

The PGPGrid Project

Ewan Borland, **Paul Cockshott**, Xiangyang Ju, Lewis Mackenzie, Viktor Yarmolenko
Department of Computing Science, University of Glasgow

Paul Graham

EPCC, University of Edinburgh

Abstract

The PGPGrid project aims to parallelise the process of extracting range data from an experimental 3D scanner using the Grid as a vehicle for accessing necessary resources. The application is potentially highly parallel but has some unusual features such as rapid spawning of processes in real time and a dynamic inter-process network topology. These characteristics are such as to require enhancement of the usual task migration capabilities of the Globus toolkit. The present paper initially discusses attempts to estimate the real parallelisability of the scanner application. It then describes a new Java API, based on Milner's π -calculus, which could be used to extend Globus in a manner capable of supporting systems with this kind of dynamic parallel structure.

1. Introduction

PGPGrid, a joint venture with *Peppers Ghost Productions* and the *Edinburgh Parallel Computing Centre* aims to parallelise extraction of range data from Glasgow University's 3D TV scanner [1], which uses 24 video cameras to image a subject from many directions at once. The data is then used to build a dynamic 3D model, spatial resolution 4mm, temporal resolution 0.04 seconds. Software allows an animator's model to be conformed to data captured from an actor, transferring the movement of a real human subject to equivalent virtual movement of the model. The conformation software was developed for "still" models but current work extends it to moving cartoon-like characters.

The cameras are organised into *Pods*, each comprising two 8-bit monochrome cameras and a single 16-bit colour camera. The monochrome cameras are synchronised by a 25Hz master trigger signal. A separate trigger, with a phase lag, synchronises the colour cameras. Using continuous light, the actor is illuminated with a speckle pattern which provides features for the ranging algorithm to home in on. Without speckle there is insufficient contrast for effective ranging. Strobe lamps synchronised with the colour cameras overflash suppressing the speckle during the exposure of the colour cameras.

Each pod is controlled by a single computer which captures the three video sequences to internal RAM buffers. Next each synchronous

pair of monochrome images is matched to produce a *range map*. Each monochrome image is around 300Kbytes. The colour image is used afterwards, when images are adjusted to blend the brightness of adjacent pixels from distinct views. Each pod generates 1.2Mbytes of data per frame, a total data generation rate of over 240Mbytes/second.

The matcher algorithm itself operates via a two stage process as follows

- a) An (x, y, c) , where c is correlation, *disparity map* is created for each pod, consisting of three floating point planes, a total of about 3.6Mbytes per pod.
- b) From each disparity map a range map is computed. For each pixel this contains a 32-bit float giving the distance in meters from the camera. The map is about 1.2Mbytes in size.

The range maps from 8 pods give point cloud data to a model-builder to create a triangulated VRML model. Finally successive VRML models have to be combined into a single space/time model, (Figure 1). For each frame period we initiate a process group comprising 8 matcher tasks and 1 model-building task. The algorithms are all implemented in Java. Ideally, a process group of this type should be spawned at 25Hz; however, the operations are very CPU-intensive and the processing time exceeds the capture time on any feasible system. The capture PCs have two 2GHz Athlons, a single processor of this class, with adequate memory resources, can process one pod range-map in 50 seconds, the VRML

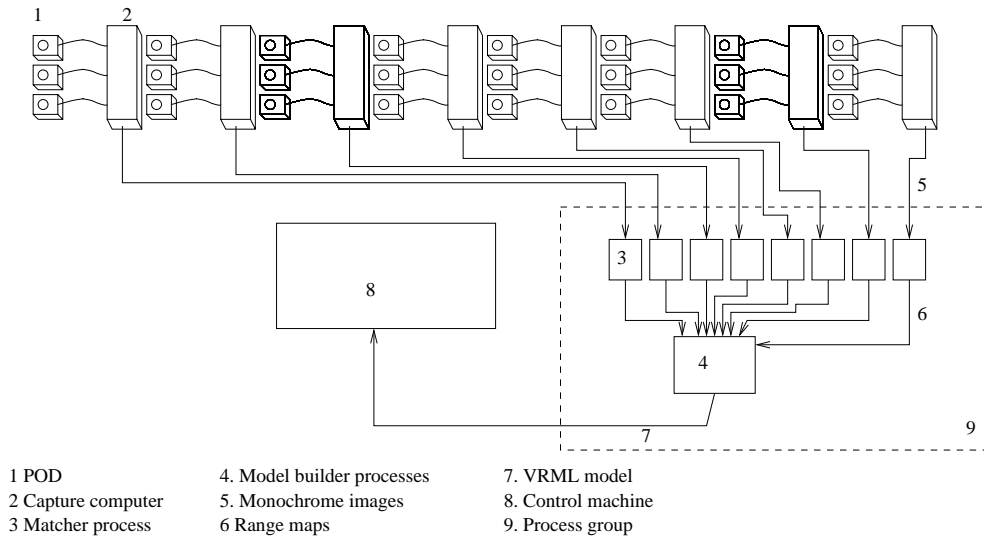


Figure 1. Data-flows and processes for 3D scanner

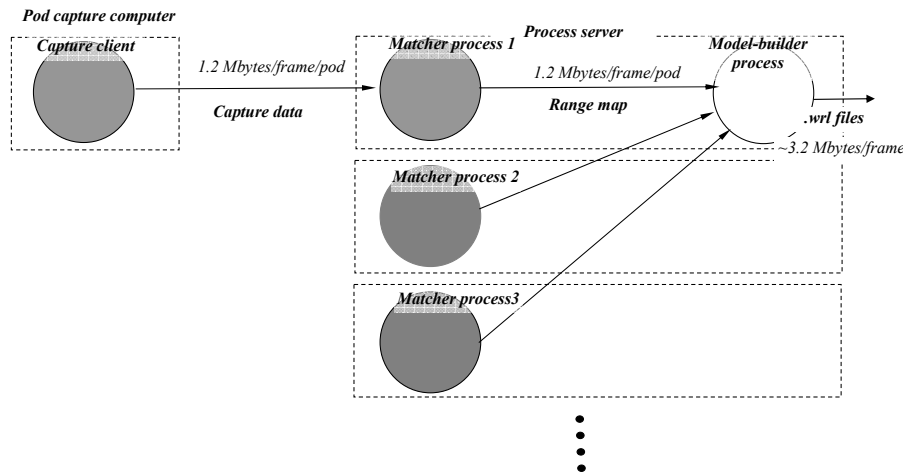


Figure 2. Basic Parallelisation data flows for process server

model build from 8 such maps taking 100 seconds.

The processes are inherently easy to parallelise, the main challenges being the sheer volume of data being generated and moved. One of our goals is to use Grid technology to acquire the resources needed. The remainder of this paper will discuss our initial attempts to achieve this goal.

2. Preliminary Experiments on Parallelisation

The parallelisation experiments conducted so far have used the process architecture in Figure 2 with statically located matcher processes and a simple server acquisition system. Four pods and their associated control computers are used, imaging a subject above the shoulders only. Instead of capturing data directly from the

cameras, a pre-recorded 300 frame sequence is streamed from local hard disk. Each capture client, runs a number of job management threads and, as soon as a new pair of monochrome images is read, control of processing is passed to one of these. The newly active thread immediately attempts to acquire an available matcher server process.

Matcher processes are always running on each available server host and each client holds a static list of the IP numbers and ports for these. A matcher will only accept a connection from a client when not already busy. Once a connection is accepted, input data (~600Kbytes) is streamed from the client. The colour image is not transmitted. When the matching process has completed, the disparity map (~3.6 Mbytes) is returned to the client, and the matcher waits for a new connection. This arrangement ensures that all the server hosts are equally utilised and each runs no more than one server task at a time.

Exp	Total CPUs	Capture CPUs	Remote CPUs	Time (min)	Comments
0	4	4	0	215	One Angel CPU/pod
1	12	12	0	75	12 Angel CPUs
2	16	0	16	48	16 NeSC blue dwarf CPUs
3	28	12	16	29	16 blue dwarf, 12 Angel CPUs
4	43	12	31	21	16 blue dwarf, 12 Angel, 15 other servers

Table 1. Results of parallelisation tests

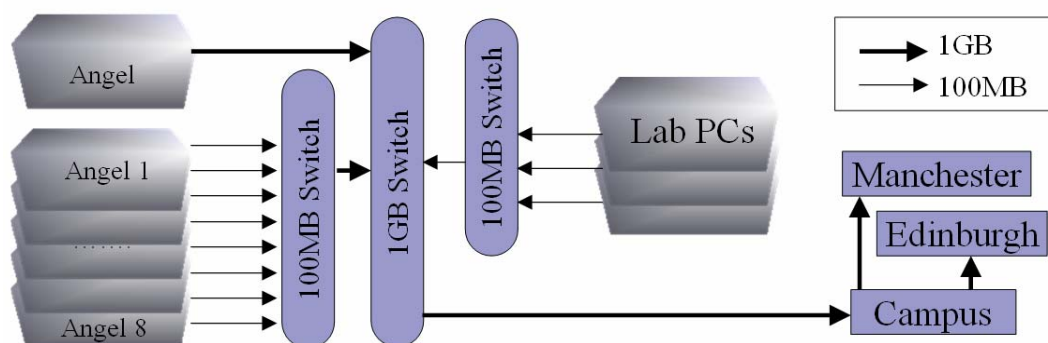


Figure 3. Topology of the Network

All work allocation is left to the client to minimise the load on the servers. The capture machines are attached to a 100Mbps Ethernet LAN linked to SuperJanet via a 1Gbps pipe (see Figure 3).

The server hosts available for running matcher processes fall into categories: the 8 capture machines (even those running capture clients have spare capacity); various hosts attached directly to the local switched Ethernet; and a remote IBM 16 CPU host, *Blue Dwarf*, sited at NeSC in Edinburgh.

Prior to running parallelisation experiments, the Angel and Blue Dwarf processors were benchmarked. The respective times taken to perform a single disparity map computation from a pair of monochrome images were about 43 and 35 seconds (first entry in Table 1, Experiment 0).

The network is a critical factor and ultimately limits the amount of parallelism possible. Each Angel is connected to a 100Mbps full duplex switch and needs to receive 3.6Mbytes returned for each remote task initiated. Assuming the machine can service its network interface at maximum speed and that there is no other network traffic, the best possible time in which an Angel can dispatch a matcher task is about 0.3 seconds. If each matcher task takes n seconds, the total number

of servers that an Angel can keep busy is about $3.5n$. This would suggest that for 2GHz class processors running our Java algorithms, parallelisation of about 150 should be achievable.

At present there are insufficient server hosts to test this limit (a total of 43 are currently assignable). Nonetheless, it is interesting to see if the prediction holds at least out to values of N which can be tested. The results of the parallelisation tests are listed in Table 1.

As can be seen, with the CPU resources available, the frame processing task is parallelisable and a speedup of an order of magnitude is demonstrated for a similar increase in the number of processors. The conditions of the experiment are pessimistic. Were matcher servers to return range maps instead of disparity maps data would be reduced to 1.2Mbytes per task, tripling the parallelisation limit. This can be improved further by also building the models remotely. With gigabit interfaces parallelisation of the order of 10^3 seems possible. Such levels of processing power and backbone bandwidth are not yet realistic, but the argument suggests that real time processing of 3D images will be possible.

3. The JPie Interface

The dynamic task creation and remote pipe migration required by PGPGGrid motivates the development of a Java interface loosely modelled on the primitives of the π calculus [2, 3]. The interface, *JPie*, is intended for use as a substratum for this and other GRID-based parallel computing applications. It allows the dynamic creation of both remote processes and communications channels between them. It also allows dynamic reconfiguration of the network of channels. *JPie* tries to achieve this with the minimum number of primitives, integrating these into the existing Java class framework. It is at roughly the same abstraction level as JPVM [4], IceT [5] and JCSP [6].

The overall *JPie* class hierarchy is as follows.

1. abstract class *JPieTask* represents the unit of work to be done as a parallel process. It implements the `java.lang.Runnable` interface, allowing the construction of threads from *JPieTasks*. It has additional methods that allow channels to be associated with *JPieTasks*. Before the *JPieTask* run method is called, the task's transmissible input and output channels, known respectively as *funnels* and *taps*, must be initialised. A running task can obtain its taps and funnels via `get` methods supplied.
2. interface *JPie* is a factory interface whose job is to create processes, taps, funnels and pipes (a pipe is a linked tap and funnel). It has a core method `spawn` which causes a *JPieTask* to be run locally or remotely depending on resources. Time and memory parameters which specify anticipated resource usage are supplied. `spawn` will fail if adequate resources cannot be found.

JPie also includes methods to create *taps* and *funnels* that can convert streams in the local environment to *JPie* streams that can be sent to remote machines. The `createPipe` method also permits interprocess communication streams to be set up between two daughter tasks.

The set of classes required by a remote task are made available to it via a custom class loader which retrieves jar files containing class definitions from the originating process. The set of jar files available to the class loader is specified via a

call to the `setJars` method of the originating *JPie* instance.

There is also a need to be able to identify sources of data that are at known locations on the Web. *JPie* will provide for this by adding methods that allow for *taps* and *funnels* to be published and associated with URIs.

3. abstract class *JPieTap* implements `java.io.InputStream`, and requires the implementation of a method to get a byte (`read`). The standard Java blocking protocols are followed.
4. abstract class *JPieFunnel* implements `java.io.OutputStream` and must provide a write method that writes a single byte to the output channel. It can block on write if the remote process has not performed sufficient reads.
5. interface *JPiePipe* provides a bi-directional inter task communications channel. This has two methods: `getTap()` and `getFunnel()`. To set up a data flow graph one creates pipes and passes the input and output channels to tasks which are then forked. It is implemented by class *UniprocessorJPiePipe* using built-in Java classes *PipedInputStream* and *PipedOutputStream*; and used by class *GridJPiePipe* using an appropriate remote communications protocol. A key issue here is serialisation of *pipes*, *taps* and *funnels*.
6. The overall aim of *JPie* is for processes to be able to run across a network of machines without knowing where these machines are or having to reference the machines explicitly in any algorithm. An algorithm expressed in Java using *JPie* should be invariant under alterations in the physical collection of machines that run it.

In principle one of the great strengths of Java is that it allows a single binary image to run on a variety of different physical computers, each of which may be running a different operating system and using a distinct instruction set. In order to ensure that a task will produce the same result regardless of its host, every machine that is to run *JPie* tasks must have the JRE installed and have access to the *JPie* library. Data can be transferred to or from a task over its *taps* and *funnels*.

Consider the problem of creating on local machine, *A*, a task, *T*, that will execute remotely, filter a source file stored on *A* and write its result to a destination file on *A*. When *T* is spawned the initiating task on *A* first locates a machine *B* able to run it. On *B* a daemon is executing which accepts task execution requests. Once the daemon agrees to run the job it execs a new java VM and provides, on the command line, details which are used to connect *A* and *B* with a pipe *P*.

The machine *A* then serialises *T*, through *P*. The new VM on *B* deserializes *T* using a custom class loader, which loads required class definitions from *A*, and runs *T*. *T* contains Taps and Funnels, which must themselves be serialised. The serialization of `JPieFunnel` and `JPieTap` classes is achieved by “linking” the deserialized object to the original stream. The method used to perform this “linking” is irrelevant from the perspective of the JPie interface and will depend on the particulars of specific implementations.

Note that a `JPiePipe` is comprised of a connected tap-funnel pair and thus is also easily serializable because the receiving machine will simply re-link both ends of the pipe back to their original locations.

4.Resource Locator

JPie can make use of various resources from a pool, but requires a mechanism to identify, check for availability and access these resources. This is where the idea of a resource locator comes in. The model we have proposed has three components: the Initiator, the Locator and the ServEnt (Server Entity), jointly referred to as the ILS model. EPCC is undertaking the major share of the development of this software with requirements and other inputs provided by 3D-Matic Lab.

A typical interaction of the components in the ILS model would be as follows: A ServEnt advertises its hardware characteristics and availability for JPie tasks by informing the Locator, where the Locator maintains a list of such information. An Initiator wishing to launch a JPie task requests suitable resources from the Locator. The Locator returns a list of resources, from which the Initiator then confirms the first resource availability directly with the ServEnt, and then if available submits the JPie task. On task completion the ServEnt passes the job metrics back to the Initiator. Currently EPCC is focusing on the Locator-ServEnt and Locator-Initiator interactions and functionality.

- **Locator** - The Locator maintains an updateable list of resources with their associated static and dynamic information. The Locator stores this information in a database such that it can be maintained, accessed and searched quickly and efficiently. The Initiator and ServEnt can request and update this information via two Webservices[7] made available by the Locator. The Locator is hosted within a Tomcat[8] server, and utilises the Axis[9] framework for constructing SOAP[10] messages and Webservices.
- **ServEnt** -The ServEnt acts as the access level for the ILS model to utilise the underlying resource. When the ServEnt is installed on a machine, the user must supply the architectural information (based on XRSL[1111]) that is in turn passed on to the Locator. The ServEnt also can accept JPie tasks from an Initiator, and will pass metrics of the task run to the Initiator when it has completed.
- **Initiator** - The Initiator instigates JPie tasks. It requests resources from the Locator and receives the addresses of the resources available and appropriate to the job. The Initiator negotiates with the resources in turn via their ServEnts to discover if they can accept the job. If declined by one resource, then the next ServEnt is negotiated with. Once the negotiations have completed, the Initiator submits the task to the ServEnt.

4.1 Methodology

There are several technologies already in existence that offer similar functionality to the interactions described above, and a survey of these was undertaken as part of the PGPGrid project. However, none of the technologies matched the requirements exactly, and those that came closest were relatively heavyweight solutions, which would have required significant investment in terms of time and effort in adapting them to our requirements. Due to the inherent difficulties of such an approach, and given the time constraints, it was decided to utilise the Webservices paradigm. This allowed an iterative approach to developing the functionality, and the resulting implementation should also be straightforwardly extensible to the emerging WSRF[12] standards.

5. Model Conformation

One of the project goals is to demonstrate the use of dynamic 3D data to drive CG animation. In addition to providing a more productive workflow we hope that animation generated in this way will prove to be much more believable than animation generated by traditional means. In particular we emphasise the subtle and complex facial movements that we can capture which are often lacking in artificially generated animations.

The model building process produces a set of discreet triangulated meshes, one per frame. We now describe the workflow developed to move from these discreet models to a single animated model suitable for inclusion in a professional studio production.

5.1 Model Mark-up

The most important step is moving from a set of discreet models to a single unified model which an animator can interact with consistently across frames. This is achieved by mapping a generic model to the captured data using a process called conformation [13]. Before we conform the generic model we must first identify corresponding feature points between it and the captured data.

This relies on tracking the position of features we call landmarks. These landmarks are points in the captured data corresponding to features that must be mapped accurately in order to preserve the quality of the animation.

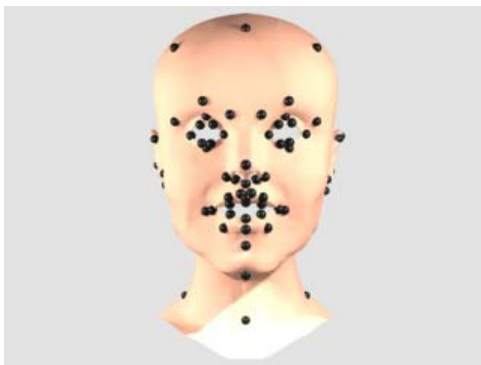


Figure 4. Landmarks in place on a generic mesh

After experimenting with several different sets of landmark points we have opted to use a subset of the points defined in the MPEG-4 standard for facial animation [14]. However, it was necessary to introduce additional landmarks to stabilise the structure of the mesh between frames (see Figure 4).

The task of recording the position of these landmarks is complicated by the volume of data. Every frame consists of one VRML model and four texture images, each of which is 24-bit colour at a resolution of 640x480 pixels. Taking into account the data structures necessary to display and manipulate this data interactively, every frame can use up to 6MB of memory (dependant on mesh complexity). A simple scaling calculation shows that an 8 second sequence requires at least 1GB of memory.

A mark-up interface has been developed which allows the user to operate on longer sequences in one session. This interface enables the user to select any VRML model as the “base” position for the landmarks. Once the mesh containing the landmarks has loaded the user can select and drag landmarks across the surface of the model. When all landmarks are in place the user moves to the next frame. Duplicated effort is minimized by projecting landmarks from the previous frame onto the model in the new frame, thus only minor corrections are needed between frames.

To reduce the demands of this process on the computer being used, partial sequences can be marked-up. The last frame from one partial sequence is loaded as the start point for the next sequence, enabling the user to continue with minimum interruption. In this way it takes 3mins to position landmarks for a single frame. Due to memory limitations, working with 1 second at a time results in the best trade-off between user effort and software performance. Although it would seem that the demands of this process could easily be reduced by only loading models into memory as they were needed, preliminary tests show that the loading time for these models is prohibitively expensive in terms of perceived interactivity.

5.2 Conformation

Once the landmark points have been recorded then the generic model can be conformed to the captured data. The conformation algorithm, which deforms a generic model to scanned data, comprises a two-step process: global mapping and local deformation.

- **Global Mapping** Global registration and deformation are achieved by means of a 3D mapping based on corresponding points on the generic model and the scanned data. The 3D mapping transforms the landmark points on the generic model to the exact locations of their counterparts on the scanned data. All other points on the generic model are interpolated by the 3D mapping; this mapping is established using corresponding

feature points through radial basis functions [15]. The global mapping results in the mesh of the generic model being subject to a rigid body transformation and then a global deformation to become approximately aligned with the scanned data.

- **Local Deformation** Following global mapping, the generic model is further deformed locally, based on the closest points between the surfaces of the generic model and the scanned data. The polygons of the generic model are displaced towards their closest positions on the surface of the scanned data.

An elastic model is introduced in the second step of the conformation. The global deformed mesh is regarded as a set of point masses connected by springs. During the second stage of the conformation process, the mesh anchored to landmarks is relaxed to minimize its internal elastic energy. The displacements of the vertices deformed to their closest scanned surface are constrained to minimize the elastic energy of the mass-spring system. The governing equation for the relaxation of a single vertex \mathbf{x}_v is

$$m_v \ddot{\mathbf{x}}_v + d_v \dot{\mathbf{x}}_v + \sum_{j=1}^l \mathbf{f}_{vj} = \mathbf{f}_v^{\text{ext}}$$

where m_v is the mass of the v th vertex, d_v its damp factor, \mathbf{f}_{vj} is the internal elastic force of the spring connecting the j th vertex to the v th vertex, l is the number of vertices connected to the v th vertex and $\mathbf{f}_v^{\text{ext}}$ is the sum of the external forces on the v th vertex.

The relaxation uses the following procedure:

1. Each vertex \mathbf{x}_v is deformed to its closest position on the measured shape. Conformation ends here if no elastic constraint is applied; else, continue.
2. The sum of the internal forces $\sum \mathbf{f}_{vj}$ on each vertex \mathbf{x}_v is calculated. There is no external force in this model.
3. The acceleration, velocity and position of each vertex \mathbf{x}_v are updated with the exception of the vertices of the triangles of the mesh in which the landmarks lie.

Steps 1 to 3 are repeated until the mesh is settled or the number of iterations exceeds a fixed number.

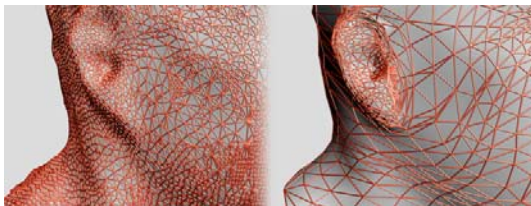


Figure 5. Unconformed versus conformed mesh structure

We have found that better results at this stage can be obtained by conforming the generic model to the first frame of the captured sequence and by then conforming the resulting mesh to each subsequent frame. This approach means that the deformation required is relatively small and is hence less prone to introduce errors or noise.

5.3 3D Studio Max Animation

After conformation we have a set of discrete models with the shape and motion from the scanned data but having the mesh structure of the generic model. In this form the sequence is not directly usable by an animator; however the transformation to a single model with a set of key frames defining the motion is straightforward.

Even after conformation there is still some noise present in the data that causes it to stand out when incorporated into “clean” CG environments. To combat this we apply a high-frequency filter to the motion of individual vertices. See Figure 6. This filtering is achieved by convolving the trajectory of each vertex with a symmetric one dimensional Gaussian kernel of the form:

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left(\frac{x^2}{2\sigma^2}\right)}$$

The accuracy of our models is improved because each vertex’s position at a given time is re-evaluated using a discrete temporal window. Applying this constraint to a vertex allows complex structure that may not be evident in a single frame to be recovered from the stream of data. In particular, highly detailed and relatively static areas, such as the ears, show significant improvements after filtering. Filtering also reduces the impact that spurious motions due to noise have on the fluidity and realism of motion as small deviations in trajectory are corrected (see Figure 6).

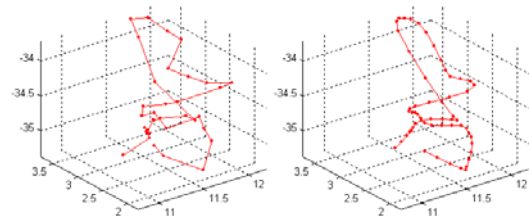


Figure 6. Trajectory of a vertex before (left) and after (right) filtering

To ensure that no important details are sacrificed, an interface has been incorporated

into 3D Studio Max allowing the degree of filtering to be specified as the animation is being imported, thus allowing the animator full control over the extent of the filtering. Early indications are that this workflow will produce very good results, maintaining a good balance between realism of motion and quality of the model (see Figure 7).



Figure 7. Image from test sequence rendered by Pepper's Ghost

Conclusions

The parallelisation experiments described above show that the 3D scanner application is extremely well-suited for distribution in a Grid-type environment. With sufficient resources, parallelisation of sufficient degree to support real-time processing of images can reasonably be envisaged. However, the dynamic nature of the process creation and dispatching requirements and the necessity of supporting a reconfigurable inter-process communications network requires an extension to the normal Globus task control capabilities. With this in mind the authors have described a Java interface modelled on the pi-calculus which satisfies these requirements. This has been successfully implemented in a preliminary form and work is proceeding to integrate it fully with the Grid protocols.

Acknowledgments

The authors wish to acknowledge the funding by NeSc, Edinburgh, UK, and thank our project partners Peppers Ghost Productions and The Edinburgh Parallel Computing Centre for their input.

Bibliography

1. "Experimental 3D TV Studio", Cockshott, W.P., Hoff, S., Nebel, J-C, IEE Proceedings - Vision Image and Signal Processing, 2003.
2. "The Polyadic π -calculus, a tutorial", Milner, R., 1991.
3. "A calculus of mobile processes", Milner, R., Parrow, J., Walker, D., Information and Computation, 100:1-77, 1992.
4. "JPVM: Network Parallel Computing in Java", Ferrari, A., ACM Workshop on Java for High Performance Network Computing, 1998.
5. "IceT Distributed Computing and Java", Gray, P., Sunderam, V., Concurrency, Practice and Experience, Vol. 9, No. 11, Nov. 1997.
6. "A CSP model for Java Multithreading", Martin, J.M.R. and Welch, P.H., 'Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)', pages 114-122, IEEE Computer Society Press, June 2000.
7. Webservices, <http://www.w3.org/2002/ws/>
8. Apache Tomcat, <http://jakarta.apache.org/tomcat/>
9. Axis, <http://ws.apache.org/axis/>
10. Simple Object Access Protocol, <http://www.w3.org/TR/soap/>
11. Extended Resource Specification Language, <http://progress.psnc.pl/English/architektura/xrsl.html>
12. WS-Resource Framework: FAQ, <http://www.globus.org/wsrf/faq.asp>
13. "Conformation from generic animatable models to 3D scanned data", Ju, X. and Siebert, P., Proc. 6th Numérisation 3D/Scanning 2001 Congress, Paris, France, 2001, pp 239-244.
14. "MPEG-4 Facial Animation", Pandzic, I.S. and Forchheimer, R., John Wiley & Sons Ltd. ISBN 0-470-84465-5.
15. "Individualising Human Animation Models", Ju, X. and Siebert, P., Proc. Eurographics 2001, Manchester, UK.