Helios: Sunflow Grid Rendering

Helios enables the distributed rendering of 3D models building on open source technologies that are all 100% pure Java. The rendering engine is Sunflow with the distributed computations managed by the JGrid service-oriented Grid system that uses the Jini technology as its base.

This software kit enables the distributed rendering of 3D models building on a number of open source technologies that are all implemented in 100% pure Java. The rendering engine is the Sunflow open source rendering system for photo-realistic image synthesis, built around a flexible ray tracing core and an extensible object-oriented design. The distributed computations are managed by the JGrid service-oriented Grid system that uses the Jini technology as its base.

Using this software kit you can easily involve a number of interconnected computers into the rendering computations resulting in much lower rendering times. Due the Java technology the system is platform independent thus capable of running atop a heterogeneous environment. Moreover the Jini technology enables such a dynamic behavior that makes it possible to add and remove computers at runtime without interrupting computations and without any configuration on the client side. Using this software kit you do not require a dedicated cluster of identical computers, but a varying number of heterogeneous (both in architecture and OS) computers can be used on demand.

In this pdf you can find information how to install and set up your render farm and you can also download the latest release.

Introduction

Although this distributed rendering software kit was bundled to make the installation and usage as easy as possible, I think, without at least a minimal understanding what happens behind the scenes one cannot leverage the full potential of the system. This page shortly introduces the Jini technology, then describes the architecture of the rendering environment and a typical rendering scenario, and finishes with some practical advices how to design and set up your own renderfarm.

Jini

Jini is distributed object technology developed by Sun Microsystems Inc. from 1999. Jini provides tools and mechanisms to build a robust and dynamic distributed system where every component is either a service or a consumer of a service (called client from now). Jini services are described with plain Java interfaces and all service operations can be accessed via those interfaces. Jini provides a Service Oriented Architecture (SOA) where the three main components are the Service, the Client and the Registry which is called Lookup Service in Jini terminology. The relationship of these entities is show in figure 1. When a Service starts up it registers its proxy object with the Lookup Service which is a plain Java object implementing the service interfaces and wraps a remote reference to the service backend. The proxy object hides all communication details to the service backed and makes Jini protocol independent, since any communication protocol can be implemented between the proxy and the

service backend. After registering with the Lookup Services clients can lookup the suitable service based on the interface description. Moreover clients can subscribe for remote events that are triggered when the required service appears or disappears, so they can react quickly to any changes.

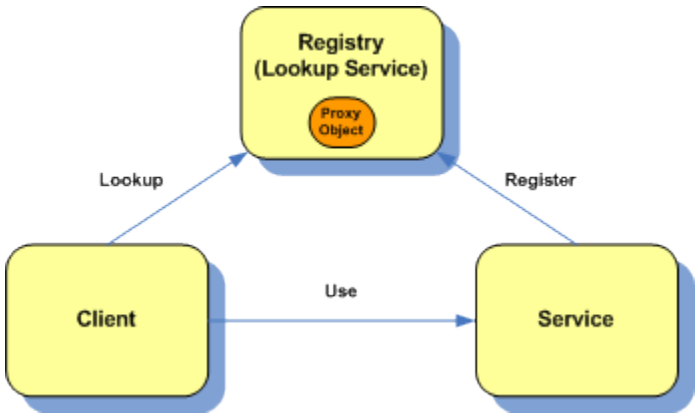


Figure 1: The relationship of the three main components of the Jini technology.

Of course Jini provides much more to build a robust distributed system, namely full security architecture, resource leasing, transactions and JavaSpaces that are all accessible via a mature and well designed API. Since Jini is used as the middleware in this distributed rendering software kit, the user will only perceive that after starting the client application the available services will be discovered automatically, and the addition and removal of computational services will be handled easily at runtime.

Architecture of the distributed rendering system

From the user's perspective using this distributed rendering facility is very easy and straightforward provided there is an already set up renderfarm. Just start the client application, wait a little until the required number of services are automatically discovered and then simply load your scene file and press a button that immediately starts rendering using the remote services. If you have to set up your own renderfarm then you will have a bit more work, but If you understand what to start and where then setting up your rendering environment is relatively easy. This section introduces the main components of this distributed rendering environment and they relationships.

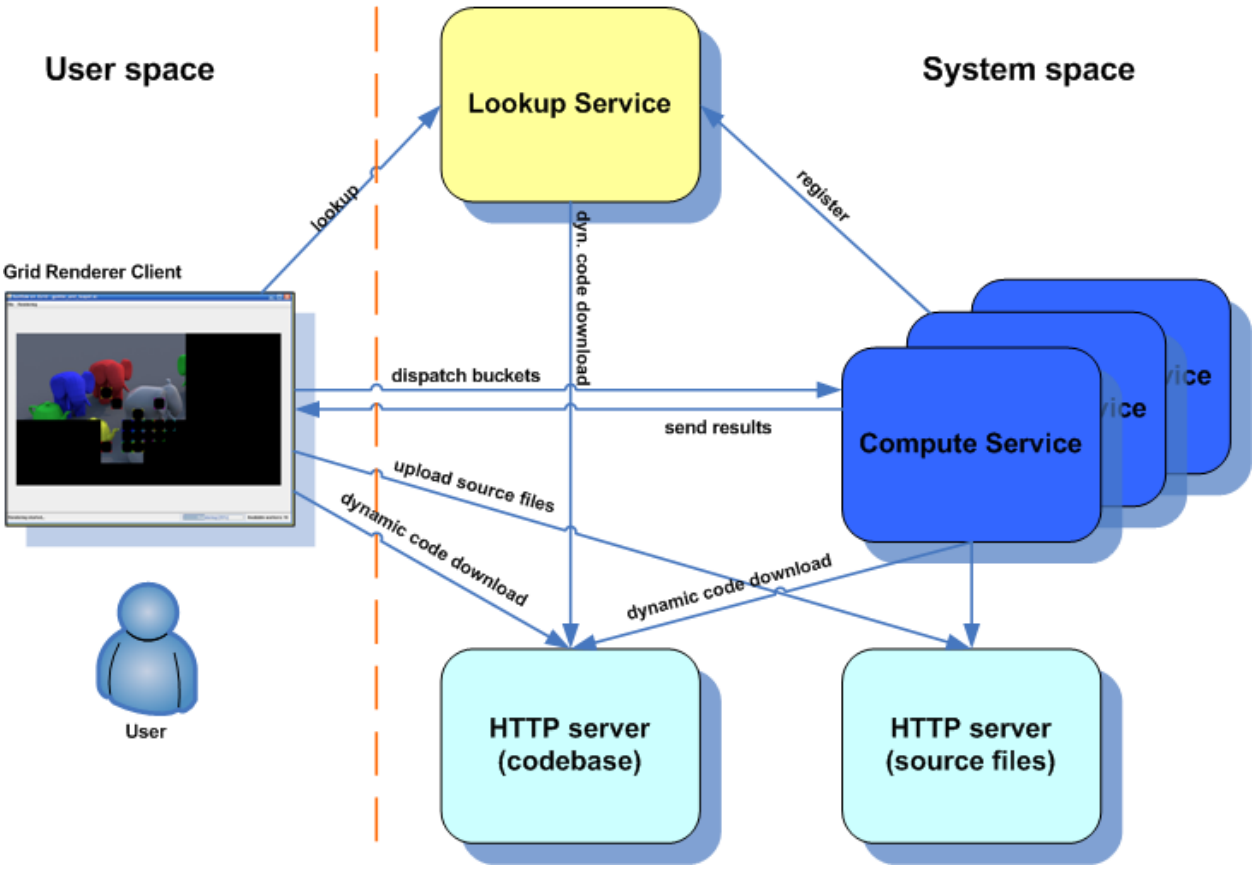


Figure 2: The architecture of the distributed rendering environment.

If you look at figure 2, you can recognize the three main components introduced in the previous section. The client in this case is the Distributed Rendering Client which has a GUI for easier use, the service is a special Jini service that is part of the JGrid infrastructure, called Compute Service, and finally the Lookup Service which is the registry so that the communicating parties could find each other. There are two more components in the figure, the two HTTP servers, one for the dynamic code download (required for RMI), and one for downloading scene and other necessary files by Compute Services for remote computation.

The actual computations required for rendering a scene will be carried out by Compute Services. These are Java programs that exports out a JVM as a computational resource and behave as Jini services so that clients could dynamically discover them and communicate with them via the network. The Compute Service is a general purpose Java task execution service, and was designed to be a platform independent compute node in the JGrid service-oriented grid infrastructure. It can execute both sequential and parallel tasks both in synchronous and asynchronous mode, it can be remotely administered and monitored, supports secure execution, and provides a high level API to program distributed applications building on it. For more details read the Compute Service Manual or the related research papers.

Typical rendering scenario

After introducing the main components a typical rendering scenario looks as follows:

1. A Lookup Service and the HTTP servers are started on a machine.
2. A Compute Service is started on each machine that is to carry out computations. After a successful start up services register themselves with the Lookup Service.
3. The distributed rendering client is started that first discovers the Lookup Service in the vicinity by multicast messages (or with unicast if multicast does not work).
4. The client looks up all the Compute Services available in the Lookup Service and downloads their proxy objects.
5. Via the proxy objects the client spawns as many renderer tasks (called workers from now) to each Compute Service as many processors they have (processor number is specified in the service attributes). With this the client is ready for rendering.
6. The user opens a scene file, and optionally adds required files, such as textures, bump maps, images etc., in the GUI.
7. When the user starts the rendering the client application distributes the name of the scene file, and all other required files to the workers waiting inside the Compute Services.
8. The workers first download all necessary files via the HTTP server or use a locale copy if available, build the scene objects and start the rendering bucket by bucket.
9. The renderable buckets are distributed by the client in a master-worker style. Each worker asks for a new bucket, renders it, sends back the rendered image and asks for the next bucket.
10. During rendering the user can continously follow the progress via the client GUI and can cancel the rendering session whenever he wishes.
11. If the user did not cancel the rendering then it finishes when there are no more unrendered buckets left. The workers return into a waiting state, the user can start a new rendering session.

Configurations

When you would like to set up your distributed rendering environment first you have to design it depending on the resources that are on your disposal. Here design means to decide which component will run on which machine. A minimal configuration is shown in figure 3. Here the Lookup Service, the HTTP servers and the client GUI runs on the user's machine, while the computations are made by another machine running one Compute Service. A real application of this configuration can be when the user wants to relieve his working machine meanwhile rendering a large scene, so that he could work uncluttered. Or, for example, if there is a powerful – possibly multiprocessor – machine available that the user has access to. The reason, why the Lookup Service and the HTTP server is not running on the remote machine, is that it will consume near 100% CPU time that could hamper the operation of the other services.

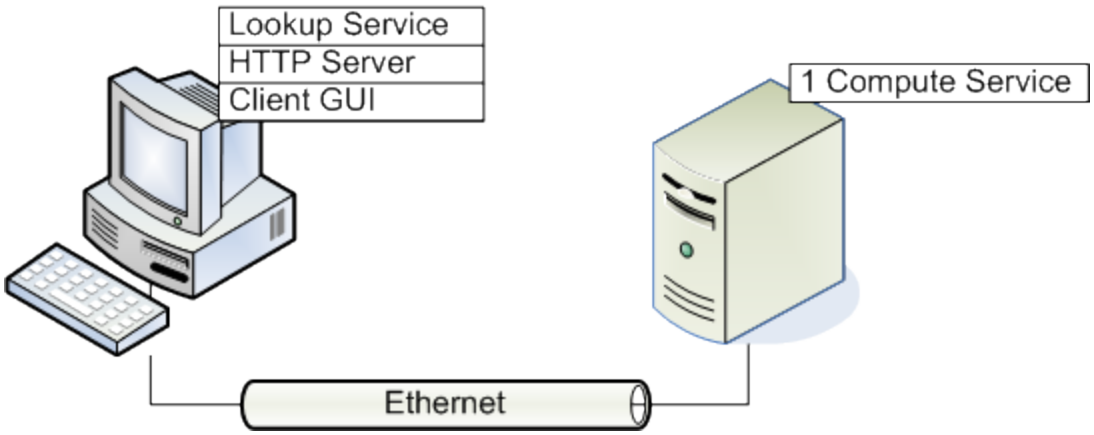


Figure 3: A minimal configuration of the rendering environment.

On the next figure (figure 4) a typical configuration is shown. This assumes that there are more computers available for rendering. In this configuration, the user machine runs only the client application because now the Lookup Service and the HTTP server (or servers) are migrated to a separate machine. This way the Lookup Service can keep running when the user machine is off that gives more stability to the system and lowers network traffic on the client side. The rest of the machines (that can be involved in the computations) each runs a Compute Service. Besides the user's and Lookup Service's machines any computer in the vicinity can be involved in the rendering. This can range from the workstations of the workmates that stay mostly idle in the network to the more powerful servers or mainframes.

However, there is one more important thing to mention. Since the scene files (.sc files) and they related files can possible be several 100 megabytes large it is necessary to ensure that the HTTP server has enough bandwidth and also scalable enough to serve dozens of Compute Services. If you work with large files, it is also possible to use a single shared directory on some file server, and configure the Compute Services to use that directory as the temporary download directory, and then mark all files in the GUI (or at least the large ones) as cachable, so no download will be made. In this case the client should also use that shared directory to select files from.

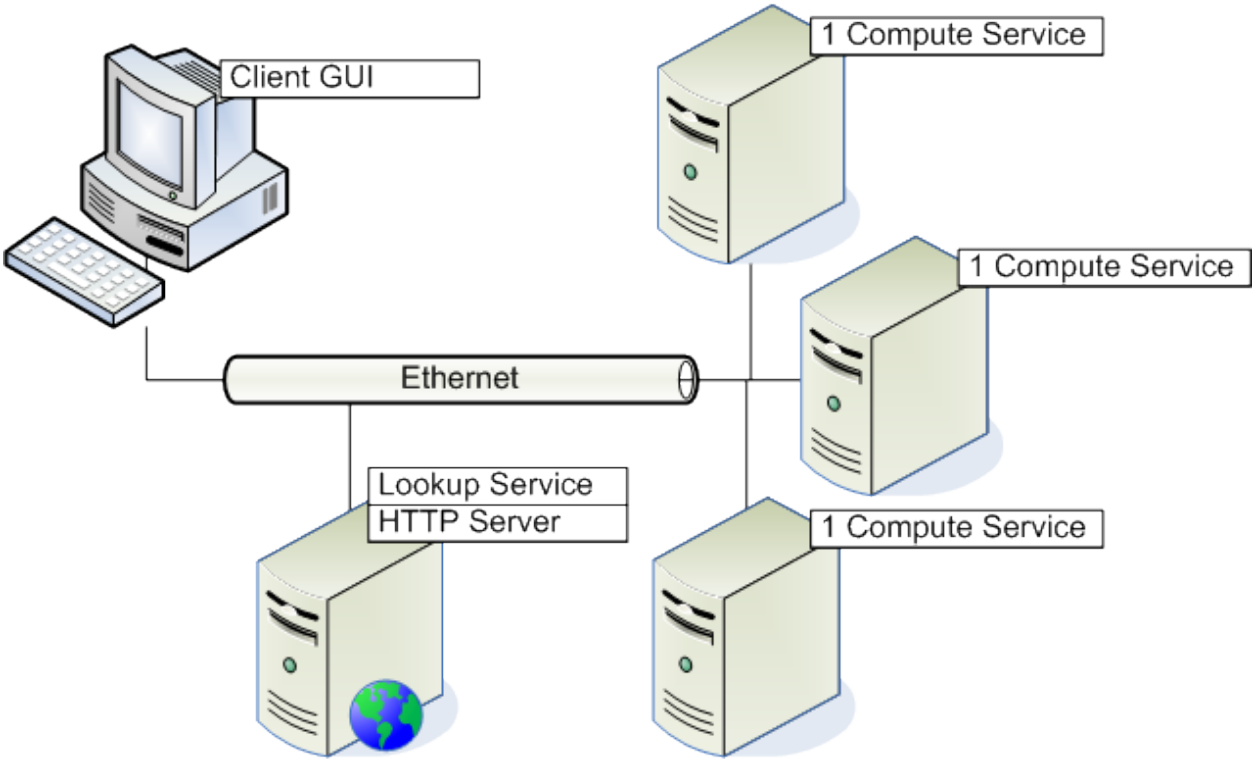


Figure 4: A typical configuration of the rendering environment.

After understanding and designing your system you can start installing the downloadable bundle to each computer.

Install & Config

Prerequisites

Before installing this software kit you have to have Java SE 6 or later installed on any computer you wish to use. You must have a JDK installed, because the service JVMs run in server mode that are only available in the JDK bundle. You can get the latest release of Java SE from the official Java web site.

This page uses the terminology and concepts introduced in the introduction, if you did not read it yet than I suggest to start with it first.

Install

To install the distributed rendering software kit:

- First download the latest release and unzip it to an arbitrary directory (RENDER_HOME).
- Next copy the RENDER_HOME/lib-ext/jsk-policy.jar file to the JAVA_HOME/jre/lib/ext directory.
- Then go to the RENDER_HOME/bin directory and open the setenv.bat or setenv.sh file (depending on your OS) to edit. The setenv file contains many environment variables that all

scripts and configuration files refer to, so it is important to set them properly. The next section describes it in more detail.

Configuring the environment

The RENDER_HOME/bin/setenv.[bat|sh] file contains all the environment variables which the user can configure the system behavior with. Note that here only those environment variables are mentioned that have to be set by all mean to start any of the components. For a detailed description of configurable variables refer to the comments in the setenv file.

- The JAVA_HOME environment variable is used to set the Java location. This must point to your JDK installation. In the setenv.bat, be sure to out the REM before the set JAVA_HOME= statement. In setenv.bat, when you set the JAVA_HOME, make sure to remove the quotation marks even if there are spaces in the path. Also, make sure both the JAVA and JAVAW have quotation marks around the full path for example: JAVA="%JAVA_HOME%\bin\java.exe".
- The next variable you have to specify is JGRID_LOCALHOST which is the IP address of the current machine. All entities (compute services, client, class server etc.) running on the current machine use this IP address to start server sockets. This IP address should be reachable for anybody in the execution environment. Use inner IP address only in case when all entities use inner addresses. To avoid problems do not use localhost or 127.0.0.1 even if you start everything on one single machine.
- Files describing the 3D model (.sc files) and all depended files (like textures, bump maps, images for image base lighting etc.) are downloaded by the Compute Services via HTTP from the client. By default this HTTP server is set to be the one that is included in this bundle and used for serving classes for mobile code. However, for large files, this simple HTTP server is not adequate thus it is suggested to use a scalable HTTP server like Apache or Tomcat instead. You can configure an external HTTP server via the FILE_DOWNLOAD_ROOT_URL and FILE_DOWNLOAD_ROOT_DIR environment variables. The first must specify the root URL of your HTTP server, while the second must point to the root directory of the HTTP server in the file system. This latter is required because the grid renderer client GUI must have a file view to your files so that you can choose them in a file chooser dialog. On the other hand Compute Services require a remotely accessible URL to download the files that the user selected.For example if you have an HTTP server accessible via the URL http://foo.bar.com that serves files from the c:/documents/web directory than you have to set the variables as follows:

  set FILE_DOWNLOAD_ROOT_URL=http://foo.bar.com/

  set FILE_DOWNLOAD_ROOT_DIR=c:/documents/web

  Note that the value of the FILE_DOWNLOAD_ROOT_URL variable must end with a "/" slash. The path of the used files will be always treated relative to these settings.

Now you are ready to start the required components like compute services or the rendering client.

Running the components

In the downloadable bundle the bin subdirectory contains all the scripts that are to start the different components. In the following listing each script is described in detail, they appear in the order as you should run them on the host computer (the extensions are .bat or .sh depending on your OS). For slave computers you only need to run the cs script:

1. classserver: this script starts a simple HTTP server (found in the Jini bundle) on the localhost. This simple HTTP server serves the JAR files required for the mobile code. You should always start this first, as it is required by all other components running on the same host. You can configure the port on which this server starts in the setenv file by the CLASSSERVER_PORT environment variable.
2. lookup: this starts a Lookup Service on the current host. This is the registry for the Compute Services and the renderer client so it should be started before them. Of course it is not a problem if you start it later, but until a Lookup Service is not running no discovery can be made, and the renderer client will not find any Compute Services.
3. cs: this script starts a Compute Service on the current host. If there is Lookup Service running in the multicast range then the Compute Service will discover it and registers itself so that clients can immediately discover and use it. You should see the “Service is ready.” text in the command line if the Compute Service started successfully.
4. browser: starts the JGrid service browser. This browser can discover the Jini services in the multicast range, or the ones found in the specified Lookup Services. The browser helps you to visualize the available services and view their attributes. If you started some services but you cannot see them in the browser than there is some problem in the discovery process (see the problem solving section below for more details). If you see your services in the browser than the renderer client can also discover them, so you can proceed.
5. client: starts the grid renderer client GUI. If you have already set up your environment then the client will automatically discover the available Compute Service in the multicast range and you can start rendering your scenes. See the user manual for more details about how to use the client.

For more details about the set up of your rendering environment please read the introduction document.

Problem in discovering the services

The most usual problem when working with Jini is that the automatic discovery of service is not working. You start the browser and cannot see any services. Here are some possible reasons why the discovery does not function:

- Check if you started the class server. If you did, and you have services running then in the command window (or prompt) you should see HTTP requests like : “compute-wrapper-dl.jar requested from IP_ADDRESS“. If you cannot see any requests then check if the JGRID_LOCALHOST variable was set to the correct IP address of the local machine, or if the port of the class server is not blocked by any firewall.
- The next reason is often that the multicast is not enabled on your network, or there are some services that are beyond the multicast range. In this case you should use unicast discovery and must specify the direct Jini URL of the Lookup Service to every entity that wish to discover other services. For example if you started a Lookup Service on a machine that has IP_ADDRESS then the Jini URL is: jini://IP_ADDRESS:4160 or just jini://IP_ADDRESS since the port 4160 is the default one (this can be changed in the lookup/config/transient-reggie.config Lookup Service configuration file). Instead of the IP_ADDRESS you can also specify the hostname of the machine like: jini://foo.bar.com:4160. To test unicast discovery the easiest way is to start the JGrid service browser and in the Locator field beneath the toolbar type the Jini URL and press the Go button. Then switch to the Registrars tab on the left side where your Lookup Service should appear. If it works than you should configure the Compute Services and the client to use this unicast address instead of relying on multicast discovery.To configure the Compute Service edit the computeservice/config/jeri/transient-compute.config file. Look for the net.jini.lookup.JoinManager block at the end of the file and within it the discoveryManager entry. Uncomment the line: “new LookupLocator(“jini://your_host:4160)” and replace “lookup_host” with the IP address or hostname of the machine which the Lookup Service runs on.

  To configure the renderer client you have to do almost the same thing but in the client/config/client.config file. In this case you have to look for the net.jini.lookup.ServiceDiscoveryManager block at the beginning of the file and within it the discoveryManager entry. Uncomment the new LookupLocator(… line again and replace the “lookup_host” with the IP address or hostname of the machine which the Lookup Service runs on.

  From now the Compute Services and the client will look for a Lookup Service specified by the Jini URL.

- An other reason for the discovery not working if you forgot to copy the RENDER_HOME/lib-ext/jsk-policy.jar file to the JAVA_HOME/jre/lib/ext directory, or you copied it to the extension directory of an other JDK than you are running the applications with.

User Manual

After you installed and configured your distributed rendering environment you can start rendering your Sunflow scenes via the distributed rendering client (DRC). This client displays the rendered images with Sunflow’s built in image display so it could be familiar from Sunflow’s GUI application.

This page uses the terminology and concepts introduced in the introduction, if you did not read it yet than I suggest to start with it first.

The status bar

If you would like to render any scene the most important is to have at least one Compute Service available. You can check the number of discovered services via the status bar at the bottom of the GUI, or to be more precise it displays the number of worker tasks spawned to the discovered services, which is equal to the total processor number available. The status bar is an important part of the GUI during rendering too, since you can follow the rendering progress here.

From left to right the status bar contains the rendering status messages, the rendering progress bar and the display of available workers.

Rendering a simple scene

Under simple I mean that the Sunflow .sc file contains no dependent files, like other .sc files, images, textures and so on. To open the desired scene file select File->Open from the menu or press Ctrl+O. A file chooser dialog opens and the current directory is set to the root directory of the HTTP server that you configured in the setenv file (see the install document for more details). You can select files only in this directory or subdirectories, otherwise you will get an error message.

After you selected the file a small message dialog is displayed that asks whether the scene file can be cached or not. If you select YES then the Compute Services will use already downloaded local copies if available, otherwise download the file. If you select NO, the scene file will be downloaded before each rendering session. If you wish to render the same scene many times to experiment with different rendering settings then you should force download each time.

If the file opened successfully you will get a message, just click OK and you can start rendering. Note that that the client application itself will not parse the file and will not build the scene objects. It only looks for the image dimensions (the image {…} block in the .sc file) if it is found that the open is successful.

To render the scene with Sunflow's Simple renderer select Rendering->Simple from the menu or press Ctrl+Z. As its name says, this render is faster but produces only a preview quality image, if you are not sure about your settings select this one first.

To render a scene with Sunflow's bucket renderer, which produces the full quality image, select Rendering->Bucket from the menu or press Ctrl+X. This type of rendering could take considerably more time to finish.

During the rendering session you can follow the progress in the status bar as it is seen in figure 2, and also on the display as the rendered buckets are displayed on by one. The colored frame of yet unrendered buckets means that it was already dispatched to a remote service but the result did not arrive yet. When the rendering is finished a small message dialog displays the total rendering time, but you can also see it in the status bar.

You can cancel the current rendering session by selecting Rendering->Stop from the menu or pressing Ctrl+C.

During the rendering It is important to take into considerations that each rendering session involves a great number of network communication. If you render a large file, or it takes a long time for Sunflow API to build the scene objects then you can perceive a large delay between the start of the rendering and the return of the first rendered bucket. In the Window->Log Panel menu (or Ctrl+L) you can pop up the log panel, where you can get more detailed messages about the current progress of the rendering or possible failures.

For a better performance if you have large scene files then separate them into a smaller one containing only the always changing settings, such as image parameters, shaders, lighting, camera etc. and into a bigger one containing the geometry that changes very rarely. This way the larger file can be marked as cacheable so after the first rendering only the small file will be downloaded.

Rendering scenes with external files

If you have external files that the main scene file refers to, such as textures, bump maps, images, geometry data, object files etc., then in the current version you have to add them manually before the rendering session. Open the main scene file that should always contain the image {…} block and that is the one that is parsed first at the remote site selecting File->Open from the menu. To add external files that are required during the rendering select File->Add required files in the menu or press Ctrl+R. A dialog opens where you can manage the external files. Here you can add and remove files and you can also mark files as cacheable or not. Cacheable files will be downloaded only once.

After selecting the required files you can start rendering as it was described in the previous section.

Rendering animations

Animation is a series of scene files rendered one by one. First open the scene file of the firs frame in the File->Open menu then add the rest of the scene files selecting File->Add animation files from the menu or pressing Ctrl+A. A dialog opens that is very similar to the one where you added external files. Select the scene files of the frames, Toggle caching whether you want the files to download once or not. If the scene files require some external files then select File->Add required files.

Before starting the rendering switch to animation mode by selecting Rendering->Animation, otherwise only the first frame (the main scene file) is rendered. Then you can start rendering in the same way as it was described in the previous sections. The rendered frames will be automatically saved in the client/frames directory as frameNNN.png where NNN is the frame number. (In future versions this will be configurable).

Setting the number of threads/cpus

Currently there are no automatic recognition of CPUs or CPU core numbers but you can set it manually. You have to edit the Compute Service's discovery attributes in the "computeservice\config\jeri\transient-compute.config" file. By going to the following section:

```
// Edit this attributes to reflect real values
initialAttributes = new Entry[]{
    ...
    new Processor(
        "Enter your processortype here",
         new Integer(1),
         new Integer(2400),
         new Integer(2400)
    )
};
```

The parameters of the Processor attribute are as follows:

- processor type – String
- processor number – Integer
- processor performance (MFlops) – Integer
- processor speed (MHz) – Integer

Currently only the second parameter, the processor number, is taken into consideration by the rendering client. This is where you have to set to the number of CPUs or cores. For example, you would set it to “new Integer(4)” if you have a four core CPU.

If you restart the service you’ll see that the number of available workers will be the number that you’ve set. You can test the gained performance by benchmarking the Compute Service with the new settings.

Before closing the client application

When you decide to close the client application, before it you can save the rendered image under the File->Save image menu or pressing Ctrl+S.

When closing the application either via File->Exit menu, Ctrl+E shortcut or on the window, the client will cancel all tasks spawned to the Compute Service.

Limitations and known issues

- If you forget to add external files to a scene that requires it that the rendering can hang, because the Sunflow API in remote services blocks from some reason if the required file is not found. When this happenes cancel the rendering and add the required files.
- Currently Java scene files cannot be rendered because Janinio requires so many permissions that are not allowed for tasks running in the Compute Services. Or you can try to configure the Compute Services to run with a policy file containg all permissions, but I did not tested it yet.
- This software kit uses a slightly modified version of Sunflow so do not overwrite the sunflow and janino JAR files with newer ones. Sunflow is compiled without the Class-Path set and should be added to the client/lib and lib-dl folder. Once compiled, you have to rebuild the client with Ant (build.xml in the client directory) because this will regenerate the lib-dl/client- wrapped-dl.jar file that contains MD5 hashes of the client-dl.jar, sunflow.jar and janino.jar files and also

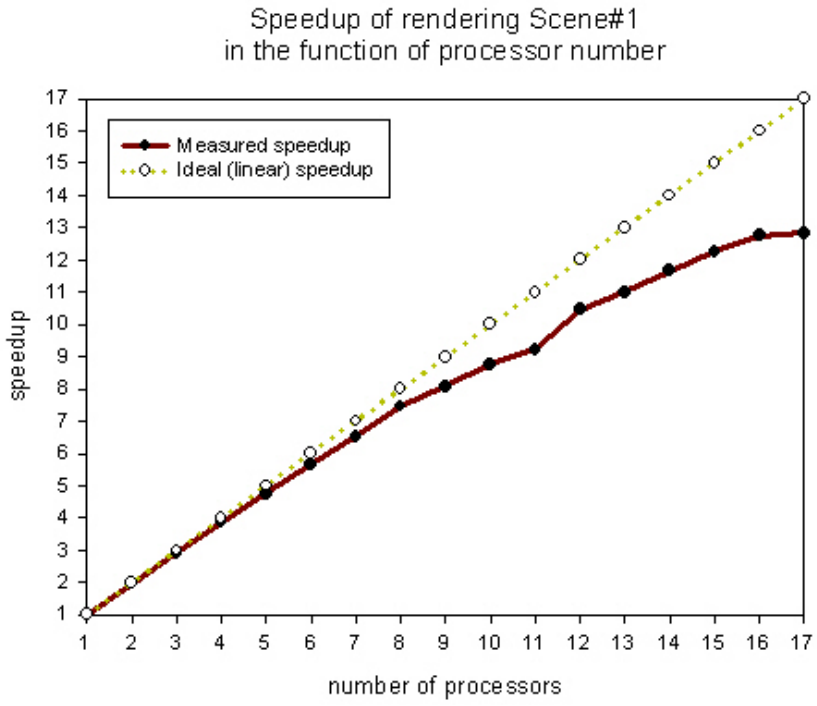
replaces the MD5 hashes in the starter scripts in the bin directory. Modifications to Sunflow source can be found in this diff against the Sunflow source.

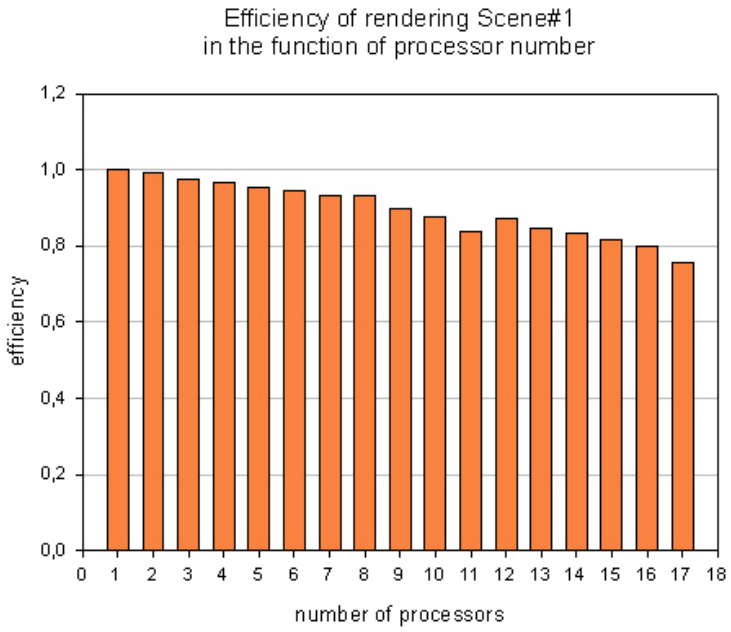
Performance

Results of speedup and efficiency measurements:

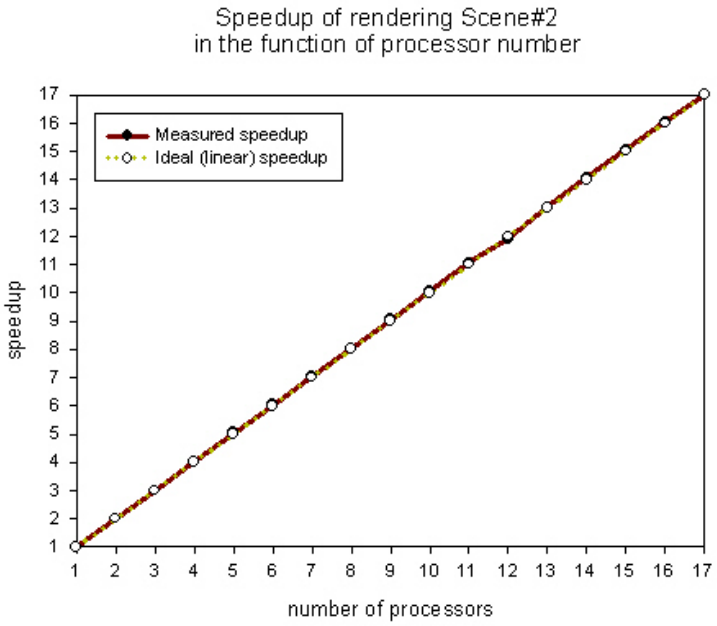
- Measurements were done using 17 PCs with identical hardware setup and connected with 100Mbit/s Ethernet.
- MFlops values were computed using the Java version of SciMar2 benchmark with the option "large"

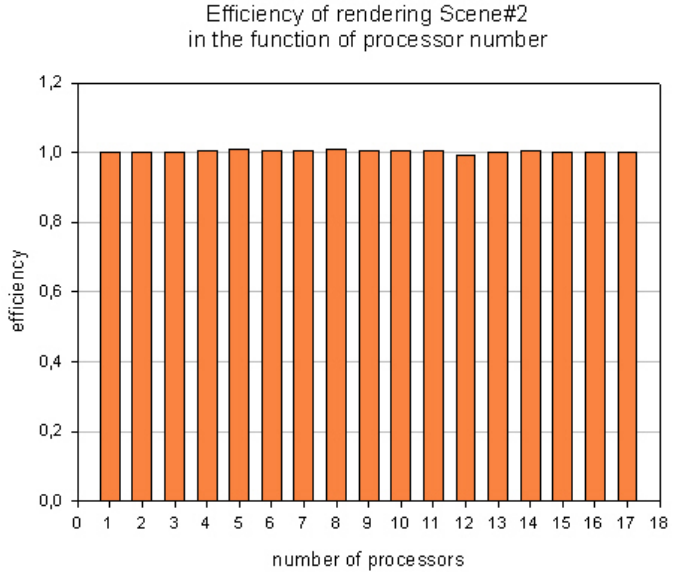
Scene #1: Toyota
Resolution: 1024×768
Number of processors: 1 – 17
MFlops/processor: 198,7
Total MFlops: 3379,0
Render time on 1 processor: 35 min 56 sec
Render time on 17 processor: 2 min 48 sec





Scene #2: Gobelets
(image by Tartiflette)
Resolution:1520×1012
Number of processors: 1 – 17
MFlops/processor: 198,7
Total MFlops: 3379,0
Render time on 1 processor: 15 hours 17 min
Render time on 17 processor: 54 min


Speedup of rendering Scene#2
in the function of processor number
Measured speedup
Ideal (linear) speedup
speedup
number of processors
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17


Efficiency of rendering Scene#2
in the function of processor number
efficiency
number of processors
0,0
0,2
0,4
0,6
0,8
1,0
1,2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18