clients for various platforms, taking advantage of specific languages/frameworks allowing quick prototyping. We used GNUstep [FSF] and Cocoa [App] frameworks to program a crossplatform Objective-C/OpenGL client running on Windows, MacOS X and Linux. Squeak [Squ] (a Smalltalk environment) was used to prototype the system and provides a PDA implementation, and Java was used for the web client. Figure 5 shows the PDA and Java clients, while Figure 6 shows the Squeak environment used for prototyping. Figure 7 shows the GNUstep client.

In parallel to this project, we participate in another project which will integrate this agent system using the grid toolkit in order to access the security framework provided by that system.

4.8.1 Modifying the User Interface

One problem with adding unplanned, "intelligent" functionalities to a graphic pipeline as proposed in Section 7 is obviously that the user interface needs to be changed

[1] while XML/RPC doesn't specify asynchronous messages, many implementations provide them, as it is a very simple modification

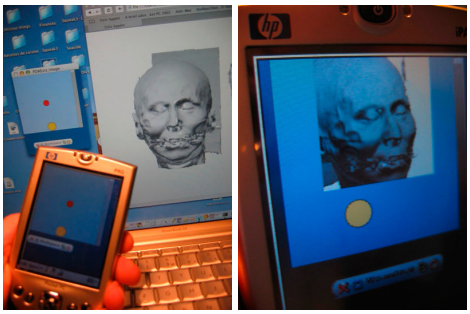

Figure 5: The PDA and Web clients, with the PDA acting as a remote control on the left, and as a visualization and control client on the right

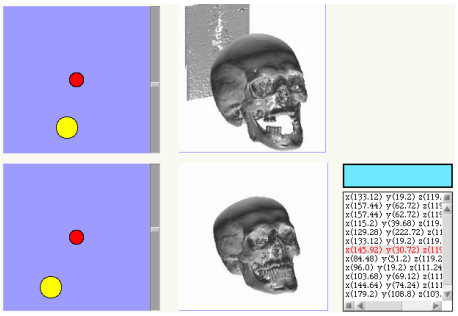

Figure 6: The Squeak User Interface with two different datasets, linked by a mediator agent

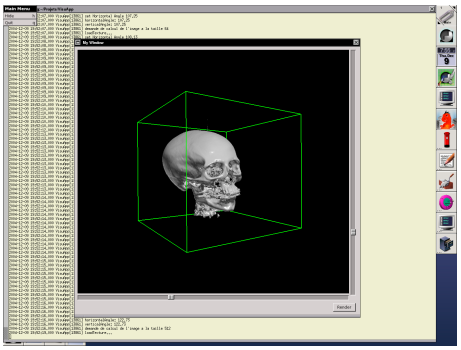

Figure 7: GNUstep visualization client on Linux

to use them. At the moment we mostly use a Squeak UI for rapid prototyping (Figure 6 shows the mediator agent along with the viewpoint agent described in Section 7) which make it easy to add new elements to the user interface, but we plan to extend our current web client to automatically generate the user interface for a pipeline (currently the web interface uses two java applets, one for visualizing the results of a pipeline, and one for controling the point of view), as html combined with java applets would gives us a very customizable and extensible user interface.

5 APPLICATIONS OF THE SYSTEM

The previous sections detailed the general system architecture. We will now introduce some applications of the reflective nature of the system.

The graphic pipeline is reificated through different agents; agents can use this architecture to gather information,

or even modify the architecture. The following sections show some examples of this approach:

- Section 6 “Visualization Strategies” details the Visualization Strategy pattern and its applications.
- Section 7 “Modifying the pipeline” introduce examples customizing the pipeline with additional agents adding new functionalities.

6 VISUALIZATION STRATEGIES

6.1 Presentation

If we look at the architecture for a standard rendering loop, Figure 4 can be simplified as shown in Figure 8.

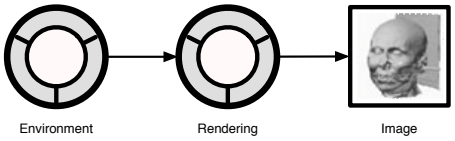

Figure 8: Simplification of the architecture

In this architecture, we consider a Rendering process as a Rendering agent using an Environment to generate an Image.

6.1.1 Visualization Strategies

A Visualization Strategy is a specific visualization process, like splitting the viewing output in different images, or distributing a rendering. We encapsulate such visualization patterns into a single agent. This agent answers the same rendering protocol as a “real” rendering agent, and can thus be used without any modification in place of a rendering agent – the other parts of the pipeline (*e.g.* the visualization client or the controls in Figure 4) are left untouched.

6.1.2 Genericity and Composition of Strategies

In general, Strategies do not provide a rendering algorithm themselves, but use existing rendering agents implementing the algorithms.

As long as an agent answers the rendering protocol it can be used transparently in a Strategy; which means that, for a given Strategy, any of the available rendering algorithms can be used. Conversely, we can say that Strategies are generic behaviors: a new rendering algorithm will be able to transparently take advantage of the existing strategies in the system.

As Strategies respond to the rendering protocol, they are considered by the system as Rendering Agents, and can thus be composited: in a given Strategy, instead of a “real” Rendering Agent, another Strategy can be used.

One needs to take care of the composition cost when building complex composited pipelines as performance can be impacted by the communication overhead.

The following sections demonstrate some of the strategies we created.

6.2 Progressive Rendering Strategy

Progressive rendering is a mechanism that computes a rendering in a low resolution, then gradually increments the resolution to improve the quality. Figure 9 shows an example of a 3-step progressive rendering for a volume dataset.

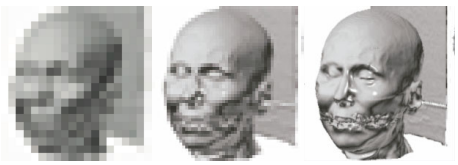

Figure 9: Progressive rendering

Although progressive rendering is a longer process to get the final image than calculating the final image directly (unless it is distributed by the pipeline), it’s a very useful mechanism, as it allows the user to have a quick feedback of what will be the final result. Moreover, the low quality rendering can be fast enough to achieve interactive frame rates.

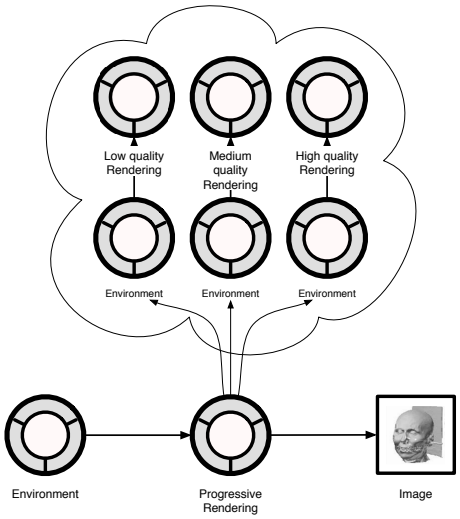

Figure 10: Using a strategy: progressive rendering

To create a progressive rendering strategy, we use an agent answering the rendering protocol, which will get the environment and where visualization clients can connect to.

This agent implements a three-step progressive rendering, and so requests three rendering agents for its needs. When receiving a rendering request, it modifies the resolution in the rendering environment, and then passes the new environment to the rendering agents. The progressive rendering agent is registered as a *visualization client* to each of the rendering agent. When it gets a result, it forwards it to the “real” visualization client.

Figure 10 shows the architecture of this progressive rendering agent – as can be seen on this diagram, the only modification lies “in” the rendering agent, and consists of only modifying the environment. All the other parts

of the pipeline are left untouched, an important characteristic of the Visualization Strategy pattern.

6.3 Distributed Rendering Strategies

The general approach to Distributed Rendering consists of splitting the computation load of a rendering on multiple machines, creating several partial results, then merge them to get the final result.

We implemented two approaches to distribute a rendering process, by using image-space methods or object-space methods to split the computation load, then merge the rendered isolated fragments in a single image as a final step.

The general idea is that once you have a process that can adequately split the rendering over multiple agents and then merge the result, the agents can easily be scattered on different machines, and their distribution controlled by another agent depending on specific parameters (*e.g.* taking in account the load of the machines).

6.3.1 Image-Space Splitting

We consider here the rendered image as our work space. We want to split it in multiple parts, where each part of the final image is rendered by a separate agent.

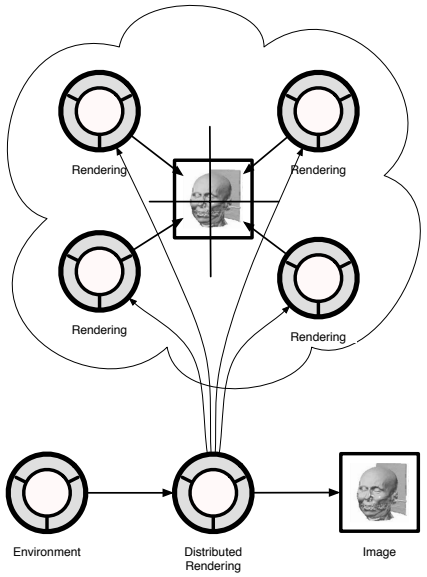

Figure 11: Image-space distributed rendering

To do that, we simply need to compute four different camera viewpoints that match the desired parts of the image. We do that in a strategy agent that gets the original rendering environment and extracts the camera viewpoint from it. It then sends corresponding computed viewpoints and "look-at" points to the four different rendering agents it requested (Figure 11 shows the architecture).

6.3.2 Compositing of Image-Space Parts

Merging the resulting partial images is very simple in that case, as we only need to build the final image by aggregating the different parts at their correct position.

6.3.3 Object-Space Splitting

With Object-Space Splitting, the idea is to render only a part of the dataset instead of the complete dataset in each agent. The dataset can be either physically split into multiple parts and each part sent to rendering agents, or alternatively have a rendering algorithm that accepts to render a part of a dataset (the right approach depends on the size of the dataset; we only implemented the selective part rendering for now).

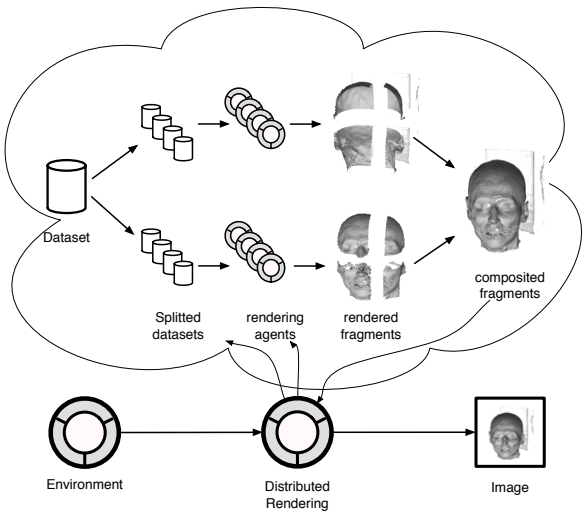

Figure 12: Object-space distributed rendering

Figure 12 shows the general architecture. A rendering strategy agent is used in a pipeline. This agent then computes the different data segments and sends them to the rendering agents. Each rendering agent will render a partial dataset into an image. Renderers need to generate the alpha channel for the image.

6.3.4 Compositing of Object-Space Parts

We obtain the final image by merging the partial images in the rendering strategy agent, using a simple back-to-front rendering algorithm.

7 MODIFYING THE PIPELINE

One of the differentiating aspects of our system is its reflective nature – agents can inspect the system and modify it. The following sections detail some examples of how agents can be added to a graphic pipeline to add functionalities.

7.1 Framerate Steering

As a first example, we would like to have a pipeline where the resolution of the image is tied to the framerate; that is, automatically set the resolution to match the desired framerate as close as possible.

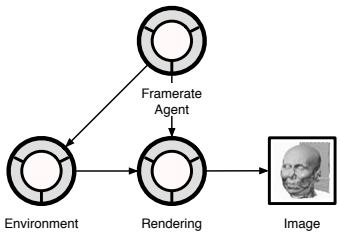

Figure 13: Framerate steering agent

We can do that by adding a single agent to a standard pipeline (Figure 13). The agent registers as an observer to the rendering agent, to monitor the rendering times. This framerate agent can then create a feedback loop by modifying the image size in the pipeline environment until the minimum framerate is reached.

This simple behavior could be extended by choosing among different rendering agents instead of fixing the image size.

7.2 Saving Viewpoints

In a graphic pipeline, the immediate agents' environment is the pipeline itself and the associated agents (see Figure 4).

One application we developed is a method to save the current camera position (viewpoint) and retrieve a list of the saved positions, so the user can build its own customized list of usual positions for a particular dataset.

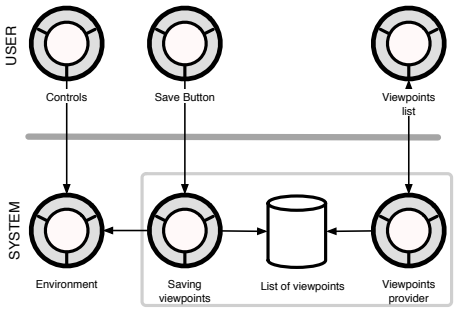

Figure 14: Saving and using viewpoints

Figure 14 shows the architecture enabling this functionality. We add three agents (framed on the figure) on the system side, one holding the viewpoints list (a data agent), one that can add a viewpoint to the list by checking the current viewpoint in the pipeline environment, and the third one that can returns the list of viewpoints. On the user side, we need two clients, one to request the addition of the current viewpoint, one to request the list of viewpoints.

This system can be used in any pipeline and, albeit it's a simple example, highlights how agents can add new knowledge to a pipeline and how they can use it.

7.3 A Mediator Agent

Another example of how agents can interact with a pipeline and extend its functionalities is a mediator agent, which can transform automatically some coordinates (*e.g.* camera position) used to visualize one volumetric dataset into equivalent coordinates for another volumetric dataset.

We can then manipulate one pipeline and have the movements replicated on a second pipeline, in real time, after transformation.

The current prototype can be used as the start of a collaboration mechanism, and we also want to use it in the future coupled with the "saving viewpoints" agents to automatically use viewpoints saved for one dataset with another, which can be useful for medical applications.

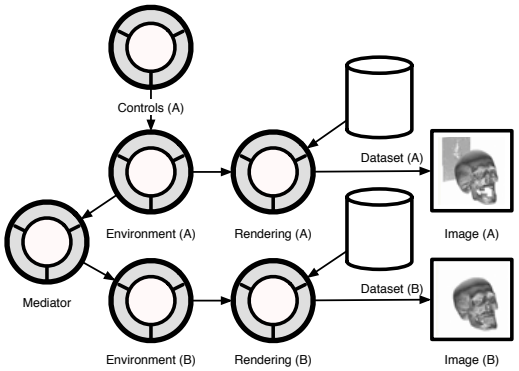

Figure 15: A mediator agent

Figure 15 shows the actual architecture of the system. We use here two pipelines running in parallel, with a mediator agent coupling the pipelines' environments; the mediator agent is registered as a listener to the environment agent of the *Environment A* and knows the deltas between both environments (after a calibration step). Using the controls of *Pipeline A* will update *Environment A*; the Mediator is then notified of the change, and will update *Environment B* (see Figure 6 for a screenshot of the current system).

8 PERFORMANCES

For the 256^3 data sets rendered in these examples, all these agents work in real-time, and achieve high frame rates when distributed (> 25 fps). We aim in the future to use UDP instead of the current TCP based transmission mechanism in order to augment the performances and improve the latency of the system, particularly when dealing with bigger datasets.

9 CONCLUSION

The use of web services and XML/RPC gave us the opportunity of mixing different languages and programs in a common system, allowing us to take advantage of each language's/platform's strong points.

Focusing on a simple, dynamic, agents architecture and building our system on top of it proved to be very useful during the conception of the architecture. This dynamism also allowed us to create new architectures and discover interesting patterns.

The current system is already a useful research tool to test new ideas and architectures, although there is different aspects that we would like to improve:

- use rendering agents that utilise GPU instructions, thus enabling a fully featured GPU cluster using the existing distributing and performance agents.
- extend the reflective system to have an "intelligent user interface" that agents can modify, by partly specifying it in a pipeline.
- implement planning agents and in general more autonomous algorithms – add intelligent behavior to the distributing mechanism
- implement ontologies on top of the current system would be interesting to provide another information level agents could leverage
- work on collaboration scenarios, taking advantage of the current PDA client and Web client
- extend the current Volume Rendering pipeline to a more general graphic rendering pipeline (polygons)

Acknowledgements

Financial support for this work was provided by the UK Engineering and Physical Sciences Research Council through grant numbers GR/S46567/01, GR/S46574/01 and GR/S46581/01.

REFERENCES

[ADK+99] Martin Aeschlimann, Peter Dinda, Loukas Kallivokas, Julio López, Bruce Lowekamp, and David O'Hallaron. Preliminary report on the design of a framework for distributed visualization. *Parallel and Distributed Processing Techniques and Applications*, pages 1833–1839, 1999.

[App] Apple. Cocoa. `http://developer.apple.com`.

[ASA01] Mark Astley, Daniel C. Sturman, and Gul A. Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5):99–107, 2001.

[ASF05] Apache Software Foundation ASF. Apache xml-rpc. `http://ws.apache.org/xmlrpc/`, 2001-2005.

[BBC+05] Ken Brodlie, John Brooke, Min Chen, David Chisnall, Ade Fewings, Chris Hughes, Nigel W. John, Mark W. Jones, Mark Riding, and Nicolas Roard. Visual supercomputing: Technologies, application and challenges. *Computer Graphics Forum*, 24(2):217–245, 2005.

[BDG+04] Ken Brodlie, David Duce, Julian Gallop, Musbah Sagar, Jeremy Walton, and Jason Wood. Visualization in grid computing environments. *IEEE Visualization*, pages 155–162, 2004.

[Cam95] Gordon Cameron. Modular visualization environments: Past, present, and future. *Computer Graphics*, 29(2):3–4, 1995.

[Dye90] Scott D. Dyer. Visualization: A dataflow toolkit for visualization. *IEEE Computerr Graphics and Applications*, 10(4):60–69, 1990.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[FK99] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2, "Computational Grids". Morgan-Kaufman, 1999.

[FKT01] Ian Foster, Carl Kesselman, and Steve Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

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

[FSF] FSF. GNUSTEP, a FSF implementation of the OPENSTEP APIs. `http://www.gnustep.org`.

[GAW05] Ian J. Grimstead, Nick J. Avis, and David W. Walker. Visualization across the pond: How a wireless pda can collaborate with million-polygon datasets via 9,000km of cable. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, pages 47–56, 2005.

[GFB05] Zahia Guessoum, Nora Faci, and Jean-Pierre Briot. Adaptive replication of large-scale multi-agent systems – towards a fault-tolerant multi-agent platform. In *Proceedings of the fourth international workshop on Software engineering for large-scalemulti-agent systems*, pages 1–6, 2005.

[HEvLRS03] Hans Hagen, Achim Ebert, Hendrik van Lengen Rolf, and Gerik Scheuermann. Scientific visualization – methods and applications –. In *Proceedings of the 19th spring conference on Computer Graphics*, pages 23–33, 2003.

[IBM] IBM. Autonomic deployment model. `http://www-306.ibm.com/autonomic/levels.shtml`.

[JJZ+04] Hu Jun, Gao Ji, Huang Zhongchao, Liao Beishui, Li Changyun, and Chen Jiujun. A new rational model of agent for autonomic computing. In *Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics*, volume 6, pages 5531–5536, 2004.

[KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, pages 41–50, 2003.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[KHRV04] Dieter Kranzlmüller, Paul Heinzlreiter, Herbert Rosmanith, and Jens Volkert. Grid-enabled visualization with gvk. *Across Grids 2003*, LNCS 2970:139–146, 2004.

[MF00] Isabel Harb Manssour and Carla Maria Dal Sasso Freitas. Collaborative visualization in medicine. In *Proceedings of The International Conference in Central Europe on Computer Graphics, Visualization And Interactive Digital Media (WSCG)*, pages 266–273, 2000.

[MGR+99] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed agents for networking things. In *Proceedings of the First International Symposium on Agent Systems and Applications / Third International Symposium on Mobile Agents*, page 118, 1999.

[Mül03] Marcus Müller. XML/RPC framework for objective-c. `http://www.mulle-kybernetik.com/software/XMLRPC/`, 2002-2003.

[RKACM03] R. Rangel-Kuoppa, C. Aviles-Cruz, and D. Mould. Distributed 3d rendering system in a multi-agent platform. *Computer Science, 2003. Proceedings of the Fourth Mexican International Conference on*, pages 168–175, September 2003.

[RWB+05] Mark Riding, Jason D. Wood, Ken W. Brodlie, John M. Brooke, Min Chen, David Chisnall, Chris Hughes, Nigel W. John, Mark W. Jones, and Nicolas Roard. e-viz: Towards an integrated framework for high performance visualization. In *UK e-Science All Hands Meeting 2005*, pages 1026–1032, 2005.

[Squ] Squeak. `http://www.squeak.org`.

[SWB03] John Shalf and E. Wes Bethel. The grid and future visualization system architectures. *IEEE Computer Graphics and Applications*, 23(2):6–9, 2003.

[UFJK+89] Craig Upson, Thomas Faulhaber Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[Win99] Dave Winer. XML/RPC specification. `http://www.xmlrpc.com/spec`, June 1999.

[ZM05] Avelino Francisco Zorzo and Felipe Rech Meneguzzi. An agent model for fault-tolerant systems. In *Proceedings of the 2005 ACM symposium on Applied Computing*, pages 60–65, 2005.