

# GPU, Framework Extensions for the Distributed Search Engine and the Terragen Landscape Generator

Rene Tegel and Tiziano Mengotti  
Technical report  
GPU Development Team

August 20, 2005

DRAFT

### Abstract

This document describes new features introduced in the GPU network, a framework for distributed computing based on the Peer-to-Peer network Gnutella, and soon on a new protocol called Pastella.

Frontends and plugins are now able to communicate with the main GPU application using UDP/IP channels. Frontends and plugins can now listen for broadcasts and get packets even if they are not explicitly intended for them. Frontends can also provide a webinterface for the submission of queries by users who are not running the GPU application. Plugins accept strings as parameters and send back partial results, while they are computing. Most features were introduced to support the Distributed Search Engine, a plugin/frontend extension of the network to index webpages using crawlers.

The framework is now able to generate artificial landscape videos using Terragen [8], a photo-realistic landscape generator. A new application launcher plugin and a wrapper to download and upload the frames back on a central server are described. These frames are then merged together into videos, which become photo-realistic fly-byes of the generated landscape.

Finally, we discuss new ideas turning the broadcast-based framework into an agent-based infrastructure.

## 1 Introduction

This document is intended for developers, who would like to experiment with the advanced features of the GPU<sup>1</sup> network. Newcomers first should take a look at [3,4,5] for an introduction to the GPU framework. The framework extensions described here were made possible by the VisualSynapse [1] and Synapse [2] libraries. These libraries provide Delphi components for UDP, TCP, HTTP and several other protocols.

Chapter one explains how frontends<sup>2</sup> capabilities were improved since the publication of [5]. They communicate now with the main GPU application using UDP/IP channels, in addition to *message passing* and *shared memory* [5,6]. Communication through UDP is more straightforward. It will easier cross platform development and will enable other programming languages to speak with GPU. Additionally, plugins are allowed to use the UDP/IP channel to submit own queries and receive results, too.

Frontends and plugins can register for network broadcasts. Doing so, they can get messages intended for the entire network. The Network mapper uses this feature to improve its efficiency. Chatbot plugins can listen for messages on chat using the same mechanism.

Chapter two describes improvements made to plugins. Plugins are now able to accept strings on the stack and are not limited to real numbers anymore. An application launcher implemented as plugin enables GPU to execute batch jobs. As an example, we provide wrappers for the Terragen plugin, a landscape generator that can run on a GPU cluster.

<sup>1</sup>Global Processing Unit or Gnutella Processing Unit

<sup>2</sup>A frontend is an application that interfaces to the GPU main application. It is intended to visualize results and submit new jobs.

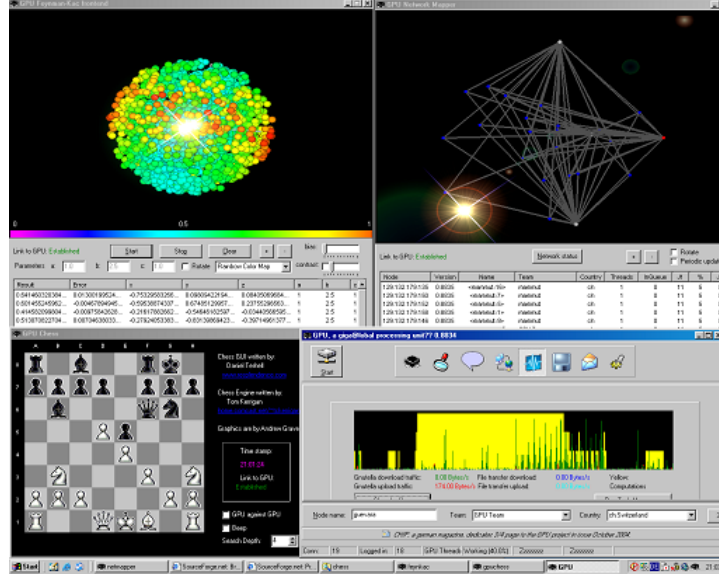


Figure 1: *Some minor changes introduced into the framework* - the Feynman-Kac frontend supports colors, Netmapper displays up to 40 connected computers. The chess system still does not work in parallel, unfortunately.

Plugins can send back multiple results while they are computing as well. The Distributed Search Engine described in chapter three extensively uses these feature to send back partial results, while it scans the database for web pages matching the query.

Chapter four gives insights in how to use the color maps to plot points, describes how to build local clusters and describes in detail the autoupdate system.

Chapter five discusses new ideas on getting a much more efficient transport protocol called **TPastella** [1] (based on *Distributed Hash Tables*[reference here]). Additionally, we show an agent based extension for the framework, which might overcome some inherent limitations of GPU in respect to job distribution.

In the conclusion, we give statistics for the cluster and discuss directions for future work.

## 2 Frontend improvements

### 2.1 Frontend Communication through UDP/IP

We describe here, how a frontend can ask GPU to distribute jobs and then listen for results<sup>3</sup> using UDP/IP packets. These connectionless channels allow programmers of different languages and platforms to write frontends and understand how communication with the main GPU application. In fact, UDP/IP is a very common protocol and most languages provide libraries to use it. For

<sup>3</sup>One way to do so using *windows messages* is described in [4]

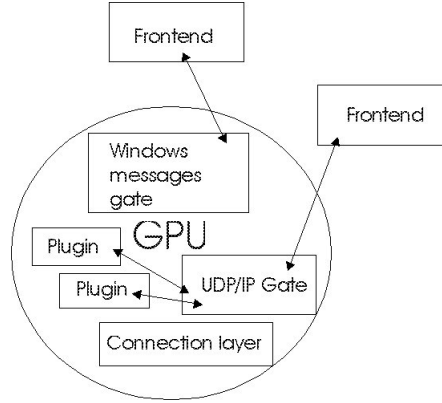


Figure 2: *UDP/IP Gate* - Besides the Windows message gate, a new UDP/IP gate is introduced in GPU to allow communication with the core client.

example, implementing a UDP/IP channel in Java should not be more than 10 lines of code.

### 2.1.1 General proceeding

The main GPU application listens on port 27077 for incoming UDP packets. This port number is defined as constant `GPU_UDP_PORT` in `common.pas`. The frontend listens on some other port, defined by the frontend's developer.

UDP is a protocol built on top of IP and therefore it contains a source and destination address. The address is a couple (IP/Port). To communicate with GPU, the frontend creates a packet with (`localhost/27077`) as destination address and (`localhost/frontend port`) as source address. The packet contains the GPU command in a simple format described later. The packet is sent to the main application. The main application spreads the packet through the network, and connected computers will start returning results to the main application as usual.

When results come in, the main application builds UDP/IP packets directed to (`localhost/frontend port`). In this way, results reach the frontend. The results are then edited or visualized by the frontend as final step.

### 2.1.2 Format for request and result

The format for a request is very simple. It is a string with three fields separated by a colon.

The first field is normally set to 0. GPU spreads the job through the network, then. If the first field is 1, the job is only executed on the local machine. The second field is the job ID number of the packet; the third field is the GPU command itself.

Format: `Is Local:Job ID:GPU command`

Example: `0:1500:1,1,add`

Results sent back by GPU do not have the `Is Local` field.

Format: `Job ID:GPU Command`

Example: `1500:2`

### 2.1.3 Frontend Communication with Delphi and Visual Synapse component

In the following, we limit ourself to the Delphi [7] environment. Additionally, we assume that the Visual Synapse [1,2] component is already installed into the Delphi environment. Please refer to [1] for information on how to install the Visual Synapse component.

A frontend listens on a developer's defined port, for example port 27078 as `simple_udp.pas` in directory `frontend/simple-udp` does. To achieve this, we drop the VisualSynapse [1] component called `TVisualUDP` on a new Delphi form.

In the Object Inspector window, once the new `TVisualUDP` component is selected, we set the property `BindAdapter` to `127.0.0.1`, the IP address for `localhost` that represents the local computer. The `BindPort` property is set to the listen port of the frontend, freely chosen by the developer, `27078` in our case. Finally, we give a distinct name to the component by setting the `Name` field to `SynapseUDP`.

### 2.1.4 Receiving results from GPU

The routine that handles incoming results sent by GPU is the event of `TVisualUDP` called `OnData`. It is enough to set this event, no more events should be defined. Here a simple example on how the routine for `OnData` could look like.

```
procedure TSimpleForm.SynapseUDPData(Sender: TVisualSynapse; \\  
  Handle: Integer; Data, Query: String; From: THostInfo); \\  
  var JobID : String;  
  
begin  
  JobID := ExtractParam(Data,':');  
  GPUResultEdit.Text:= Data;  
end;
```

The `Data` parameter contains the entire string in the way GPU sends it, defined in the previous section. The method `ExtractParam(...)` defined in `utils.pas` writes in the variable `JobID` the first part of the string up to the delimiter `:`. `Data` contains the rest of the string without delimiter. The result is then showed in a `TEdit` component called `GPUResultEdit`.

### 2.1.5 Sending a command to GPU

To send a command to GPU, the following method is proposed:

```
uses common.pas  
...  
procedure TSimpleForm.SendUDPCommandToGPU(JobID, Command : String);
```

```

var S : String
begin
    S := '0'+':'+ // 0 will spread command to all
                // connected GPUs,
                // 1 only to this machine
    JobID+':'+Command;

    SynapseUDP.Connect('localhost',GPU_UDP_PORT);
    SynapseUDP.SendTo('localhost', GPU_UDP_PORT, S);
end;

```

The two routines `Connect(...)` and `SendTo(...)` create and send a UDP packet to GPU containing the computational command.

To send a simple command, one can call for example:

```
SendUDPCCommandToGPU('100','1,1,add');
```

## 2.2 Registering a frontend to network broadcasts

Currently<sup>4</sup>, there are two broadcast services offered by the network. For these two services, the broadcasting is done by the GPU core client itself. Plugins can offer broadcast services using partial results. We will describe this technique later in this document.

One service is a string containing some status information of participating computers. Each two minutes, the GPU core client broadcasts its status information, including communication status (IP, connections, traffic load), virtual machine status (jobs, percentage of jobs, number of jobs in queue) and physical status (MHz, RAM, result of speed benchmark using FFT routines, collected data of crawlers in megabytes). The job id of this service is set in constant `NETMAPSTATS_BROADCAST` in `common.pas`. The frontend that listens to these messages is the 3D Network mapper in directory `frontend/netmapper`. This frontend has to register for the job ID `NETMAPSTATS_BROADCAST`.

To listen for the job ID 100, one types

```
SendUDPCCommandToGPU('100','register');
```

using the routine previously defined. The same approach works with *window messages* as well. If a frontend closes, it should unregister itself with

```
SendUDPCCommandToGPU('100','unregister');
```

Note that on these two commands, the `Is Local` field is always forced to 1. In fact, both the `register` and `unregister` command are never sent to the entire network, for obvious reasons.

## 2.3 Plugins can use the same UDP/IP channel to submit and receive results

Plugins can use the UDP/IP channel to submit queries and receive results, as well. A good plugin example can be found under `dllbuilding/chatbot_echo`.

---

<sup>4</sup>in version 0.910

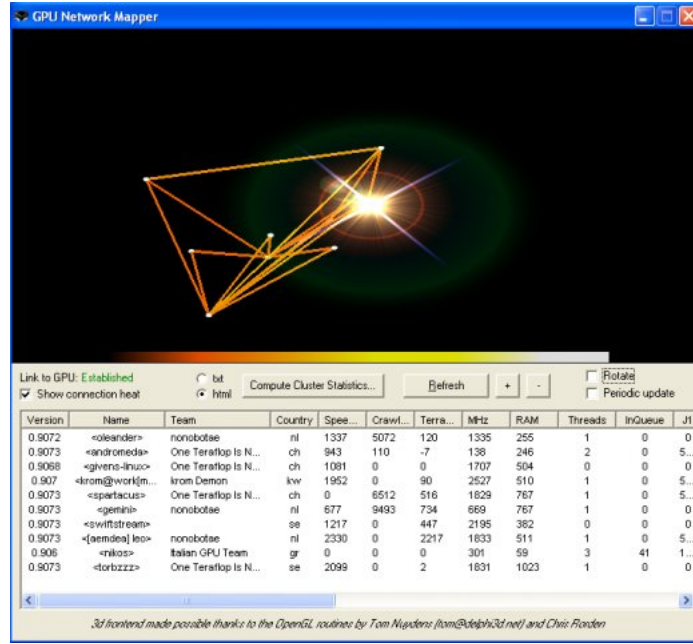


Figure 3: Netmapper frontend listens for Network Broadcasts. Connection load is displayed with a heating color map.

The plugin registers for the service `CHATBOT_BROADCAST`, so that it is notified of chat entries on chat channels of the GPU client.

The example uses the non-visual version of Synapse to submit queries. The defined UDP socket sends data to the `GPU_UDP_PORT` and is bound to another listening port. A thread is instantiated, each time it gets data through the listening port, it resubmits it to the GPU UDP port.

Therefore, the `chatbot_echo` plugin acts like a parrot which repeats sentences on chat channels. Activating two parrots on the network, and sending an initial sentence can produce an interesting avalanche effect.

The following source code shows how to use `SynapseUDP` without dropping visual components on a form (DLL's do not have such a form).

```
var
  thread : TUDPEchoThread;

...

thread := TUDPEchoThread.Create(false);

constructor TUDPEchoThread.Create(CreateSuspended : Boolean);
begin
  ...
  udpcon := TUDPBlockSocket.Create();
```

```

udpcon.Bind('127.0.0.1','31255');
end;

procedure TUDPEchoThread.Execute;
var res,
    channel : String;
begin
    sendmsgtogpu(CHATBOT_BROADCAST,'register');

    while not stop do
        begin
            res := udpcon.RecvPacket(1000);
            ...
            sendmsgtochannel(res);
        end;

        sendmsgtogpu(CHATBOT_BROADCAST,'unregister');
    end;

    procedure TUDPEchoThread.sendmsgtogpu(jobid, msg : string);
    begin
        if udpcon = nil then Exit;
        udpcon.Connect ('127.0.0.1', GPU_UDP_PORT);
        udpcon.SendString ('0:'+jobid+':'+msg);
        //0 means sending message to all gpus
    end;

    procedure TUDPEchoThread.sendmsgtochannel(msg : String);
    begin
        sendmsgtogpu('1','chatbot:14:'+msg); //jobid does not matter here
    end;

```

`dllbuilding/aibot` is a more complex chatbot which uses markov chains to build complex sentences and is compiled into `gina.dll`. To test Gina, create a text file (say about 2 MB) containing books of your favourite writer or scientist. As an example, take some Einstein's books and create a text file `Einstein.brn`<sup>5</sup> With the GPU command `gina_enable`, you activate Gina and with 10, 'Einstein', `gina_brain` you'll send Gina through the books. After some minutes, Gina will have built a markov chain net of Einstein's book and will be ready to speak with anyone on chat channel 10.

## 2.4 Adding a webinterface to a frontend

Webinterfaces allow GPU to offer services to users who cannot run a GPU on their computer.

A frontend, like the Distributed Search Engine described later, can easily implement a `THTTPServer` provided by the VisualSynapse [1] libraries. This frontend becomes therefore a webserver listening on port 80.

---

<sup>5</sup>.`brn` extension = brain





Figure 4: *Webinterface of the Distributed Search Engine* - Users can use a webbrowser (here e.g. Konqueror) to access GPU services, even if they do not run a copy of GPU on their computer.

Users can browse the webserver with an usual Internet browser and interact with the frontend, although they do not run a GPU on their computer. To activate the webinterface of the Distributed Search Engine, run first **searchfrontend.exe**, go to panel *Web service configuration* and check *Enable webserver*. Now open a browser and point it to **http://127.0.0.1**.

#### 2.4.1 Using THTTPServer

This and the following subsections are taken from [1]. To get a webinterface inside any application (frontend or not), it is first necessary to instantiate the **THTTPServer**, give it a hostname, tell the port to listen to. Then one should map directories, PHP (or other preparers) and CGI directories. It is possible to map those against a specific host name<sup>6</sup> or to any hostname, by leaving the 'domain' empty. Each virtual host can have it's own set of CGI directories, preparers etc.

#### 2.4.2 Multiple virtual directories

It is possible to map several virtual directories, although care is needed if directory names overlap. As an example, assume this structure on disk:

```
c:\web
c:\web\downloads
c:\files
```

---

<sup>6</sup>Virtual Hosts

We would like to map the directories on the webserver as follows:

```
c:\web => /  
c:\files => /downloads
```

At this point, there is a virtual directory `downloads` that is mapped to two locations. As rule of thumb, the first mapping counts, and will be used to parse files. In this case, `downloads` will point to `c:/web/downloads`.

### 2.4.3 Recursive mapping

By default, folders are mapped recursively<sup>7</sup>.

### 2.4.4 Configuring the server

We give here a simple example, on how to instantiate a `THTTPServer`.

```
H := THTTPServer.Create (Self);  
H.ListenPort := '80';  
H.Active := True;  
H.SupportedProtocols := H.SupportedProtocols + [hpConnect, hpPost];  
H.AutomatedProtocols := H.AutomatedProtocols + [hpConnect, hpPost];  
H.RegisterCGI ('c:\www\scripts', '/scripts/');  
H.RegisterDir ('c:\www', '/');  
H.RegisterDefaultDoc ('index.htm');  
H.RegisterDefaultDoc ('index.php');  
H.RegisterPHP ('c:\bin\webtools\php\php.exe', '.php');  
H.RegisterManualURL ('/test');  
H.OnGet := OnGet;
```

This section is intended to give an overview on webinterfaces. Please check [1] for the latest documentation.

---

<sup>7</sup>Care on Unix based systems is required. There is no 'follow symlink' check yet, so the reader should avoid endless loops inside the directories

## 3 Plugin improvements

### 3.1 String as parameters for the virtual machine

Since version 0.860, the TStack record in `gpu_component/definitions.pas` is modified. It contains now an array of *PChars*<sup>8</sup> to keep compatibility with standard *Windows* dlls.

In consequence, plugins can take strings as parameters, push or pop them on the stack, and return them.

We described in [4], how the plugin `basic.dll` contained simple GPU commands like `mul` to multiply real numbers. Similarly, the plugin `strbasic.dll` contains simple routines intended for basic operations on strings. Its source code can be seen in the directory `dllbuilding/strbasic`.

#### 3.1.1 Examples from the user's perspective

We look here at the simple examples that come along with GPU. From these examples, the reader should be able to infer how the stack works.

The virtual machine understands strings only if enclosed by simple quotes (`'`). `Hello` is not a string for GPU, but `'Hello'` is.

- `'Hello','World',concat`

Command `concat` takes two parameters, concatenates them and returns `'HelloWorld'`.

- `'5','123456789',substr`

Command `substr` returns the position of the first parameter into the second parameter. In this example, the number 5 is returned. If the first parameter is not found in the second, 0 is returned.

- `'gpu is cool,8,4,copy'`

This command copies part of the first parameter, beginning from position 8 to a length of 4 chars. The result is `'cool'`.

- `'Do',' not',concat,' drink',concat`

The virtual machine evaluates this command from left to right. The first `concat` returns `'Do not'` on the stack. The result is understood as first parameter to the second `concat`. The outcome is then `'Do not drink'`.

Plugin `strbasic.dll` understands additional commands like `insert`, `delete` and `length`. Examples of these can be seen in the combo box near the *Compute locally* and *Compute globally* buttons.

The ability to handle strings is already used in the Distributed Search Engine and might be important to overcome the packet distribution problem described in the last chapter.

---

<sup>8</sup>PChars are C-style strings; they are a simple pointer to a null terminated string. In contrast, Delphi strings are an array of chars where the first element contains the length of the string.

### 3.1.2 Examples from the developer's perspective

As we know from [4,5], a function defined in a plugin takes a record `TStack` as argument. This argument is passed by reference: any change the function will do on the record will stay persistent: other functions might be called with this modified stack, or simply the modified stack is shown as result by GPU. The method signature for any command looks as follows:

```
function concat(var stk : TStack):Boolean;stdcall;
```

The `TStack` record contains three important fields: `StIdx` contains the position where the function should perform changes, `Stack` is an array containing `MAXSTACK` (constant defined in `gpu.component/definitions.pas`) real numbers and `PCharStack` is an array containing `MAXSTACK` pointers to strings. In consequence, `StIdx` should be in range 1 to `MAXSTACK` or zero if the stack is empty.

```
TStack = record
    StIdx      : LongInt; // Index on Stack where Operations take place
    Stack      : Array [1..MAXSTACK] of Extended;
    PCharStack : Array [1..MAXSTACK] of PChar;
end;
```

Adding a real number to the stack should be done as follows. First we check if the stack is not already full, we then add one to `StIdx`, we modify the `Stack` at the position `StIdx`. Additionally, we make sure `PCharStack` contains a nil pointer (should we also free memory here?).

```
function load_number_5(var stk : TStack):Boolean;stdcall;
begin
    Result := false;
    if stk.stIdx >= MAXSTACK then exit;
    stk.stIdx := stk.stIdx + 1;
    stk.Stack[stk.stIdx] := 5;
    if stk.PCharStack[stk.stIdx] <> nil then FreeMem(stk.PCharStack[stk.stIdx]);
    Result := true;
end;
```

To load a string on the stack, we check again if the stack is not already full, we reserve memory with `getmem(...)` at the address stored into the `PCharStack` pointer in the array at position `stk.stIdx`. We copy the normal Delphi string `S` into the freshly reserved memory with a call to `StrPCopy(...)`. Finally, we delete any number in the `Stack` array by setting its value to infinity (`INF`).

```
inc (stk.stIdx);
if stk.stIdx >= MAXSTACK then Rexit;
getmem (Stk.PCharStack [stk.stIdx], length(S)+1);
StrPCopy (Stk.PCharStack [stk.stIdx], S);
stk.Stack[stk.stIdx] := INF;
```

In the following, we see the `concat` function of the `strbasic.dll` plugin. A non thread-safe variable `temp` is used to store the concatenated string. In this way, collisions might result. It is better to use `getmem` to reserve memory for the `tmp` string.

```

var tmp : String;

function concat(var stk : TStack):Boolean;stdcall;
var
    Idx : Integer;
begin
    Result := False;
    Idx := stk.StIdx;

    {check if enough parameter}
    if Idx < 2 then Exit;

    {check if both parameters are strings}
    if not (Stk.Stack[Idx-1] = INF) then Exit;
    if not (Stk.Stack[Idx] = INF) then Exit;

    tmp := StrPas(Stk.PCharStack[Idx-1])+
           StrPas(Stk.PCharStack[Idx]);

    Stk.PCharStack[Idx-1] := PChar(tmp);

    stk.StIdx := Idx-1;
    Result := True;
end;

```

### 3.2 Sending partial results from a plugin

untSearchThreads.pas under /dllbuilding/gpuse

```
Stk.SendCallback (@Stk);
```

### 3.3 Broadcasting from a plugin

#### 3.3.1 How to use broadcasting in genetic algos

*explain here how genetic algos can profit by broadcasting their best individuals together with the corresponding fitness score*

### 3.4 Identification commands

The virtual machine intercepts special commands that ask the client for identification, since the philosophy of the platform is that plugins should stay unaware of their environment.

- **nodename:** this command returns the node name as set by the user
- **team:** it returns the team the user belongs to
- **country:** the country code as set by the user is returned
- **IP:** returns the outside IP number that GPU detects
- **opsys:** returns the operating system GPU detects

- **version**: returns GPU's version numbers
- **MHz**: returns computer speed in Megahertz
- **RAM**: returns how much physical memory is on board

These identification commands are useful: sending **MHz** to the cluster will return in the sum field the current power of the cluster. Similarly, **RAM** will return the amount of physical memory in the cluster.

By using the GPU Packet logger in **frontend/simple**, it is possible to identify clients who did some computation. For example, sending **3600000,pi,nodename,MHz** will return results like **3.1417,oleander,1400** and **3.1411,iron-linux,1800**.

Identification commands play a role in the last chapter as well.

### 3.5 Application Launcher

Plugin **applaunch.dll** was introduced in the framework to execute external programs that take parameters and work as a batch process. To avoid security flaws, the plugin does not accept paths with slash or dots: only **.exe** programs located in the directory **/binexec** are executed.

Syntax for the plugin is:

```
'param3', 'param2', 'param1', 'thunder', launch
```

This means that the program **thunder.exe** located in the subdirectory **binexec** of the main GPU application is started along with parameters **param1**, **param2** and **param3**.

This is equivalent to open a DOS window, move with the command **cd** to **c:/Programme/GPU/binexec** and execute there **thunder param1 param2 param3**.

As we will see in the next example, the executable is typically a wrapper to some other external application located in the subdirectory of **sandbox**. Remember that before being released on autoupdate or as a package, plugins are extensively tested to minimize potential security flaws and that the decision to autoupdate always is left to the user.<sup>9</sup>

#### 3.5.1 Example: The Terragen Plugin

Terragen[8] is a scenery generator, created with the goal of generating photo-realistic landscape images and animations. Terragen is free for personal, non commercial use. Although Terragen is a continually evolving work-in-progress, it is already capable of near-photo-realistic results for professional landscape visualization, special effects, art and recreation. Terragen has been used in a variety of commercial applications including film, television and music videos, games and multimedia, books, magazines and print advertisements[8].

At time of writing, it is possible to use the GPU cluster to render images using Terragen and in a distributed fashion. An additional program is still required to merge frames together in a video, at the end.

GPU comes along with Terragen files installed under the directory **sandbox**. In directory **binexec**, an additional GPU wrapper program called **earthsim.exe**

<sup>9</sup>GPU developers are aware that a virus or Trojan horse might hurt the research network hardly and that these kind of programs take penal consequences with them



Figure 5: *Artificial landscape* - This picture was generated in about 4 minutes on a Pentium III 1 GHz with Terragen, a landscape generator. The **applaunch** plugin can run Terragen on a GPU cluster and therefore calculate many frames at once.

is able to download and upload files from a central FTP server, and to pass parameters to the main **terrigen.exe** application stored in **sandbox** directory. Finally, the application launcher **applaunch.dll** will mediate between the main GPU application and **earthsim.exe**.

In the beginning, a **launch** command is issued to execute the GPU wrapper, either locally or globally as usual. The GPU wrapper **earthsim.exe** attempts to contact a FTP server and downloads configuration files for terrain, sun and camera position for all frames that compose the video. It then randomly decides to render one of the frames out of the available ones, and uploads back on the FTP server an empty **image\_xxxx.lock** file, to signal to other clients that frame **xxxx** is being computed, avoiding double work. Once the image is downloaded, the wrapper launches the true **terrigen.exe** program and lets it compute in batch modus, minimized in a DOS box. The computation can take from minutes to hours, depending on the image resolution and on several other factors, including terrain files and water rendering. Finally, the computed image is uploaded to the FTP server and is available for further processing: for example, to be glued in a video.

A frontend for Terragen is provided as well; However, we will describe how to setup a project manually, here. If you are a terragen artist, read further.

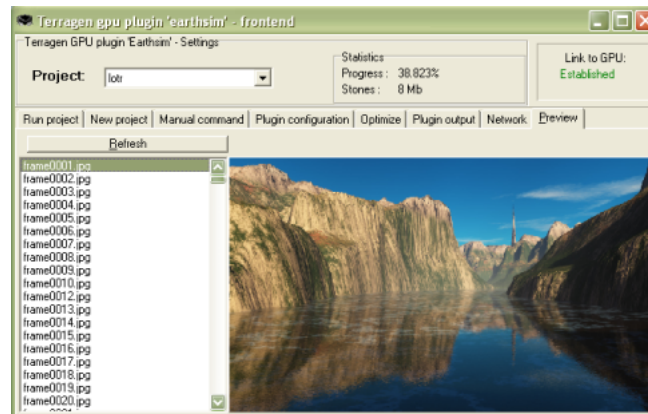


Figure 6: *Terragen Frontend* - The Terragen frontend is able to display FTP server contents and overall progress for one particular project.

**How to setup a terragen project** As first step, it is necessary to log on on a ftp sever anonymously<sup>10</sup>, for example on `ftp.dubaron.com:5555`. Then, one should create a new folder in the main directory of the FTP server. The name of the folder is the project name! In this new folder, the following files should be uploaded, and their name must exactly match.

```
terrigen.tgs
terrigen.tgw
terrigen.ter (optional)
```

At `ftp.dubaron.com:5555`, in the directory `test`, there are some sample files, too. It is now time to Configure the `tgw` nicely. At best, the reader should get documentation from [8]. *Reference good starting point!*

Generally speaking, a job should not last longer than about 2 hours to take in account older computers like PIII-500MHz (2004). Also, beware to set a proper buffer size in the terragen project. 16MB are reasonable, 32 optional. The 64MB set by default are somewhat high (2004).

Once the ftp server is set up correctly, it is time to run GPU and tell the clients about it.

In GPU, go to commands and execute the following command globally: `'5555', 'ftp.dubaron.com', 'project1', 'terrigen', launch`

The explanation of the command is straightforward, if read from right to left:

- `launch` - the actual command. It tells the GPU wrapper to search for executables in `binexec` directory with the name of left parameter.
- `'terrigen'` - name of the executable in `binexec`. It will automatically be transformed into `'terrigen.exe'`. If you need to adjust behavior by writing a batch file, take care that `exe` files are searched first.

<sup>10</sup>At the moment, all ftp access is done anonymously. In future, maybe client may look for account `terrigen`



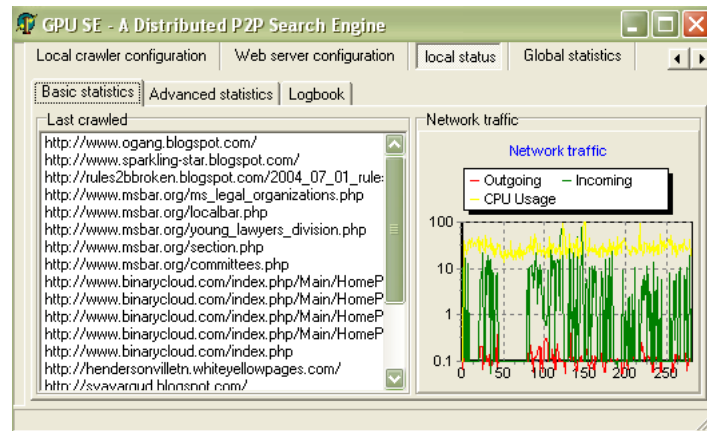


Figure 7: *Distributed Search Engine* - we see the crawler in action. Pages are indexed and stored in a MySQL database for later retrieval.

- 'project1' - the name of your project. It must *exactly* match the folder name on the ftp server.
- 'ftp.dubaron.com' - the ftp server where project files are.
- '5555' - the port number of this ftp server. If omitted, port 21 (default ftp) is assumed.

As final note, remember that on successful upload, the lock file is removed and the local computed bitmap deleted. A lock file indicates then a computation which was interrupted in the middle. To receive the missing images, one can delete all lock files and issue the `launch` command again.

With the Terragen frontend, it is possible to send out this command in periodic intervals. Typically, to generate an entire video of two minutes, it takes from two days up to a week.

## 4 Distributed Databases: The Distributed Search Engine

The Distributed Search Engine is the most complex extension GPU currently has (2004). Its development drew the development team into new fields. Improvements like *string parameters*, *partial results* and *UDP/IP channels* up to *web interfaces* turned out to be essential, in order to get the system running.

Although with many limitations, in particular search speed<sup>11</sup>, a more advanced system like the proposed Distributed Search Engine (shortly DSE) could provide a search engine difficult to shutdown in its decentral nature and difficult to censor or manipulate<sup>12</sup>. Automatic censorship against hard pornography,

<sup>11</sup>typical answer time is twenty-thirty seconds. As a comparison, the search engine Google answers in half a second

<sup>12</sup>a search for Linux on Microsoft Network returns Unix-*Windows* migration tools as first entries (2004)

crime and political extremism could be integrated in the client.

## 4.1 Description

As any complex GPU extension, the DSE is composed by two pieces: a plugin charged with the tough "computation", in this case the crawling, indexing, storing and retrieval of webpages, and a frontend charged with the "visualization", to present the plugin's work in a human readable form.

This extension allows GPU to crawl the web and index it using a SQL database. Users can then search each database created independently by GPUs for strings contained in the website or in the URL.

Using the search frontend, the user first submits to the crawler initial websites (Submit URL tab). Then, the user can enable a different number of crawlers. After selecting a place for the database, local statistics will show the visited URLs and display graphically statistics about CPU and network bandwidth usage. The new GPU command `crawlstones` will return the number of megabytes crawled by the local GPU (Compute locally button) or crawled by the entire GPU network (Compute globally button).

Experienced users can download an optimized plugin that uses `mySQL` instead of `SQLite` for better performance. The plugin should be placed in the subdirectory `/plugins` of GPU and the file `search_plugin.dll` should be removed from there.

The signature of the crawler is

Mozilla/4.0 (compatible; GPU p2p crawler [http://gpu.sourceforge.net/search\\_engine.php](http://gpu.sourceforge.net/search_engine.php)) and can be sometimes seen in statistics of visited pages.

## 5 Additional documentation

### 5.1 Local cluster configuration

**Warning** This describes the behavior of versions around 0.890 and might change in the future. In particular, GPU already implements `GWebCaches` [11].

In general, each GPU looks at the file `gpu_fix.txt` to find entry gates to the network. Each minute, the GPU client tries to establish a connection to each computer listed in the file. A typical `gpu_fix.txt` looks like this:

```
spartacus.is-a-geek.net
nanobit.is-a-geek.net:6543
129.132.12.72
129.132.12.73:4444
```

The files contains therefore DNS names or IP numbers. If a port is not specified with `:`, the default Gnutella port 6346 is taken. We emphasize that GPU will sweep through `gpu_fix.txt` once a minute and attempt to establish a connection to each computer described in each line. GPUs know each other also by broadcasting PINGs through the network. One can use the 'Disable GPU discovery' option to suppress PING broadcasting.

In order to create a local cluster, you could simply replace `gpu_fix.txt` with a list of valid computers in your local environment. At best, you will choose the

fastest computers as entry gates for the network. Additionally, GPUs will know each other through the initial entry gates and attempt connections to other clients on the local cluster.

Alternatively, one can first disable the 'Connect to GPU net' checkbox on all GPUs and add connections manually using the **Connect** button. However, TCP/IP connections tend to break down after a while, so that this method is not recommended anymore.

As final remark, if the local network should be optimized for speed, it is good idea to design `gpu_fix.txt` files such that the local network stays tree-like. On the contrary, if the network should be resistant against failures, recall that a  $k$ -connected network with  $k \geq 2$  stays connected even if  $k + 1$  nodes fail.

## 5.2 Color maps

Color maps were ported from C [9] to Delphi. The library is available as Delphi unit and C file `.c, .h` under `libraries/colormap`.

The library translates a floating point number between 0 and 1 into a (R,G,B) triple of values in range 0 to 1 again. One might need to multiply that value with 255 in some cases. Depending on which color map is selected, the triple of values is different. Internally and for each color map, the input value in range between 0 and 1 is interpolated using three different linear functions (one for R, G and B channel). These functions were taken from the program Deep Space Nine [10].

In order to use colormaps one should first define a `TColorMap` object. At the end, in a procedure like `FormClose`, the object should be freed.

```
uses colormap;
...
var CMap : TColorMap;
...
CMap := TColorMap.Create;
...
CMap.Free;
```

The appropriate color map is selected with `LoadColorMap()`. Doing so, the internal interpolation functions are changed. Allowed maps are `GREY_MAP`, `B_MAP`, `HEAT_MAP`, `COOL_MAP`, `BB_MAP`, `HE_MAP`, `A_MAP`, `RAINBOW_MAP` and `STANDARD_MAP`.

```
CMap.LoadColorMap(COOL_MAP);
```

Finally, to convert the value into RGB, we use `GetRGB()`.

```
uses Dialogs;
var Color, R, G, B : Real;
...
Color := 0.3;
CMap.GetRGB(Color, R, G, B);
ShowMessage(FloatToStr(R) + ' ' + FloatToStr(G) + ' ' + FloatToStr(B));

{this writes a line in OpenGL with the interpolated color}
```

```
glBegin(GL_LINES);
glColor3f(R, G, B);
glVertex3f(0,0,0);
glVertex3f(1,1,1);
glEnd;
```

### 5.3 Autoupdate mechanism

Current autoupdate implementation is very simple and straightforward, although some work is required from developer side. On the other side, the update system is not really powerful: rollbacks, branches and similar are not implemented. The autoupdate mechanism arisen from the need to quickly fix a release done through sourceforge. The releasing process on sourceforge can require up to 3 quarter hour. Repeating everything because of a wrong detail which compromised the network was quite common, until autoupdate came with release 0.815.

A list of files which changed is stored on a central FTP server with known address, its name is `update.txt`.

A typical `update.txt` looks like this:

```
0.9073  ,   New version number after performed update

0.828   ,   glut32.dll
0.836   ,   plugins\teapot.dll
0.836   ,   cleancode.bat
...     ,   ...
0.860   ,   mkdir plugins\input
0.863   ,   del plugins\piproject.dll
0.885   ,   frontend\simple\simple.exe
0.9072   ,   gpu.exe
0.9073   ,   messages.txt
```

This list is downloaded by the autoupdate program. The first line of `update.txt` advertises the latest experimental release. Autoupdate downloads an additional file, `gpu_version.txt` where the latest stable version is mentioned at the top. Depending on user's settings, autoupdate will choose either `update.txt`'s experimental version number or the normally lower `gpu_version.txt`'s stable number. Autoupdate checks if the current version stored in `gpu.ini` is lower than the stable or experimental version depending on user's settings. If not, the program exits with a *nothing to be done* message.

Autoupdate closes all running instances GPU and of frontends like Network Mapper and Terragen frontend, e.g. Reason for this is that overwriting a running application is not allowed by the operating system<sup>13</sup>. All files between current version and target version are downloaded from the ftp server and stored relatively to the GPU path. Finally, the current version in `gpu.ini` is set to the new stable or experimental version, respectively.

<sup>13</sup>In order to update the autoupdate program a file called `updater2.exe` is downloaded. Once autoupdate closes and runs GPU, GPU detects the file `{textttupdater2.exe}` and renames it into `updater.exe`.

Checking 'force autoupdate' option will change autoupdater behavior: the test if the version number is lower is skipped. This option is used by developers if short development iterations are needed.

## 5.4 Job Modifiers

GPU introduced job modifiers to be added on the command string like "chat-bot:14:". That command means the command is redirected to the channel 14 of the chat system.

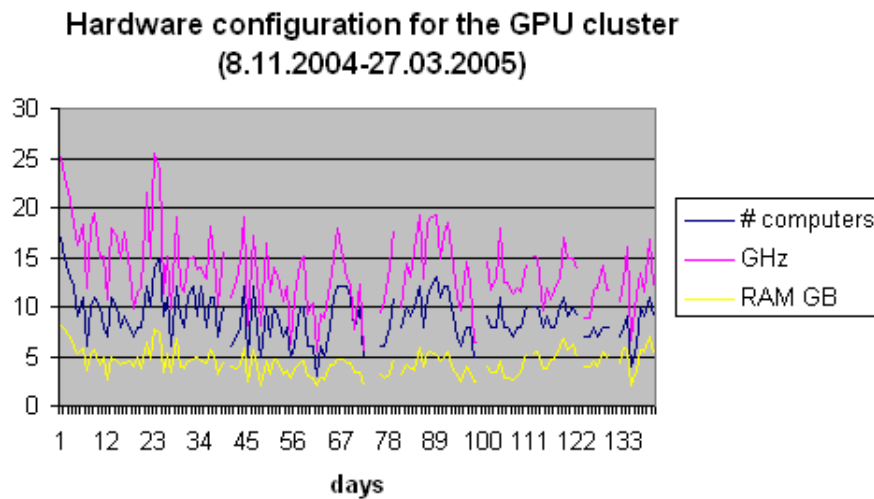
Three additional and useful modifiers are "user:username:", "team:teamname:" and "country:countrycode:". They narrow the job request to special users, teams and countries. This can be used to build an instant message system on top of GPU and in general to send direct commands to other computers. Internally, the command is still broadcasted to everyone, but only clients with the matching code will react.

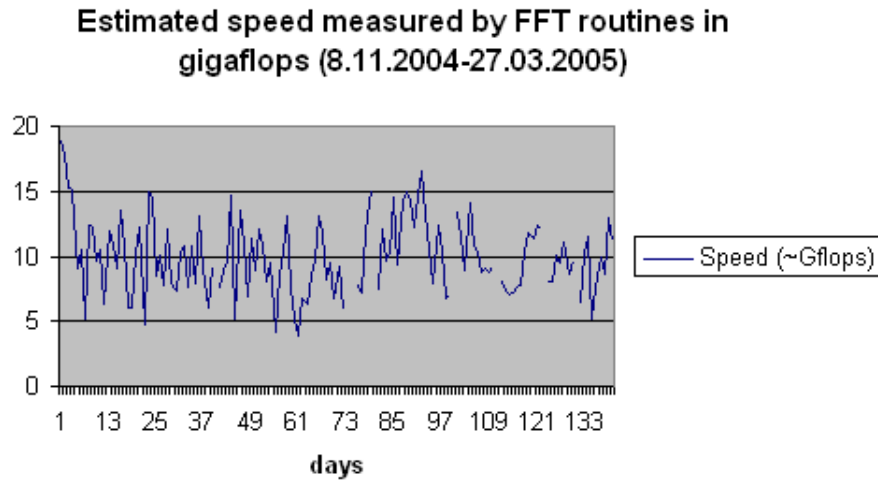
As an example, if one wants to send the job 1,1,add only to the computer `gemini`, he/she should type `user:gemini:1,1,add` and then click on the `Compute globally` button.

### 5.5 Cluster behaviour

In the following, we see performance for the cluster in a span of about three months. Each day, a snapshot of the cluster was taken. The first image shows how many computers were online, their total amount of gigahertz by adding up individual processor speeds and the total amount of RAM. The second image shows the speed measured by an internal FFT routine in `fftplug.dll` on each computer when the GPU program starts up. This number reflects better the real performance of a computer, taking in account speed between processor and memory. The third image shows how big the database of the search engine described previously was (in gigabytes). The fourth image is a report for each computer of the amount of images computed with Terragen.

#### 5.5.1 GPU Cluster statistics between 08.11.2004 and 06.01.2005





## 6 TPastella as general communication layer for P2P

[12] *Following still very rough:* What is pastella

Pastella is an alternative p2p protocol, just like gnutella is. the main goal is to perform network optimalization, at the cost of some memory power and cpu cost for some intelligent caching and hashing. It's design goals are: less network overhead, avoidance of redudant data. good broadcast support. minimalization of packet loss. host-to-host routing by other hosts. walker and agent support.

What is TPastella

TPastella is an object pascal implementation of the Pastella protocol. It is a simple component, build on the Visual Synapse Server library base.

Why is it called Pastella?

Well. there are several fantasy-rich theories about that. For example, that we replaced the GNU from GNUtella to PASTella to honour the language Pascal.

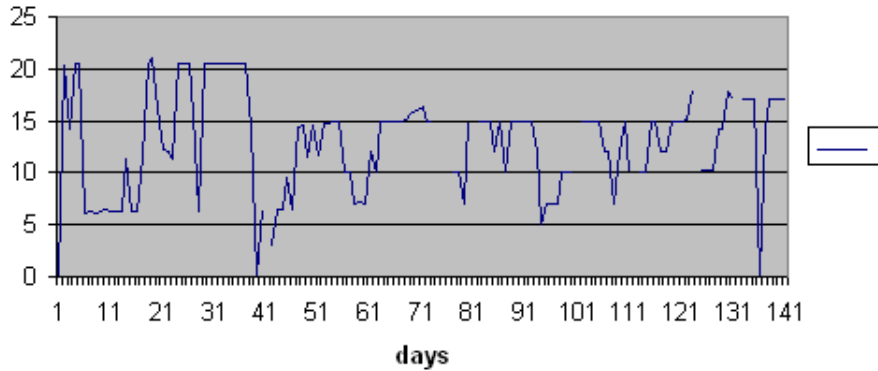
Where can i download pastella? You can fetch latest version from CVS, generally the preferred method. It is also part of the Visual Server distribution. Both can be found at sourceforge files section, visual server package: <http://sf.net/projects/visualsynapse>

## 7 Future development ideas

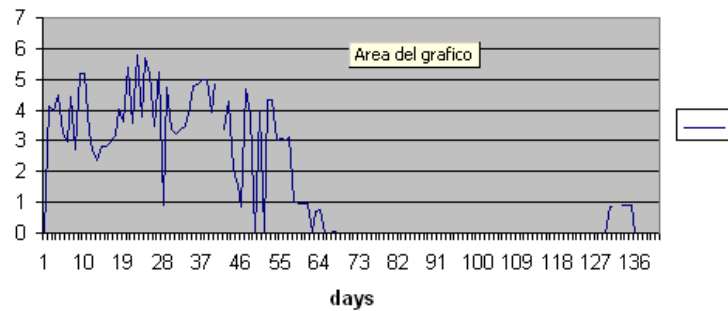
### 7.1 Turning the broadcast framework into an agent framework

GPU already includes a file transfer system offered by the TGnutella component, a PGP component to verify integrity of files and logic to load and unload plugins at runtime.

**Available crawled data in gigabytes for  
websearches (8.11.2004-27.03.2005)**



**Amount of Terragen images reported by nodes on  
the cluster in gigabytes (8.11.2004-27.03.2005)**



Naturally, one might think of a collection of routines, an API offered to plugins, so that they can move through the network in an agent-like manner. This API might include calls like `MoveToTarget(NodeIP : String)`, `CopyToTarget(NodeIP : String)`, `GetNetworkTopology : TIPNodeGraph`.

Once a plugin issues a `MoveToTarget(...)` call, the GPU will charge itself with the file transfer to the target node, the loading of the plugin at the target, after verifying with PGP that it was not changed to run malicious code. If all operations were successful, GPU should then unload the plugin which issued the request.

`MoveToTarget()` would then mimic the weak transition paradigm, where binary code is moved through the network. However, the internal plugin status (process counter, stack, heap) is not copied to the target.

The `GetNetworkTopology(...)` call should return a graph of the current network, and particularities of the nodes, in a similar way the network mapper



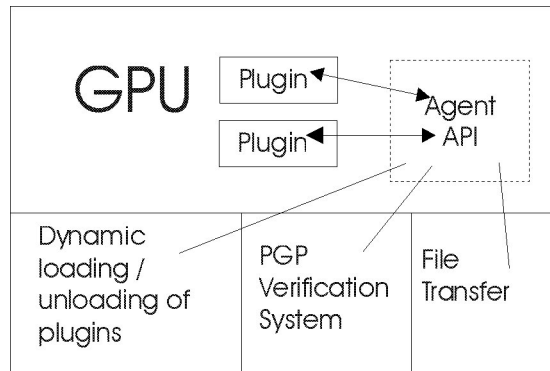


Figure 8: *GPU as a platform for agents* - A missing API might give plugins the possibility to move around the network.

already does. Using this call, plugins can decide if and where to move.

The `CopyToTarget(...)` call might be used to establish a cooperative network of plugins. Using broadcast services and normal network traffic they might exchange internal status information. `CopyToTarget()` could be used to spread new plugins or updated plugins through the network without needing autoupdate services. Other calls like `MoveToRandomTarget()` should then use the three previous routines as building blocks.

As an idea, the GPU framework could run a competition of plugins, which should visit as many nodes as possible in the network, and then return back to the initial node.

Such plugins might be named agents, although literature does not seem to imply that an agent is software which moves from one computer to another. Agents could be used for non-critical jobs like collecting statistics and searching for particular resources.

## 7.2 Global Earthquake and weather station sensor network

# 8 Conclusion

Developing with lazarus IDE and freepascal

## 8.1 Acknowledgments

Rene entirely developed Pastella, the Distributed Search Engine, Application Launcher for Terragen and most extensions including chatbot Gina. Tiziano did integration and documentation work. Many thanks go to our artists paula-treides, red, MM, nico for their breath-taking Terragen videos and to our users, in particular to swiftstream, johnatemp, lwm for running GPU for very long times.

If you found omissions or mistakes in this document, please mail them to `gpu-world@lists.sourceforge.net`. This document is version 0.700.

## 8.2 Legal notice

Terragen[8] is copyrighted by PlanetSide Software. Terragen is also a registered trademark. It is free for non-commercial use. Commercial users are required to register.

GPU, its frontends and plugins (without Terragen) are under the GPL, the GNU General Public License. TGnutella is commercial and cannot be distributed. However, we can share TGnutella source code among people working on the GPU project without paying additional fees.

Copyright ©2002-2005 the GPU Development Team, all rights reserved.

## 9 References

- [1] R. Tegel, *VisualSynapse*, 2004, available from <http://visualsynapse.sourceforge.net>.
- [2] L. Gebauer, *Synapse*, 2004, available from <http://synapse.ararat.cz>.
- [3] GPU Development Team, *The GPU project*, Sourceforge, 2002-2004, source code available from <http://gpu.sourceforge.net>.
- [4] T. Mengotti, W. P. Petersen and P. Arbenz, *Distributed computing over Internet using a peer to peer network*, September 2002, available from <http://gpu.sourceforge.net>.
- [5] T. Mengotti, W. P. Petersen, F. Marchal, K. Nagel *GPU, a framework for distributed computing over Gnutella*, April 2004, available from <http://gpu.sourceforge.net>.
- [6] R. Vivrette, *Getting the Message, Simple Techniques for Communicating Between Applications*, Delphi Informant, November 1997, <http://www.und.com/Articles/991221b.html>
- [7] delphi.about.com, *Introducing Borland Delphi*, 2001, <http://delphi.about.com/library/weekly/aa031202a.htm>.
- [8] PlanetSide Software, *Terragen - photo realistic scenery rendering software*, United Kingdom, 2004, <http://www.planetside.co.uk/terrigen>.
- [9] Hubble Development Team, *Hubble in a bottle - a 3D particle viewer optimized with SSE, 3dNow! and MPI*, University of Zuerich, 2003 <http://hubble.sourceforge.net>.
- [10] High Energy Astrophysics Division, *DS9: Astronomical Data Visualization Application*, Harvard University, 2002, <http://hea-www.harvard.edu/RD/ds9>.
- [11] Gnucleus Team, *Gnutella Web Caching System*, 2002, <http://www.gnucleus.com/gwebcache>.
- [12] Rene Tegel, *Pastella, a multipurpose P2P connection layer*, 2005, <http://sourceforge.net/projects/visualsynapse>.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Frontend improvements</b>	<b>3</b>
2.1	Frontend Communication through UDP/IP . . . . .	3
2.1.1	General proceeding . . . . .	4
2.1.2	Format for request and result . . . . .	4
2.1.3	Frontend Communication with Delphi and Visual Synapse component . . . . .	5
2.1.4	Receiving results from GPU . . . . .	5
2.1.5	Sending a command to GPU . . . . .	5
2.2	Registering a frontend to network broadcasts . . . . .	6
2.3	Plugins can use the same UDP/IP channel to submit and receive results . . . . .	6
2.4	Adding a webinterface to a frontend . . . . .	8
2.4.1	Using THTTServer . . . . .	9
2.4.2	Multiple virtual directories . . . . .	9
2.4.3	Recursive mapping . . . . .	10
2.4.4	Configuring the server . . . . .	10
<b>3</b>	<b>Plugin improvements</b>	<b>11</b>
3.1	String as parameters for the virtual machine . . . . .	11
3.1.1	Examples from the user's perspective . . . . .	11
3.1.2	Examples from the developer's perspective . . . . .	12
3.2	Sending partial results from a plugin . . . . .	13
3.3	Broadcasting from a plugin . . . . .	13
3.3.1	How to use broadcasting in genetic algos . . . . .	13
3.4	Identification commands . . . . .	13
3.5	Application Launcher . . . . .	14
3.5.1	Example: The Terragen Plugin . . . . .	14
<b>4</b>	<b>Distributed Databases: The Distributed Search Engine</b>	<b>17</b>
4.1	Description . . . . .	18
<b>5</b>	<b>Additional documentation</b>	<b>18</b>
5.1	Local cluster configuration . . . . .	18
5.2	Color maps . . . . .	19
5.3	Autoupdate mechanism . . . . .	20
5.4	Job Modifiers . . . . .	21
5.5	Cluster behaviour . . . . .	22
5.5.1	GPU Cluster statistics between 08.11.2004 and 06.01.2005 . . . . .	22
<b>6</b>	<b>TPastella as general communication layer for P2P</b>	<b>23</b>
<b>7</b>	<b>Future development ideas</b>	<b>23</b>
7.1	Turning the broadcast framework into an agent framework . . . . .	23
7.2	Global Earthquake and weather station sensor network . . . . .	25

<i>CONTENTS</i>	29
<b>8 Conclusion</b>	<b>25</b>
8.1 Acknowledgments . . . . .	25
8.2 Legal notice . . . . .	26
<b>9 References</b>	<b>27</b>