The University Of Sheffield.

Department of Computer Science

Grid Computing Techniques for Synthetic Procedural Planets

Memorandum CS–07–01

Manuel Noronha Gamito

January 2007

Contents

List of Figures

List of Tables

1 Introduction

As part of his PhD in Computer Science at the University of Sheffield, the author has been developing an image synthesis application with the purpose of rendering complex, synthetically generated, planetary landscapes. The idea of modelling and rendering an entire synthetic planet, with a radius roughly equal to the radius of the Earth, is reasonably new and has benefited from recent advances in procedural fractal models [Peachey, 2003]. Until recently it was only feasible to model limited landscapes that were sampled onto a two-dimensional grid of elevation values [Marshall et al., 1980]. With a procedural synthetic model for the terrain, it is now possible to have a purely functional description of the surface of an entire planet, with the ability to generate surface detail at any point on the surface of the planet and at any scale [Musgrave, 2003].

Despite the much greater flexibility offered by procedural models, there is a significant impact on the time that it takes to render an image of the terrain. Conversion of the terrain into a polygon mesh, that could be subsequently rendered in hardware with modern graphics boards, is not a possibility because it implies a loss of surface detail when discretising the terrain into triangles. If all the visible surface details are to be preserved, the resulting triangle mesh would have to be so fine as to be computationally intractable. Procedural terrains must be rendered with a direct ray tracing approach [Whitted, 1980]. For each pixel on the screen, a ray is traced into the scene and its intersection point with the terrain is found. The complexity of a ray tracing solution for rendering procedural landscapes comes from the fact that the terrain function is fractal and, therefore, has a very irregular shape. Finding the ray-terrain intersection point accurately requires significant computational resources.

Early landscape renderings performed by the author during his first year of PhD studies in 2005 hinted that a conventional ray tracing solution, running on a single computer, would soon become impractical as the research progressed and the complexity of the terrain functions kept increasing. The problem would be further worsened when rendering computer animations of camera flybys over the terrain, with hundreds of individual images having to be rendered to achieve a typical rate of 25 images per second of playback. After optimising the ray-terrain intersection algorithm to achieve maximum efficiency, the only possibility of further reducing the rendering time is to implement a distributed ray tracing solution. Fortunately, in this case, ray tracing is a type of algorithm that is easy to parallelise. The computation of the intersection point for the ray passing through a pixel is performed independently of the same computation for every other pixel. One just has to divide the image into smaller rectangular regions, that are called *tiles* in this report, and run a ray tracer independently for each tile. As a final step, one gathers all the individual tile renderings and assembles the complete image. Because each pixel is rendered independently, there is no need to use complex message passing mechanisms or to manage shared memory segments as part of the implementation for a distributed ray tracer.

A distributed ray tracer for terrain rendering was implemented by the author in 2005 on the Department of Computer Science Night Train. The Night Train is a Beowulf cluster of desktop computers, installed on the Department's student computer rooms. The Night Train only becomes active during the night and on weekends, when the rooms are closed, and is managed with the Sun Grid Engine. The same distributed ray tracer was later ported to the University of Sheffield High Performance Computing node on Iceberg. Both distributed versions of the ray tracer, however, only handle the rendering of individual images. A new version of the ray tracer was subsequently developed, handling not only the distribution of tiles of an image but also the distribution of images that are part of a computer animation. The new ray tracer was tested on the set of High Performance Computing nodes that are available through the White Rose Grid by the Universities of Sheffield, Leeds and York.

2 Photorealistic Rendering of Synthetic Landscapes

A description of the algorithms for the photorealistic rendering of synthetic landscapes is beyond the scope of this report. Here, only a mention is given to the three main problems that a photorealistic landscape renderer must solve. These problems are responsible for the computational complexity of the renderer and must be solved in the following order:

- Ray-terrain intersection tests.
- Light scattering in the atmosphere.
- Anti-aliasing and motion blur.

Ray-terrain intersection tests are performed with the *sphere tracing* algorithm [Hart, 1996]. The terrain is represented by an implicit surface $\{f(\mathbf{x}) = 0 : \mathbf{x} \in \mathbb{R}^3\}$, where $f : \mathbb{R}^3 \to \mathbb{R}$ is a *Lipchitz continuous* function with a Lipchitz bound λ that must be supplied beforehand. The intersection point between a ray and the implicit surface is found by successive iteration of the equation $t_{i+1} = t_i + |f(\mathbf{x}(t_i))|/\lambda$, where t_i is the distance along the ray after the i-th iteration.

After the intersection point between a ray and the terrain has been found, a light model must be evaluated to obtain the colour intensity for the pixel through which the ray passes. This light model must account for the Rayleigh scattering of light in the atmosphere. The scattering occurs as photons bounce on air molecules and are redirected along the ray towards the observer. Rayleigh scattering is responsible for the blue colour of the sky under noon conditions, which subsequently turns red under sunset conditions. The scattering of light in the atmosphere is a complex phenomenon and a model is employed here that considers many simplifying assumptions [Nishita et al., 1993]. For example, multiple scattering events are ignored and so is the bending of rays due to the lensing effects that are caused by the varying atmospheric density. The scattering model requires the numerical integration of the optical depth and light extinction factors along the length of a ray, up to the point of intersection with the terrain.

Rendering an image by passing a single ray through each pixel leads to aliasing artifacts due to an insufficient sampling density. The image must be convolved with a low pass filter function in order to be properly anti-aliased. In practical terms, this image convolution is achieved with an anti-aliasing Monte Carlo integration technique [Gamito and Maddock, 2006]. Several rays are shot for each pixel, where the probability density of rays around the pixel is given by the anti-aliasing filter function. The final pixel colour is obtained by accumulating the colour of all the rays that are involved in the evaluation of the integral. The same concept is extended for image renderings that change through time, i.e. computer animations, by distributing the rays not only in space, around the pixel, but also in time, around the time instant when the image is taken. This produces an effect called *motion blur*, which is visible on terrain features that are close to the camera as the latter moves.

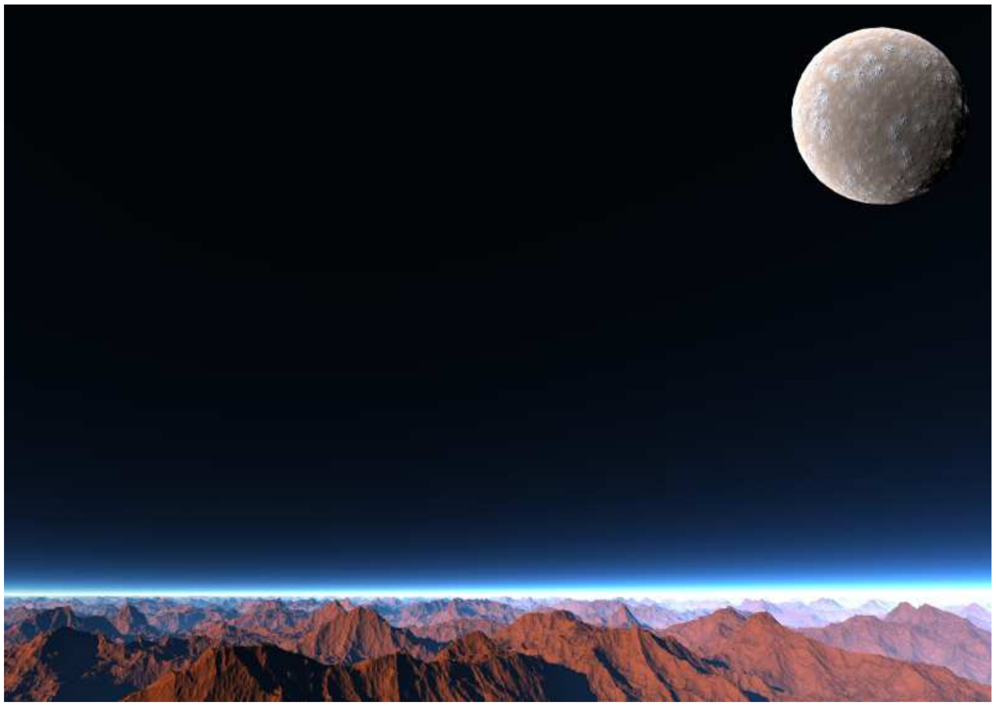

Figure 1: A computer rendering of a synthetic landscape. The surface of both the planet and the moon are generated with procedural fractal models. Rendering of the atmosphere is achieved through the solution of a Rayleigh scattering model.

Figure 1 shows an image that was synthesised with these three algorithms. The image was rendered with a resolution of 3740×2645 pixels on the Iceberg node, using the early version of the distributed ray tracer that existed before the work described in this report was undertaken. The original high resolution image can be downloaded from the address:

`http://www.dcs.shef.ac.uk/~mag/poster.jpg`

Anti-aliasing was implemented by shooting 100 rays per pixel. The parameters for atmospheric scattering were exaggerated, relative to the equivalent parameters in the Earth's atmosphere, to create a more dramatic effect. Nevertheless, the atmospheric lighting remains physically correct, subject to the model's simplifying assumptions.

3 Grid Computing and the White Rose Grid

Grid computing is a new computational model that exploits the availability of high performance computer nodes across a heterogeneous and geographically dispersed network. Many scientific and technological problems are known to have such a computational complexity that only the most powerful computers can solve them [Levin, 1989]. In the early years of computing, such powerful computers were built by individual companies using their own proprietary hardware and featuring thousands of processors inside a single computer. During subsequent years, a more flexible and cost-effective approach was taken whereby a large computing facility was assembled out of a collection of smaller computers that used commercially available processors, from companies like Intel or AMD, and that were connected together through a high speed network. Techniques for tapping the power of these computer clusters lead to the development of High Performance Computing (HPC) as a new field of Computer Science. Libraries that implement the Message Passing Interface (MPI) protocol allow a programmer to build applications that can be distributed across several networked computers [Gropp et al., 1999]. For HPC installations that feature multi-processor computers, where groups of processors have access to a common memory space, the OpenMP standard can also be used to implement parallelism inside an application [Chandra et al., 2000]. At a higher level, job schedulers like the Sun Grid Engine or Condor are put in control of HPC clusters to mediate the access of users to the resources and to manage the load on the cluster [Gentzsch, 2001; Thain et al., 2005].

HPC installations are normally found at universities and government funded research laboratories due to the high cost of acquiring and maintaining this type of hardware. Traditionally, access to these HPC installations was restricted to students and staff of each university or laboratory. More recently, a new trend has emerged where a group of institutions agrees on a protocol that allows accredited researchers to have Internet access to their joint set of HPC computing facilities. This joint set of facilities is called a *grid* and features potentially dozens of HPC nodes from different institutions that may be spread across a large geographical area. The emergence of grids lead to the development of *grid computing* as a new computational model that can be regarded as being one step above high performance computing. Issues that would not normally worry the user of a single HPC node become significant when dealing with grid applications. Such issues are related to the execution of common tasks like secure access or job management in a transparent manner across the grid. Ultimately, the issues that grid computing must solve arise as a consequence of the extreme level of heterogeneity that can be found in a grid. This is because each grid participating institution has its own type of HPC platform, has its own set of software applications and development tools running on that platform and implements its own management policies for users of that platform.

3.1 Grid Middleware

As part of the grid computing model, an intermediate layer of software, called *grid middleware*, must exist between a grid application and the grid nodes. Grid middleware addresses the problems created by the heterogeneity that is inherent to grids. It is responsible for providing a common access infrastructure to all grid services while, at the same time, hiding from the user the details of how these services are implemented in each grid node. The following list describes the most common services that are required from the grid:

Authentication and Secure Access Access of users to grid services must be properly authenticated. The communication between a grid application and the grid nodes must be encrypted, considering that it takes place over the Internet.

Resource Location A user must be able to query the grid to find diverse information such as what grid nodes are available, what services they offer, what is their hardware configuration and what is their load at any time.

Job Management With job management services, individual jobs or sequences of jobs can be issued to the grid. The state of these jobs can then be queried and jobs can be deleted, suspended or migrated to other nodes.

Data Retrieval Typically, after a job completes in a grid node, a file containing the output data is left on the node's file system. A grid application must be able to gather all the data files that were left on the grid nodes after a distributed computation has finished.

A standard called Open Grid Services Architecture (OGSA) is under development and specifies a set of requirements for grid services that compliant grid middleware is expected to obey [Foster et al., 2002]. One implementation of grid middleware that satisfies many of the requirements set forth in the OGSA is the Globus Toolkit [Foster and Kesselman, 1997]. The Globus Grid Security Infrastructure (GSI) manages user authentication and data encryption. User authentication in Globus is done with X509v3 digital certificates that are issued by a Globus recognised certification authority. When a user wishes to access grid services through Globus, he requests a grid proxy based on his certificate. Grid proxies have a limited validity. The user is given transparent and encrypted access to the grid while the proxy is valid. When a grid proxy expires a new proxy must be requested. Resource location in Globus is implemented by the Monitoring & Discovery System (MDS) that uses a Web Services interface. The Globus Grid Resource Allocation and Management module (GRAM) supports job management. GRAM exists as a collection of command line tools but it also provides language bindings for programming languages such as C/C++, Java and Python. Data retrieval in Globus is performed with its Global Access to Secondary Storage (GASS) module, which implements the GridFTP protocol.

In many applications, a grid user does not have to be aware of the individual HPC nodes that are part of the grid. The user should be able to launch distributed applications transparently across the grid as if he was working with a single virtual HPC installation. This degree of abstraction is achieved by having a *meta-scheduler* that uses the existing grid middleware to reroute the user jobs to individual grid nodes. The meta-scheduler knows which nodes are available at every moment and what kind of services they are able to provide. When a user submits jobs for some particular computational task, the meta-scheduler selects the best nodes on the grid for that type of task and passes the jobs to the relevant job schedulers. Meta-schedulers have to be built with specific grid applications in mind because each application has its own scheduling policies. The Globus Alliance, the same software group that develops the Globus Toolkit, also offers the GridWay Toolkit to help in the implementation of meta-schedulers [Hudo et al., 2005]. GridWay is an extra layer of grid middleware that exists on top of the Globus Toolkit. It uses the GRAM module to provide adaptive scheduling of jobs in response to changing grid conditions.

3.2 The White Rose Grid

The White Rose Grid is a consortium of three universities in the Yorkshire area. These are the Universities of Sheffield, Leeds and York. The grid currently provides a total of four HPC nodes, one called Iceberg from the University of Sheffield, two called Everest and Snowdon from the University of Leeds and one other node called Pascali from the University of York[1]. The Iceberg node is a cluster of 40 Sun Fire V20Z servers from Sun Microsystems with two

[1]There is an extra node at Leeds called Maxima but maintenance of this node has recently been discontinued. The node is also not accessible through the Internet. It must be accessed indirectly through Everest or Snowdon.

2.4 GHz AMD Opteron processors each plus another 20 Sun Fire V40Z servers, which are similar but have four Opteron processors instead. All the machines are connected through a low latency, high bandwidth, Myrinet network. The Everest node consists of 87 V20Z servers with two dual core 2.2 GHz AMD Opteron processors each plus 7 V40Z servers that use four dual core 2.2 GHz AMD Opteron processors. The Snowdon node is a cluster of 128 Intel servers, using dual 2.2 GHz or 2.4 GHz Intel Xeon processors. Both the Everest and the Snowdon nodes use a Myrinet 2000 network as their backbone. A description of the Pascali node at York is not presented here as access to this node was not available for this work. In total there are 820 processors (including processors in dual core chips) in the White Rose Grid, available to perform image rendering tasks. All these processors have access to significant memory resources, varying between 2 Gb and 32 Gb in size. The size of main memory, however, is not a constraint for the type of rendering application that is described here. Procedural models of terrain have a very small memory footprint since they are essentially procedures in a computer program. There is no need to store large arrays of geometric data such as vertices or triangles.

All the available nodes in the White Rose Grid have a reasonably homogeneous configuration, compared to what is normally expected from a grid, and have the following characteristics in common:

- Scientific Linux as the operating system.
- GNU development tools.
- Bash shell interpreter.
- Single sign on access through SSH with public/private key pairs.
- Single sign on access with the Globus Toolkit.
- Job scheduling with the Sun Grid Engine.

All nodes run Linux and use the Sun Grid Engine to schedule jobs. Once a public RSA encryption key is copied to each node, it is possible to invoke any Unix command remotely through the SSH protocol. Because of the relative homogeneity and the ease in accessing grid nodes from a remote application, it did not seem necessary to use the more complex Globus Toolkit. When running remote Unix applications in the White Rose Grid, the `ssh` command can successfully fill the role of grid middleware. The situation may change in the future, however, if this work is later ported to a larger grid such as the National Grid Service.

4 A Grid Application for Rendering Synthetic Landscapes

This report focuses on the development of a grid application on the White Rose Grid that implements the distributed rendering of sets of synthetically generated images. These images are then put together into a video sequence that can be played back with the appropriate video software. A meta-scheduler was written to coordinate the scheduling of the rendering jobs on all the available grid nodes. The meta-scheduler consists of a Python script that runs on a Linux desktop PC with Internet access to the grid. The Python scripting language was chosen because it allows the rapid development of new applications and also because it offers a rich set of system calls to interact with the operating system and the network.

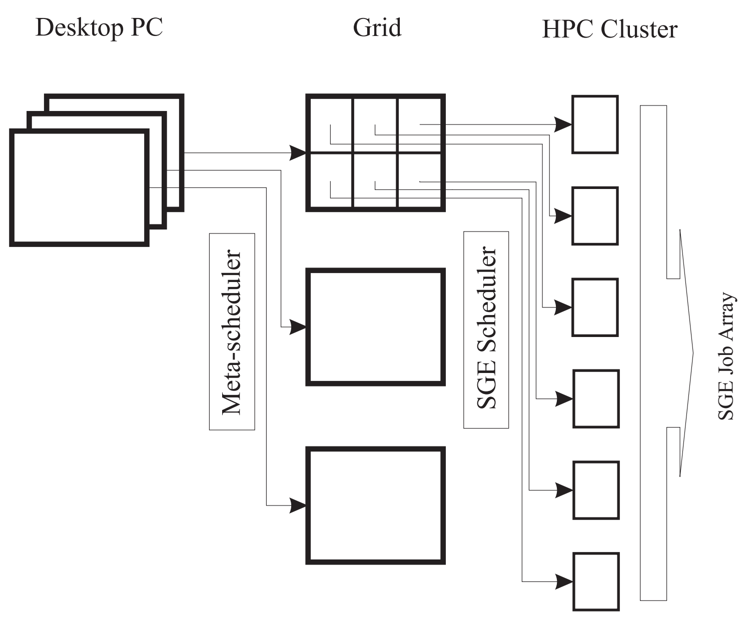

Figure 2: The two tiered distribution architecture of the grid application. The larger rectangles represent images. The smaller rectangles represent image tiles.

As Figure 2 shows, the grid application enforces a two tiered model of load distribution. In the first tier, images are individually distributed across the grid nodes. Each image is entirely rendered by the node to which it was assigned by the meta-scheduler. A load balancing scheme at the image level is the result of implementing a dynamic scheduling policy in the meta-scheduler. Grid nodes that are faster receive more images to render. Whenever a grid node becomes idle, the meta-scheduler assigns the next available image to it. In the second tier, load distribution is implemented by splitting an image into smaller rectangular regions, called tiles, and assigning these tiles to different processors of the HPC cluster that is installed at the grid node. The set of rendering jobs for all the tiles of an image forms a *job array* in the Sun Grid Engine (SGE) scheduler at the node. So, load distribution at the image level is handled by the meta-scheduler while load distribution at the tile level is handled by the job schedulers at each individual grid node. The SGE schedulers will attempt to distribute the tile rendering jobs in a fair way between all the processors of a cluster, keeping in mind the other users of the same cluster.

4.1 Application Deployment

The first step in setting up the grid is to deploy the ray tracing application to the grid nodes. Deployment of an application is often a source of problems in grid computing, especially when the application depends on third-party software libraries. It may happen that different grid nodes have slightly different versions of the same library. In this case, the user must tweak his code in different ways at each node to accommodate the idiosyncracies of the supporting software that resides there. This type of problem does not occur for the ray tracing application since it only requires support from the operating system libraries. Given that Scientific Linux is Posix compliant, the API calls that are required by the ray tracer can be found on all nodes.

The ray tracer is a C++ application that was written on a Windows laptop, inside the Cygwin emulating environment. It was written with the portability to Unix systems other than Cygwin in mind. The GNU set of portability tools (Autoconf, Automake and Libtool) was used to manage the software installation and configuration procedure. After unpacking the source code at each node, the command `./configure` was issued to generate the makefiles, followed by `make`. The source code compiled cleanly on all three nodes. The ray tracer is a command line tool that accepts several command line options. The line below shows an example of what a typical invocation of the ray tracer at one of the nodes might be:

```
gaea -r 800,600 -w 0,60,80,120 t=0.01
```

This command starts the rendering of an image with a resolution of 800×600 pixels but it only renders a tile of this image where the pixels have coordinates (i, j) that are constrained by $0 \leqslant i < 80$ and $60 \leqslant j < 120$. The output is an image file with a dimension of $(80 - 0) \times (120 - 60) = 80 \times 60$ pixels. If no image filename is given, as above, the ouptut image is sent to the standard output. The command line variable `t=0.01` indicates to the ray tracer that it should render a snapshot of an animation corresponding to the time instant $t = 0.01s$. The setting up of the terrain functions and the lighting conditions is currently hardwired in the application. This is clearly a limitation as editing the landscape requires changing the source code in the Windows laptop and migrating copies of the source to all the grid nodes again. In the future, a grammar parser will be included to read the landscape settings from a text file without having to recompile any code.

A set of six Bash scripts must also be installed on the grid nodes to ensure proper communication between the meta-scheduler and the local SGE schedulers. The list of the necessary scripts is as follows:

`gaearun` This is the script used by the Sun Grid Engine to start the rendering of a new tile at a processor. It invokes the `gaea` application, with the proper command line options.

`gaeasub` Submits a new job array that resulted from the subdivision of an image into tiles. This is basically a wrapper for the `qsub` SGE command. It passes the name of the `gaearun` script to the scheduler.

`gaeadel` Removes a job array, including all instances of that array that may be running at the time, from the scheduler. This is basically a wrapper for the `qdel` SGE command.

`gaeals` Returns a list of the image files that have been completed as part of the execution of a job array. This command can be invoked at any time, returning only the filenames for tiles that have finished rendering by that time.

`gaeaget` Returns the content of a given image file. The command basically sends the content of the file to its standard output, which is then piped back to the meta-scheduler.

`gaearm` Removes all image files associated with a given job array from the node's file system.

The scripts are quite portable since they rely only on the existence of a Bash shell interpreter (which all grid nodes have) and a minimal set of Unix command tools like `find`, `tail` and `rm`. Copies of the scripts are migrated to the grid nodes inside the source code package for the ray tracer. The original scripts are maintained at the Windows laptop, along with the Python script for the meta-scheduler, and are part of the ray tracer's development tree.

Node	**maxujobs**
Iceberg	10
Everest	unlimited
Snowdon	4

Table 1: The value of the `maxujobs` parameter at the three grid nodes.

4.2 Grid Resource Location

Information about the grid nodes is read on startup by the meta-scheduler from a local text file, called `grid.conf`. The text file lists the available nodes and further information, relative to each node, that is required by the meta-scheduler. The current contents of this static resource file are as follows:

```
# Node name             Username   Min_tsize  Max_jsize
iceberg.shef.ac.uk      acp04mog     2500      75000
everest.leeds.ac.uk     wrsmog       1600      75000
snowdon.leeds.ac.uk     wrsmog       5000      75000
```

The resource information required by the meta-scheduler, for each grid node, is the fully qualified address, the username of the account, the minimum size (in pixels) of a tile and the maximum size of a job array (correspondingly, the maximum number of tiles) that is allowed by the SGE scheduler at the node. The meta-scheduler uses the last two parameters during its initialisation to compute an optimal tile subdivision of the image for each node. The parameter `Max_jsize` is equal to the `max_aj_tasks` configuration parameter of the SGE schedulers. None of the grid nodes enforces a particular value for this parameter in their configuration files and it defaults to the value 75000. The `Min_tsize` parameter is chosen by the user and is loosely dependent on the maximum number of jobs that a user can run simultaneously on a node. This, in turn, is given by the `maxujobs` configuration parameter of the SGE schedulers. The rationale is that, if a user can only run a small number of jobs concurrently, there is no advantage in having many small tiles since most of them will have to be kept waiting in the scheduling queue. If the SGE `maxujobs` parameter is small, it is slightly more efficient to have a few larger tiles as it decreases the overhead of launching many small jobs on the cluster. The `maxujobs` at the three grid nodes is shown in Table 1. The content of this table motivated the values of the `Min_tsize` parameter that are used in the resource file.

The meta-scheduler performs a subdivision of the image into tiles for each grid node, based on the following list of criteria:

- The size of the tiles must be as small as possible but not smaller than `Min_tsize`.
- The total number of tiles must not exceed `Max_jsize`.
- The difference between the width and the height of the tiles must be as small as possible, i.e. the shape of the tiles must be as close as possible to a square.

The meta-scheduler computes all integer divisors of the image width and all integer divisors of the image height as part of its initialisation procedure. A list of all possible pairs of these two sets of divisors is then formed. Every element in this list is a potential candidate for the width and height of a tile. The list is sorted by increasing order of tile size and searched from the smallest to the largest size. The first element that obeys all criteria is the chosen tile size.

The data contained in the resource file is not likely to change frequently and, therefore, this static resource allocation scheme is adequate for our purposes. Situations where the data in the `grid.conf` file might change would be caused by one of the grid node system administrators reconfiguring the SGE scheduler by changing either its `maxujobs` or `max_aj_tasks` parameter. It could be possible to initialise the `Min_tsize` and `Max_jsize` parameters dynamically by querying the scheduler parameters of each node through a remote invocation of the SGE `qconf` command. This increased coding complexity does not seem warranted at this point, however.

4.3 Grid Access and Authentication

After installing an RSA public key at each grid node, it is possible to remotely invoke commands on that node through `ssh`, without having to interactively provide a password authentication every time. For example, to schedule an image rendering on the Iceberg node, a new job array must be submitted to that node's scheduler in the following way:

```
ssh acp04mog@iceberg.shef.ac.uk "~/gaeasub -t 1-192 \
  -v RESX=800,RESY=600,WINX=50,WINY=50,TIME=0.01 <filename>"
```

This specifies that a 800×600 resolution image is to be rendered after being split into 192 tiles, where each tile has a dimension of 50×50 pixels. The `-t` option specifies the range of tiles to be rendered, from tile 1 to tile 192. The `-v` option specifies a comma separated list of variables that are exported by the SGE scheduler as environment variables into the `gaearun` script. Remote invocation of the other shell scripts through the SSH protocol is performed in a similar manner.

The meta-scheduler launches these `ssh` commands through an `os.popen()` system call that is defined in the Python interpreter. The single argument to the `os.popen()` call is a string that contains a Unix command to be executed (a `ssh` command in this case). A Unix pipe is opened between the Python interpreter and the command. In particular, the standard output of the command is redirected back through the pipe and can be read by the Python script. For example, the remote invocation of the `gaeasub` script shown above can be written in Python in the following way:

```
pipe = os.popen("<string for ssh gaeasub script>")
output = pipe.read()
pipe.close()
jobID = parse(output)
```

The Python variable `output` stores the standard output of the `gaeasub` command, which among other things indicates the job ID of the newly created SGE job array. If the `ssh` command executes without error, the job ID can then be parsed from the `output` variable. This ID may be required later, should the meta-scheduler wish to delete the job array.

4.4 Implementing a Polling Strategy

The initial idea at the start of this project was to have a client-server architecture between the meta-scheduler and the grid nodes. The meta-scheduler would keep a socket open, listening for incoming packets. Each grid node processor, once it finished rendering a tile, would open a connection to this socket and send over the computed tile data. The meta-scheduler would receive the rendering results in real time from all the processors in the grid and this would allow it to schedule new rendering jobs on the fly. There were, however, difficulties in implementing this architecture because of the firewalls at Sheffield and Leeds that do not allow machines from inside their HPC clusters to connect to arbitrary addresses on the outside. There was still the possibility of keeping a daemon proxy running inside each HPC cluster. The cluster processors would send their packets instead to this proxy and it would relay the packets back to the meta-scheduler through a SSH tunnel. The proxy would have to be left running outside the control of the Sun Grid Engine (otherwise it would be killed after a maximum allowable running time had elapsed) but this would represent a breach of policy of the HPC clusters, where every job must be supervised by the SGE in order to maintain fairness between users.

Rather than using a client-server architecture, the meta-scheduler implements a polling strategy instead and checks the state of the tile renderings on all grid nodes at regular intervals. The polling interval has a default duration of five minutes but this can be changed through a command line option when the meta-scheduler is launched. The rendering state of each grid node is remotely queried with the `gaeals` script, which returns a list of all the filenames for the tiles that have been completed so far. This allows the meta-scheduler to detect new tiles that have been completed since the last poll to the same grid node was performed. When all the tiles of an image that is being rendered on a grid node have been completed, the meta-scheduler sends instructions through `gaeasub` for a new image to be scheduled on that node. This polling mechanism works well in situations where tiles take, on average, significantly longer to render than the duration of the polling interval since the rendering state of a grid node will undergo only small changes between polls.

4.5 Implementing a Schedule-Ahead Policy

The polling mechanism introduces only one source of inefficiency, which is related to the scheduling of new images. Consider the situation where a grid node finishes rendering an image shortly after a poll from the meta-scheduler was completed. For the remainder of the following five minutes the grid node is going to be idle with respect to the grid rendering application. The meta-scheduler will only detect the completion of the image and schedule a new image after the five minutes have elapsed. The problem is made worse by the fact that, during those five minutes of idle time, jobs from other users will occupy the processors that have since become available. When the new image is scheduled, it will have to wait for the processors to become available again. If the new image had already been present in the scheduling queue by the time the previous image finished it might have been able to recapture some of the same processors. This is because the SGE uses a dynamic priority strategy and the job array for the new image might have a higher priority that some of the other user jobs in the system. The rendering efficiency of the grid nodes would be improved in this way without violating the principles of fairness to other users that are always enforced by the SGE.

The inefficiency that stems from scheduling new images under control of the meta-scheduler's polling mechanism is alleviated by implementing a schedule-ahead policy. This basically means that the meta-scheduler speculatively assigns a new image in advance to a grid node while a previous image is still rendering at the same node. At any time there will always be two job arrays in a scheduling queue. One job array will be undergoing rendering (with some of the tiles in the array being rendered while the remaining tiles wait for available processors) and the other job array will stay in wait until the previous array finishes. The following partial output of the `qstat` command shows a typical scheduling state in the Snowdon node, in what concerns the grid rendering application:

```
job-ID  prior   name     user   state  ja-task-ID
-------------------------------------------------
177620  0.51000 gaearun  wrsmog r      3
177620  0.51000 gaearun  wrsmog r      4
177620  0.51000 gaearun  wrsmog r      2
177620  0.51000 gaearun  wrsmog r      1
177620  0.00000 gaearun  wrsmog qw     5-48:1
177621  0.00000 gaearun  wrsmog qw     1-48:1
```

The job array with ID 177620 has image tiles 1 to 4 being rendered, all with a priority of 0.51, while the remaining tiles 5 to 48 remain waiting. There is another job array with ID 177621 that will start rendering once the last tiles from the previous job array begin to complete. The `maxujobs` parameter for Snowdon is 4 (recall Table 1) and, therefore, only four tiles can be rendered concurrently on this node. There will be a brief transitional period when the four processor slots available to user `wrsmog` will be shared between the last tiles of job 177620 and the first tiles of job 177621. Once all the remaining tiles from job 17760 finish, a new job array will be placed on the queue by the meta-scheduler, sometime within a five minute period.

The schedule-ahead mechanism begins to break down for fast enough grid nodes that can render all the image tiles in an amount of time comparable to the meta-scheduler's polling interval. This is more likely to happen in the Everest node, since it does not impose any limit on the number of jobs running concurrently[2]. In this type of situation, the meta-scheduler may not have enough time to place another image on the scheduling queue before the previous image completes rendering. To solve this problem one must either decrease the polling interval or increase the number of simultaneous job arrays that must be kept in the queue.

4.6 Retrieving Computation Results

The outcome of a tile that has finished rendering on a grid node is an image data file, residing in the node's file system. The data file is written in the PPM format, a minimalistic image file format that basically contains a header, indicating the tile dimensions, followed by the raw pixel data [Murray and vanRyper, 1994]. Despite its simplicity, the PPM image format is

[2]Even in grid nodes where the `maxujobs` parameter imposes no restriction on the number of simultaneous jobs from the same user, the load imposed by other users always constrains this number in practice. In the very best of situations, the number of active tile renderings will always be constrained by the `max_aj_instances` parameter, the maximum number of tasks from the same job array that can run simultaneously.

accepted by virtually all Unix-based image viewers. The filename of the PPM tile data obeys the following convention, where the parameters between angle brackets signify fields with variable information:

```
<filename>_<imgnum>_<tilenum>.ppm
```

The `imgnum` field indicates the image number in the sequence of images that constitutes the computer animation. The `tilenum` field indicates the tile number inside the image `imgnum`. The `gaeals` script retrieves lists of filenames that obey this naming convention. The meta-scheduler then invokes the `gaeaget` script for individual files. The content of a file is sent from the standard output of `gaeaget`, through the `ssh` command and through the Unix pipe that was set up by the `os.popen()` call until it arrives at the meta-scheduler, where it is temporarily kept in a string variable. The meta-scheduler then stores the results of the tile rendering in the local image file:

```
<filename>_<imgnum>.ppm
```

The meta-scheduler opens the image file and uses the value of the `tilenum` field to move the file pointer to the correct position inside the image with a `file.seek()` Python call. The pixel data from the tile is then transferred to its correct place in the image[3].

5 Results

Figure 3 shows four frames from a computer generated animation of a camera flyby over the same synthetic landscape of Figure 1. The camera is travelling forward with a constant speed of 60 km/h and at a constant altitude of 25 metres[4]. The animation has 500 frames, which, at a rate of 25 frames per second, corresponds to 20 seconds of playback time. At the speed the camera is travelling, the animation would need to have a duration of almost a month of continuous playback time for a complete circumnavigation of the planet to be achieved. In this impractical scenario, the first and the last frames of the animation would be equal and the animation could be looped indefinitely. The animation can be downloaded as a Quicktime movie file from the address[5]:

```
http://www.dcs.shef.ac.uk/~mag/flyby.mov
```

An estimate of the image rendering times on the White Rose Grid was obtained by averaging the rendering times for the first hundred frames at every grid node. The results, referenced as T_I, T_E and T_S on the Iceberg, Everest and Snowdon nodes, respectively, are shown in Table 2. It must be kept in mind that, in a general situation, the camera can follow an arbitrarily complex path over the landscape and the frame rendering times can change significantly with different camera positions. Some frames may have a higher visual complexity and take longer to render

[3]The actual procedure of pasting a tile pixel data onto an image is a bit more complex. Because PPM images store the pixel array in a row major format, a `file.seek()` call must be made for every row of pixels in the tile.

[4]Although the mountains look imposing in the renderings of Figure 3, they are actually only a few metres high.

[5]A free Quicktime player, available from `www.apple.com`, must be installed before playing the animation.

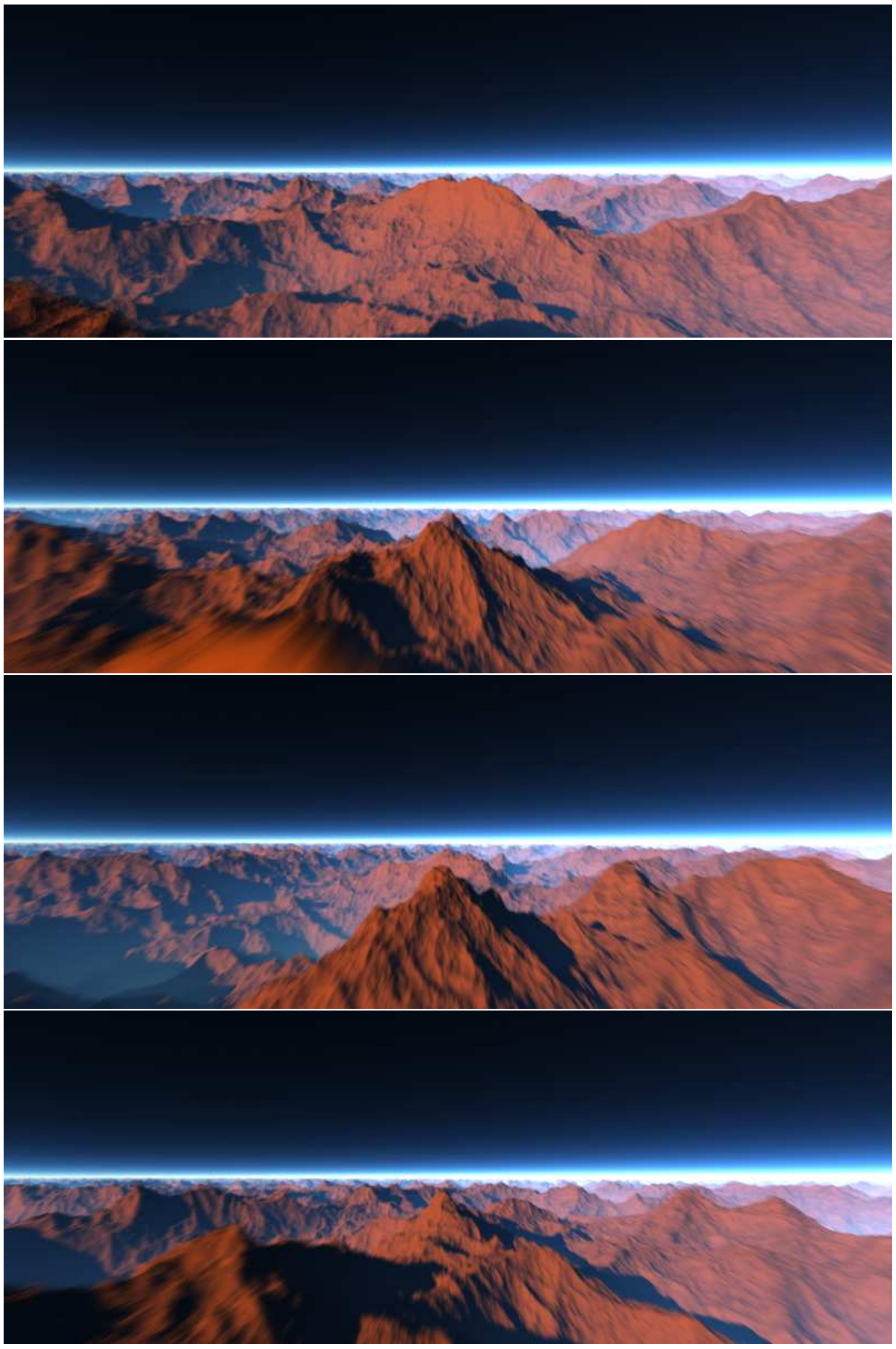

Figure 3: A sequence of frames showing a camera flyby over a procedural planetary landscape at constant altitude and speed. The motion blur effect is visible in the lower part of the frames.

Node	Time	Value
Iceberg	T_I	56
Everest	T_E	24
Snowdon	T_S	244

Table 2: The rendering time, in minutes, for the first hundred frames in the animation.

that others. This is not so much the case for the animation shown in Figure 3 where the camera merely moves forward at a constant altitude and the image complexity can be considered essentially constant for all frames. Factors that contribute to the variability in rendering times for the animation of Figure 3 are more external, being dependent on the load imposed on the grid by other users, rather than internal, considering that all frames have approximately the same complexity.

It is clear from the results in Table 2 that the rendering times are heavily influenced by the `maxujobs` scheduling parameters that are shown in Table 1. The Everest node has no restriction on the maximum number of simultaneous jobs and this leads to substantially smaller rendering times. While performing the timing measurements for Table 2 it was noticed that between 72 to 77 tiles were being rendered simultaneously on Everest. The equivalent number was 10 for Iceberg and 4 for Snowdon, which is consistent with the values in Table 1. Let us assume for the sake of discussion that the values in Table 2 did not change while rendering all the 500 frames of animation on the grid. In the time that it takes for Snowdon to render a frame, Iceberg will have rendered $T_S/T_I = 4.36$ frames and Everest will have rendered $T_S/T_E = 10.17$ frames. The number of frames N_S rendered by Snowdon as part of the 500 frame animation obeys:

$$\left(1 + \frac{T_S}{T_I} + \frac{T_S}{T_E}\right) N_S = 500.$$

The result is $N_S \approx 32$, approximated to the nearest integer. The number of frames rendered by the other two nodes is $N_I = (T_S/T_I)N_S \approx 140$ and $N_E = (T_S/T_E)N_S \approx 328$. The percentage of frames rendered by the Snowdon, Iceberg and Everest nodes is 6.4%, 28.0% and 65.6%, respectively. The total rendering time is $\max(N_S T_S, N_I T_I, N_E T_E) = 7872$ minutes or, approximately, 5 days and 11 hours. One can verify from these results that the White Rose Grid is asymmetrical in what concerns the grid rendering application. The Everest node does the bulk of the work, with the Snowdon and Iceberg nodes giving a smaller contribution.

The actual rendering time for the animation is larger than the previous estimates. The rendering was started on the 13th of June at 19:50 hours and it was complete by the 20th of June at 11:18 hours, which corresponds to 6 days and 15 hours. The discrepancies between this value and the estimates derived from Table 2 have several causes:

Variable load conditions The load on the grid nodes varied over the course of the rendering, with rendering jobs having to be suspended for arbitrary lengths of time. The meta-scheduler could not proceed to schedule another image on a node while it was still waiting for tiles from a previous image that were in a suspended state on that node.

Grid node lockouts When scheduling an image to a node one must specify a parameter called `h_rt` that indicates to the scheduler how long each tile rendering is expected to run.

The SGE will kill a tile rendering job once its running time exceeds `h_rt`. The meta-scheduler will then be left waiting indefinitely for the results from the tile that was killed and it will not be able to schedule a new image on that node. This situation can go on undetected for several hours. Once it is detected, it becomes necessary to re-initialise the grid application and, as a consequence, images that were being rendered at the time of re-initialisation have to be rendered again.

Application development in parallel with grid rendering The animation was used to debug the grid rendering application. Whenever a bug was found, the rendering would have to be stopped, the application would be debugged, and the rendering was then re-initialised from where it had previously left off. This delay was especially significant after the need for a schedule-ahead mechanism was detected. The rendering was stopped for several hours as the application was extended to include the results from Section 4.5.

The evolution of the grid rendering application was observed during the week that it took to finalise the animation and the number of frames completed by each node was verified to be in agreement with the estimated values, despite the elements of variability described above. Nevertheless, it would be useful to render the 500 frame animation again, this time taking the care to log such data as the load on the three nodes and the evolution of the rendering times T_I, T_E and T_S as the animation progressed. This would allow a better analysis of the grid performance, compared with the simple estimates obtained from Table 2 that are based on averaging the rendering times for a small number of frames. Considering that re-rendering the animation will likely take several days, these new results will have to be presented in a future opportunity.

6 Conclusions

The distributed rendering model presented in this report has the potential to significantly reduce the rendering time of complex computer animations. The current conditions on the White Rose Grid did not allow the distributed model to be fully realised. Most of the work was performed on the Everest node with only a small contribution from the Iceberg and Snowdon nodes. It is not so much the case that Iceberg and Snowdon are inefficient. The author has previously rendered individual high resolution images on Iceberg (the image in Figure 1 being one example) that were completed overnight. It is rather the case that Everest is much more efficient than the other two nodes, thus creating an imbalance in the distributed model. The inclusion of the York node, which was not considered in this work, may change the situation. The extension of this work to the National Grid Service may also bring in new nodes that are similar to Everest, allowing a more equitable distribution of the work load. The extension to the National Grid Service, however, will require changing the current SSH based remote scheduling mechanism by a Globus Toolkit based mechanism.

One possibility of increasing the throughput of the Iceberg and Snowdon nodes would be to implement parallelism based on MPI, rather than relying on job arrays [Gropp et al., 1999]. The constraints imposed on the maximum number of processors that a single MPI job can allocate are not as stringent as the constraint on the maximum number of simultaneous jobs. This approach, however, can only be regarded as a hack. The purpose of MPI is to supply a series of features that enable multiple processors to synchronise amongst themselves and to communicate by passing messages. None of this features would be required in the case of a ray tracing application where each image tile is computed independently. The sole purpose of an MPI based approach in this situation would be to defeat the constraint imposed by the `maxujobs` parameter. This might not even be more efficient because MPI jobs that use a large number of processors may have to stay in the queue for a significant amount of time, waiting for the required number of processors to become available. The long scheduling delay may not compensate for the reduced image rendering time. Yet another argument against a MPI approach is that it requires two versions of the ray tracer to be maintained: one standalone and one that is MPI based. The current approach based on job arrays uses the same ray tracer source code that is used for standalone computers, making code maintenance easier.

The design of the current grid rendering tool can be considered as a pattern for similar computing problems. In the most general terms, the current design solves computational problems that can be split into smaller and independent tasks. The pattern favours a two-tiered distribution model, where tasks can themselves be split into smaller and still independent sub-tasks. Tasks are distributed across grid nodes while sub-tasks are distributed among the processors of a node. This two-tiered model, however, is not a requirement. If tasks cannot be further split they can still be arbitrarily grouped. The task groups are then distributed to the nodes. One example from Computer Graphics were this design model can be applied is in the implementation of a distributed version of the Reyes image rendering architecture [Cook et al., 1987]. A Reyes image renderer works by splitting the geometry (which can be made of polygon meshes, NURBS patches or subdivision surfaces) into progressively smaller fragments. When a fragment becomes much smaller than the size of a pixel it is called a *micropolygon*. It is then passed to the shading pipeline for rendering. Reyes is used by major animation studios such as Pixar for the rendering of their computer animation feature films.

The Reyes algorithm can be distributed by partitioning the image space into tiles, similarly to the ray tracing algorithm described in this report, and having each processor handle its own image tile[6]. Each processor will only split geometry data whose bounding box overlaps with the processor's tile. There are two additional steps in the implementation of a distributed Reyes renderer that were not necessary for the ray tracing of procedural surfaces. During the first step, all the geometry data must be transmitted to the grid nodes, possibly using a `scp` command. The second step is a pre-computation that must be performed before the rendering work at a grid node is distributed to its processors. This pre-computation step computes, among other things, hierarchies of bounding boxes for the geometry. These bounding boxes are required in order for each processor to know which geometry elements it should be concerned with. The two steps just described can be added with some extra effort to the distributed model that has been developed for ray tracing procedural landscapes.

[6]Pixar has a render farm made of 1024 Intel servers with 2.8 GHz Xeon processors and is certain to have a distributed version of the Reyes rendering tool. Details of how this distribution is performed are not known.

7 Further Developments

There are two aspects of the distributed rendering model that deserve improvement. One is the schedule-ahead mechanism that should be able to handle grid nodes with vastly different rendering times. While the current schedule-ahead mechanism works well on the Iceberg and Snowdon nodes, it cannot keep pace with the fast rendering times of the Everest node. This would be solved by keeping two or more job arrays waiting on the Everest queue instead of just one. At the end of a polling interval, the meta-scheduler would issue the necessary number of images to Everest to ensure the desired number of waiting job arrays would be fulfilled. The number of waiting job arrays for each queue could be specified on the grid resource file `grid.conf`. Iceberg and Snowdon would have a value of 1, since this has worked well so far, while Everest would have a value of 3 or 4. With this degree of occupancy in the Everest queue, the probability of Everest becoming idle would be greatly reduced.

A second and more important aspect that needs improvement is the proper handling of grid lockouts by the meta-scheduler. Currently, the `h_rt` parameter must be estimated by the user before the meta-scheduler is launched. It is hardwired in the header of the `gaearun` shell script and is therefore the same for all image tiles. It is very difficult to predict what the value of `h_rt` should be because it depends on the complexity of the surface that is visible through a tile. Tiles that only see background sky will render much faster than tiles that focus on terrain features. Assigning an over-conservative estimate of the tile rendering time to `h_rt` would solve the problem since it would guarantee that even the slowest of the tile renderings would not be killed by the scheduler. This, however, is not a practical solution because it will make it more difficult for the tile jobs to become active. If the SGE scheduler is given a job with a long predicted run time, it will attempt to schedule faster jobs first. A better way to handle this situation is for the meta-scheduler to monitor the running jobs on the grid nodes and to detect jobs that were killed without producing a full tile rendering. The meta-scheduler would then examine the incomplete tile pixel data and re-issue the same tile jobs at the point where they stopped rendering.

A Python Source Code

The source code for the meta-scheduler, written in the Python scripting language, is as follows:

```python
import re
import os
import sys
import signal
import getopt
import curses
import curses.wrapper

from math import log10
from time import sleep
from time import ctime
from os.path import isfile
```

```
class Node:
  def __init__(self,hostname,address, \
               username,factors,res,minsize):
    def size(i,j):
      if i[2] <= j[2]:
        return i
      else:
        return j
    def aspect(i,j):
      if abs(i[1] - i[0]) <= abs(j[1] - j[0]):
        return i
      else:
        return j
    f = filter(lambda i: i[2] >= minsize,factors)
    m = reduce(size,f)
    f = filter(lambda i: i[2] == m[2],f)
    m = reduce(aspect,f)
    self.job = []
    self.tile = m
    self.donetiles = []
    self.address = address
    self.hostname = hostname
    self.username = username
    self.offset = 9 + int(log10(res[0])+1) + \
                      int(log10(res[1])+1)
    self.tilesize = 9 + int(log10(m[0])+1) + \
                        int(log10(m[1])+1) + \
                        3*m[2]
    self.numtiles = res[0]*res[1]/m[2]
  def connect(self,command):
    pipe = os.popen("ssh " +                \
                    self.username + '@' +   \
                    self.address + " '~/" + \
                    self.hostname + '/' +   \
                    command + "' 2>/dev/null")
    output = pipe.read()
    if pipe.close() != None:
      raise IOError
    return output

class Grid:
  def __init__(self,filename):
    self.res = (0,0)
    self.fps = 25.0
    self.omega = 0.0
    self.imgnum = 0
    self.numimgs = 0
    self.imgsize = 0
    self.doneimgs = 0
    self.filename = filename
    self.node = []
  def __open(self,node):
    node.file = open(self.filename + '_' +          \
                     str(node.job[0][0]).zfill(3) + \
                     ".ppm",'w')
    node.file.write("P6\n" +                  \
                    str(self.res[0]) + ' ' +  \
                    str(self.res[1]) + '\n' + \
                    "255\n")
    for i in xrange(0,3*self.imgsize):
      node.file.write('\0')
    node.file.flush()
    node.donetiles = []
  def __close(self,node):
    node.connect("gaearm " + self.filename + ' ' + \
                            str(node.job[0][0]));
    log.write("Finished image " + str(node.job[0][0]) + \
              " on node " + node.hostname +             \
              " at " + ctime() + '\n')
    log.flush()
    del node.job[0]
    node.file.close()
    self.doneimgs += 1
  def __launch(self,node):
    while self.imgnum < self.numimgs:
      filename = self.filename + '_' + \
                 str(self.imgnum+1).zfill(3) + ".ppm"
      if not isfile(filename):
        time = self.imgnum/self.fps + 0.01
        try:
          output = node.connect("gaeasub -t 1-" +             \
                    str(self.imgsize/node.tile[2]) + " -v " + \
                    "FPS=" + str(self.fps) + ',' +            \
                    "IMG=" + str(self.imgnum+1) + ',' +       \
                    "RESX=" + str(self.res[0]) + ',' +        \
                    "RESY=" + str(self.res[1]) + ',' +        \
                    "WINX=" + str(node.tile[0]) + ','         \
                    "WINY=" + str(node.tile[1]) + ',' +       \
                    "TIME=" + str(time) + ',' +               \
                    "OMEGA=" + str(self.omega) + ',' +        \
                    "FILENAME=" + self.filename + " ˜/" +     \
                     node.hostname + "/gaea")
        except IOError:
          log.write("Could not connect to " + \
                    node.hostname + "\n")
        else:
          self.imgnum += 1
          node.job.append((self.imgnum, \
                           re.search("\d+",output).group()))
          log.write("Starting image " + str(self.imgnum) + \
                    " on node " + node.hostname +          \
                    " at " + ctime() + "\n");
        log.flush()
        return
      self.imgnum += 1
  def __poll(self,node):
    output = node.connect("gaeals " +                 \
                          self.filename + ' ' +       \
                          str(node.job[0][0]) + ' ' + \
                          str(node.tilesize))
    for line in re.finditer("(.*)\\n",output):
      file = line.group(1)
      match = re.search("_(\d+)_(\d+)_(\d+)",file)
      tile = (match.group(2),match.group(3))
      if not tile in node.donetiles:
        output = node.__command("gaeaget " + file)
        i = int(tile[1]) - 1
        j = self.res[1]/node.tile[1] - int(tile[0])
        for k in xrange(0,node.tile[1]):
          node.file.seek(node.offset + \
                         3*(node.tile[0]*i + \
                            self.res[0]*(node.tile[1]*j + k)))
          node.file.write(output[3*node.tile[0]*k:\
                                 3*node.tile[0]*(k+1)])
        node.donetiles.append(tile)
    node.file.flush()
  def __kill(self,node):
    joblist = ""
    for job in node.job:
      joblist += job[1] + ' '
    node.connect("gaeadel " + self.filename + ' ' + joblist)
    node.file.close();
    os.remove(self.filename + '_' + \
              str(node.job[0][0]).zfill(3) + ".ppm")
  def schedule(self,stdscr,col):
    for n in self.node:
      if len(n.job):
```

```
        self.__poll(n)
        if len(n.donetiles) == n.numtiles:
          self.__close(n)
          if len(n.job):
            self.__open(n)
            self.__launch(n)
      else:
        self.__launch(n)
        if len(n.job):
          self.__open(n)
          self.__launch(n)
      if len(n.job):
        stdscr.addstr(col,40,str(n.job[0][0]).rjust(5),\
                      curses.color_pair(3))
        percent = 100.0*len(n.donetiles)/n.numtiles
        stdscr.addstr(col,50,"%6.2f%%" % percent,\
                      curses.color_pair(3))
      else:
        stdscr.move(col,40)
        stdscr.clrtoeol()
      col += 1
    stdscr.refresh()
  def shutdown(self):
    for n in self.node:
      if len(n.job):
        self.__kill(n)

def sighandler(signum, frame):
  raise KeyboardInterrupt

def usage(message = ""):
  print "Usage: " + sys.argv[0] + " \n\
    [-d delay (min)] (default: 5 min)\n\
    [-f framerate (fps)] (default: 25 fps)\n\
    name resx resy omega #images"
  if message:
    print "Error: " + message
  sys.exit(2)

def factorise(n):
  divisors = []
  for i in xrange(2,int(n/2)+1):
    if n%i == 0:
        divisors.append(i)
  divisors.append(n)
  return divisors
```

```
def main(stdscr):
  curses.curs_set(0)
  curses.init_pair(1,curses.COLOR_GREEN,\
                   curses.COLOR_BLACK)
  curses.init_pair(2,curses.COLOR_YELLOW,\
                   curses.COLOR_BLACK)
  curses.init_pair(3,curses.COLOR_WHITE,\
                   curses.COLOR_BLACK)
  stdscr.addstr(4,10,"Gaea Grid: $Revision: 495 $",\
                   curses.A_BOLD|\
                   curses.color_pair(1))
  col = 6
  for n in grid.node:
    stdscr.addstr(col,10,n.hostname + ":\t" +               \
                       str(grid.res[0]/n.tile[0]) + 'x' + \
                       str(grid.res[1]/n.tile[1]) +       \
                       " tiles of dimension (" +          \
                       str(n.tile[0]) + ',' +             \
                       str(n.tile[1]) + ')',              \
                       curses.color_pair(3))
    col += 1
  col += 1
  stdscr.addstr(col,10,"Node",curses.A_BOLD|\
                            curses.color_pair(2))
  stdscr.addstr(col,40,"Image",curses.A_BOLD|\
                             curses.color_pair(2))
  stdscr.addstr(col,50,"Percent",curses.A_BOLD|\
                               curses.color_pair(2))
  col += 1
  datacol = col
  for n in grid.node:
    stdscr.addstr(datacol,10,n.address,curses.color_pair(3))
    datacol += 1
  stdscr.refresh()
  while grid.doneimgs < grid.numimgs:
    signal.signal(signal.SIGINT,signal.SIG_IGN)
    grid.schedule(stdscr,col)
    signal.signal(signal.SIGINT,sighandler)
    sleep(60.0*delay)

try:
  opts,args = getopt.getopt(sys.argv[1:],"d:f:m:")
except getopt.GetoptError:
  usage()
```

```python
delay = 5.0
cfgfilename = "grid.conf"

for o,a in opts:
  if o == "-d":
    try:
      delay = float(a)
    except ValueError:
      usage("Invalid delay was specified!")

if len(args) == 5:
  grid = Grid(args[0])
  for o,a in opts:
    if o == "-f":
      try:
        grid.fps = float(a)
      except ValueError:
        usage("Invalid frame rate was specified!")
  try:
    grid.res = int(args[1]),int(args[2])
  except ValueError:
    usage("Invalid image resolution was specified!")
  grid.imgsize = grid.res[0]*grid.res[1]
  try:
    grid.omega = float(args[3])
  except ValueError:
    usage("Invalid angular speed was specified!")
  try:
    grid.numimgs = long(args[4])
  except ValueError:
    usage("Invalid number of images was specified!")
  divisors = factorise(grid.res[0]), \
             factorise(grid.res[1])
  divisors = [(x,y,x*y) for x in divisors[0]  \
                        for y in divisors[1]]
  try:
    file = open(cfgfilename)
    for line in file:
      if line == "\n":
        continue
      if line[0] == '#':
        continue
      token = line.split()
      if len(token) != 3:
        print "Incorrect node specification:\n" + line
        continue
      name = token[0].split('.',1)
      if len(name) != 2:
        print "Hostname must be fully qualified:\n" + line
        continue
      try:
        numjobs = long(token[2])
      except ValueError:
        print "Incorrect specification for numjobs:\n" + line
        continue
      minsize = float(grid.imgsize)/numjobs
      grid.node.append(Node(name[0],  \
                            token[0], \
                            token[1], \
                            divisors, \
                            grid.res, \
                            minsize))
  except IOError:
    print "Error reading grid configuration file!"
    sys.exit(1)
  else:
    file.close
else:
  usage()

log = open("grid.log",'w')

try:
  curses.wrapper(main)
except KeyboardInterrupt:
  grid.shutdown()

log.close()
```

References

R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2000. ISBN 1-55860-671-8.

R. L. Cook, L. Carpenter, and E. Catmull. The Reyes image rendering architecture. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 95–102. ACM Press, July 1987.

I. T. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

I. T. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, 2002.

M. N. Gamito and S. C. Maddock. Anti-aliasing with stratified B-spline filters of arbitrary degree. *Computer Graphics Forum*, 25(2):163–172, June 2006.

W. Gentzsch. Sun Grid Engine: Towards creating a compute power grid. In *Cluster Computing and the Grid (CCGRID '01 Proceedings)*, pages 35–36. IEEE Computer Society, May 2001.

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. The MIT Press, 2nd edition, 1999. ISBN 0-262-57104-8.

J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(9):527–545, 1996. ISSN 0178-2789.

E. Hudo, R. S. Montero, and I. M. Llorente. The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing: Practice and Experience*, 6(3):1–8, 2005.

E. Levin. Grand challenges to computational science. *Communications of the ACM*, 32(12): 1456–1457, Dec. 1989.

R. Marshall, R. Wilson, and W. Carlson. Procedure models for generating three-dimensional terrain. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 154–162. ACM Press, July 1980.

J. D. Murray and W. vanRyper. *Encyclopedia of Graphics File Formats*. O'Reilly & Associates, Inc., July 1994. ISBN 1-56592-058-9.

F. K. Musgrave. Mojoworld: Building procedural planets. In D. S. Ebert and F. K. Musgrave, editors, *Texturing & Modeling: A Procedural Approach*, chapter 20, pages 565–615. Morgan Kauffman Publishers Inc., 3rd edition, 2003. ISBN 1-55860-848-6.

T. Nishita, T. Sirai, K. Tadamura, and E. Nakamae. Display of the earth taking into account atmospheric scattering. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 175–182. ACM Press, Aug. 1993.

D. R. Peachey. Building procedural textures. In D. S. Ebert and F. K. Musgrave, editors, *Texturing & Modeling: A Procedural Approach*, chapter 2, pages 7–94. Morgan Kauffman Publishers Inc., 3rd edition, 2003. ISBN 1-55860-848-6.

D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency: Practice and Experience*, 17(2-4):323–356, 2005.

T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.