GSORF

Documentation

Georg-Simon-Ohm Renderfarm (GSORF)

Table of contents

Terms and Definitions....6
Render / Rendering....6
Renderfarm....6
Server....6
Client....6
Blender 3D....6
Job....6
Task....6
UDP....7
TCP....7
Qt....7
Introduction....7
System Requirements....7
Minimum hardware....7
Recommended hardware....7
Operating System....8
Software dependencies....8
Installation....8
GSORF....8
Blender....8
Implementation specification....9
Frameworks and Libraries....9
Qt 5....9
OpenSSL....9
ffmpeg....9
Source Code Documentation....9
gsorf.pro....9
main.cpp....9
blenderrenderengine.h, blenderrenderengine.cpp....9
void open(Task_Client &_theTask);....10
void saveTaskFile();....10
void renderTaskFile();....10

void createResult();.....10
void loadRenderedImage();.....10
void setBlenderPath(QString path);.....10
QString getBlenderPath();.....10
void quit();.....10
client.h, client.cpp.....10
void start();.....10
void stop();.....11
void onTCPServerNewConnection();.....11
void onTCPUpdateServerProgress();.....11
void onTCPConnectionClosed();.....11
void displayError(QAbstractSocket::SocketError);.....11
void onProcessTaskClient(Task_Client &clientTask);.....11
void onSetServerAddress(QHostAddress sender);.....11
void onSendResultBack(Result result);.....11
dialog.h, dialog.cpp, dialog.ui.....11
namespace RenderNode{
enum Icon;
class Node;
}.....12
struct taskStc;.....12
void startGSORFServer();.....12
void stopGSORFServer();.....12
void initBroadcast();.....12
void refreshWorkers();.....12
void callAvailableWorkers();.....12
void callIntelligentWorkers();.....12
void generateRandomGSORFServerPriority();.....13
void callAvailableServers();.....13
void checkResultID(Result &result);.....13
void saveResultOnHarddisk(Result &result);.....13
void checkResultComplete(Result &result);.....13
void packToVideo(Result &result);.....13
void serverMsgLogger(QString msg);.....13

void resendTask(QString taskKey);....13
void callAvailableWorkersPediodicly();....13
quint64 calculateSumOfIPAdresses(QQueue<RenderNode::Node> renderNodeList);....14
QListWidgetItem createRenderNodeEntry(RenderNode::Node renderNode);....14*
void updateIPLists(QQueue<RenderNode::Node> &renderNodeList, QString senderIP, bool file_cached);....14
void serverLoggerSLOT (QStringList loggerStrList);....14
void onBtnClickedNuke();....14
void render_btn_clicked();....14
void refreshAutoServerDisplay();....14
void setServer();....14
void setClient();....15
void initBroadcastSlot();....15
void onBtnExpertMode();....15
void onBtnSimpleMode();....15
void onDebugBtnClicked ();....15
void theJobHub();....15
result.h, result.cpp....15
task_client.h, task_client.cpp....15
task_server.h, task_server.cpp....15
void initTask(QString filename, int startframe, int endframe, QString RenderEngine, bool CPU, bool stereo3D, QByteArray &file, QString jobId, bool cached);....16
void sendTo(QString host, unsigned int port);....16
void startTransfer();....16
void updateClientProgress(qint64 numBytes);....16
void displayError(QAbstractSocket::SocketError socketError);....16
taskhandler.h, taskhandler.cpp....16
void addTask(int jobCount, short scheduleMode, int stcStart, int stcTotalFrames, QString stcRender, bool stcCPU, bool stcStereo3D, QString stcFileName, QString jobId);....16
taskStc popCpuQueue();....16
taskStc popGpuQueue();....17

taskStc getCpuFirst();....17
taskStc getGpuFirst();....17
bool cpuQueueIsEmpty();....17
bool gpuQueueIsEmpty();....17
void setJobCount(int change);....17
int getJobCount();....17
udpmanager.h, udpmanager.cpp....17
void writeUDP(QString status, QHostAddress receiver);....17
void readUDP();....18
void start();....18
void stop();....18
webserver.h, webserver.cpp....18
void start();....18
void stop();....18
void incomingConnection(qintptr handle);....18
QString formatTime(qint64 msecs);....18
websocketrunnable.h, websocketrunnable.cpp....19
void run();....19
Operation....20
1. Overview....20
2. Starting the Server....20
3. Starting the Client....22
4. Rendering a .blend file....23
5. Monitoring the rendering progress....24
6. Collecting the finished renderings....25
Use Case: Stereo 3D....26
1. Prepare your .blend file....26
2. Check the stereo 3d option in the GSORF Server....26
Use Case: Distributed Tile Rendering with Cycles....27
1. Prepare your .blend file....27
2. Send it to the Renderfarm....27
3. Combine the tiles....27

Terms and Definitions

We use specific definitions in this document which we would like to explain first.

Render / Rendering

To render something means converting a scene description file (a text file or 3d format, e.g. .blend, .obj, .3ds, etc.) into an image or video (i.e. .jpeg, .tiff, .tga, .mov, etc.). The file resulting from the render process is called a rendering.

Renderfarm

Since rendering a 3d scene might take up quite a bit of time, you usually don't use only one computer to do the work but distribute the work over a network of several computers. This is then called a Renderfarm.

Server

Our Server is the central software component where a user can specify a new job and send it to the clients.

Client

Our Client is the software component waiting for jobs from the server. The server will send new tasks to available clients where the rendering is being done.

Blender 3D

Blender 3D is an open source graphic suite utilized by GSORF clients to render scene files (.blend).

Job

Every new user intended project which is sent to the network is called a job. It is defined with the scene file, start frame, end frame, processing method and so on.

Task

While Jobs can contain several frames, a Task is one specific frame of a Job. Before being sent on the network, jobs are split up into several Tasks which are then given to a Client, one by one.

UDP

UDP stands for „User Datagram Protocol“ and is being used in our project for fast status updates.

TCP

TCP stands for „Transmission Control Protocol“ and is being used in our project for the transmission of Tasks and Results. It is a connection-oriented network protocol and ensures that the transmission is done properly.

Qt

Qt (pronounced „cute“) is a cross platform framework for making development of GUI Applications easier. We decided upon Qt while developing our software having platform independent software in mind (like Windows, Mac OSX and Linux). Furthermore Qt offers a wide range of libraries for many tasks, like SQL databases, XML parsing, multi-threading or network communication.

Introduction

The Georg-Simon-Ohm Renderfarm (in short GSORF) allows you to connect several computers to distribute the rendering of three-dimensional Blender scenes (.blend) onto those machines.

System Requirements

Minimum hardware

- 32-bit dual core 2Ghz CPU with SSE2 support.
- 2 GB RAM
- 24 bits 1280×768 display
- Mouse or trackpad
- OpenGL-compatible graphics card with 256 MB RAM

Recommended hardware

- 64-bit quad core CPU
- 8 GB RAM
- Full HD display with 24 bit color
- Three button mouse
- OpenGL-compatible graphics card with 1 GB RAM

Operating System

GSORF has been developed with the cross-platform framework Qt, thus it can be run on Windows, Mac OS, Linux and similar operating systems.

Software dependencies

Since the rendering is done with Blender, you need to have it installed on every computer that will be rendering later on.

Installation

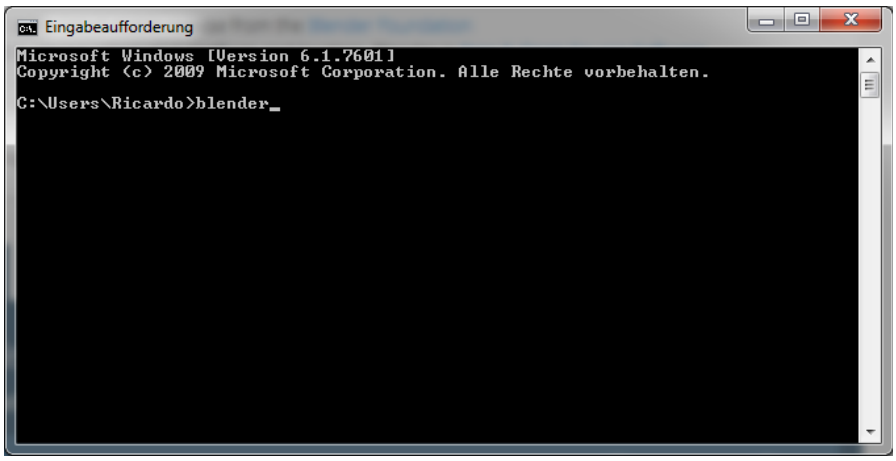
GSORF

You don't need to install GSORF. Just download, unzip and run it on every computer in your network.

Blender

Blender can be downloaded directly from the Blender Foundation website: http://www.blender.org/download/

To check if Blender has been installed properly, open a Terminal Window and type in „blender“, like this:



If this command opens Blender you are ready to use GSORF.

Implementation specification

Frameworks and Libraries

Qt 5

We use the Qt framework for cross-platform user interfaces and components. Within Qt we use the core, gui, network and webkitwidgets modules. You might also need to install the sqlite3 library for Qt to compile the project.

OpenSSL

We use the OpenSSL encryption algorithms for generating md5 file hashes to implement reliable caching.

ffmpeg

We use ffmpeg to automatically create video files after a rendering is finished.

Source Code Documentation

gsorf.pro

This is the Qt project file, where libraries are linked and post processing (e.g. platform independent copying of the asset and webinterface folders) are defined.

main.cpp

This file is the compilation entry point. It just defines a new *QApplication* and executes the main loop.

blenderrenderengine.h, blenderrenderengine.cpp

The *BlenderRenderEngine* class implements the logic for rendering *Tasks*, e.g. saving the .blend, rendering via Blender, loading the rendered images, creating a *Result* and

handing it over to the *Client* instance.

void open(Task_Client &_theTask);

This function is called when a new Task from the server arrives. It resets the gui and prepares everything to start rendering. For this a QProcess is needed, so Blender can be started in the background. Callback functions for the process are connected and we can proceed to the next step.

void saveTaskFile();

This function extracts the .blend file from the *Task* datatype and saves it on the harddrive as “awesome.blend”. Now the file can be rendered.

void renderTaskFile();

This function takes the saved .blend file and hands it to a Blender instance running in the background. After that, Blender is started and running on its own thread to avoid the gui from freezing. The QProcess does then read the state and output of Blender to react on any changes and completeness.

void createResult();

This function is called when the QProcess knows that Blender has finished rendering. It creates an appropriate *Result* datatype, which can then be sent back to the Server.

void loadRenderedImage();

This function is called when .

void setBlenderPath(QString path);

This function is called when a new Task from the server arrives. It resets the gui and

QString getBlenderPath();

This function is called when a new Task from the server arrives. It resets the gui and

void quit();

This function is called when a new Task from the server arrives. It resets the gui and

client.h, client.cpp

The *Client* class implements the logic of render clients. Namely, TCP receiving (*Tasks* from *Server*) and sending (*Results* to *Server*), threading, starting/stopping rendering and creating GUI feedback.

void start();

This function starts the Client.

void stop();

This function stops the Client.

void onTCPServerNewConnection();

This slot is called when a new TCP Connection is established. Here everything is being prepared to receive a new Task.

void onTCPUpdateServerProgress();

This slot is called when the TCP Connection is read. Here the Task is being build up while receiving it on the network.

void onTCPConnectionClosed();

This slot is called when a TCP Connection is closed. Here the received Task is saved and handed over to the RenderEngine.

void displayError(QAbstractSocket::SocketError);

This slot is called when a TCP error occurs. This error handling method resets the state of the client to an available state.

void onProcessTaskClient(Task_Client &clientTask);

This slot is called when a new Task was completely created. The Client is set to the busy state and the RenderEngine is started.

void onSetServerAddress(QHostAddress sender);

This slot sets the serverAddress like a setter function.

void onSendResultBack(Result result);

This slot handles everything to send a finished result back to the server.

dialog.h, dialog.cpp, dialog.ui

Our implementation of the QDialog class is the logic for the *Server*! It also handles every GUI functionality and creates instances of the other classes. It contains code for the initial program behaviour (e.g. the auto select mechanics, udp broadcasting to determine server). Class representations of *RenderNodes* are also defined here.

```
namespace RenderNode{
        enum Icon;
        class Node;
    }
```

This data structure holds a new RenderNode and its attributes. It is used when connected Clients (= Render Nodes) need to be displayed in the server and provide easy access to timestamps, icons and the IP address.

```
struct taskStc;
```

This data structure contains information about a certain task. It stores everything a user can set via the server user interface and internal parameters like the job id.

```
void startGSORFServer();
```

This function starts the GSORF server and also the web server.

```
void stopGSORFServer();
```

This function stops the GSORF server and also the web server.

```
void initBroadcast();
```

This function broadcasts the random server priority number via UDP to all other gsorf instances. It was planned to use it to auto-determine a new server on the network. This is currently handled manually by the user.

```
void refreshWorkers();
```

This function is calling the next two functions to determine if clients are available and cached already.

```
void callAvailableWorkers();
```

This function sends a status question whether there are Blender clients on the network via UDP.

```
void callIntelligentWorkers();
```

This function sends a file hash question to ask every client about its file hash.

void generateRandomGSORFServerPriority();

This function generated a random number for use in determining who can be the server.

void callAvailableServers();

This function sends a status question whether there are already servers on the network or not via UDP.

void checkResultID(Result &result);

This function checks if a received result has been send as a task before from this server.

void saveResultOnHarddisk(Result &result);

This function creates the directories and files for the finished results depending on whether it is a 2D or 3D image.

void checkResultComplete(Result &result);

This function checks if a received result has been the last one from a certain job and if it is completed now.

void packToVideo(Result &result);

This function creates a video file from a 2d rendering image sequence via ffmpeg.

void serverMsgLogger(QString msg);

This function adds a message to a server log file on the hard disk.

void resendTask(QString taskKey);

This function adds a task to the task queue.

void callAvailableWorkersPediodicly();

This function refreshes the client IP lists in certain time intervalls.

quint64 calculateSumOfIPAdresses(QQueue<RenderNode::Node> renderNodeList);

This function sums all client IP addresses to recognize a change in the IP list (which results in a reset of the IP list).

QListWidgetItem* createRenderNodeEntry(RenderNode::Node renderNode);

This function creates object representations of the IP list items. This makes it easy to style the elements with colors and icons.

void updateIPLists(QQueue<RenderNode::Node> &renderNodeList, QString senderIP, bool file_cached);

This function checks whether the current .blend file and a render client match and the file can be displayed as cached. It also determines if a render client is online or offline via comparing its last timestamps.

void serverLoggerSLOT (QStringList loggerStrList);

This slot logs sending of a task in the log file on the hard disk.

void onBtnClickedNuke();

This slot will send a command to kill all blender instances on a client via UDP.

void render_btn_clicked();

This slot handles the adding several new task, which belong to one certain job, to the internal task list, so that they can be sent to available clients.

void refreshAutoServerDisplay();

This slot refreshes the auto select display in the user interface.

void setServer();

This slot switches the view to the server perspective.

void setClient();

This slot switches the view to the client perspective.

void initBroadcastSlot();

This slot starts the auto select broadcasting via UDP.

void onBtnExpertMode();

This slot shows the expert mode in the user interface.

void onBtnSimpleMode();

This slot hides the expert mode in the user interface.

void onDebugBtnClicked ();

This slot shows the number of total jobs in the debug console.

void theJobHub();

This slot is the most important for sending tasks to available clients. It is called in fixed time intervalls and sends tasks as long as there are tasks to do and available clients.

result.h, result.cpp

The *Result* class implements a custom data type for the finished renderings that are being sent back to the *Server* on the network.

task_client.h, task_client.cpp

The *Task_Client* class implements a custom data type for the tasks, that are being created from a job and used in the client logic.

task_server.h, task_server.cpp

The *Task_Server* class implements a custom data type for the tasks, that are being

created from a job and used in the server logic.

void initTask(QString filename, int startframe, int endframe, QString RenderEngine, bool CPU, bool stereo3D, QByteArray &file, QString jobId, bool cached);

This function initializes a new Task.

void sendTo(QString host, unsigned int port);

This function prepares (serializes) a Task to be sent to a client.

void startTransfer();

This slot initiates the transfer of a Task to a client.

void updateClientProgress(qint64 numBytes);

This slot constantly writes bytes to the network until a Task is completely sent.

void displayError(QAbstractSocket::SocketError socketError);

This slot does the error handling for the Task send function from server to client

taskhandler.h, taskhandler.cpp

The *TaskHandler* class implements logic for managing every *Task*.

void addTask(int jobCount, short scheduleMode, int stcStart, int stcTotalFrames, QString stcRender, bool stcCPU, bool stcStereo3D, QString stcFileName, QString jobId);

This function adds a new task to the TaskHandler depending on its priority.

taskStc popCpuQueue();

This function deletes the first element in the cpu Task queue.

taskStc popGpuQueue();

This function deletes the first element in the gpu Task queue.

taskStc getCpuFirst();

This function returns the first element in the cpu Task queue.

taskStc getGpuFirst();

This function returns the first element in the gpu Task queue.

bool cpuQueueIsEmpty();

This function returns whether the cpu Task queue is empty or not.

bool gpuQueueIsEmpty();

This function returns whether the gpu Task queue is empty or not.

void setJobCount(int change);

This function increases the job count by a certain value.

int getJobCount();

This function returns the job count.

udpmanager.h, udpmanager.cpp

The *UdpManager* class implements logic for managing sending and receiving status updates via the udp protocol.

void writeUDP(QString status, QHostAddress receiver);

This function broadcasts a specific status via UDP.

void readUDP();

This function reads specific status messages which are received via UDP.

void start();

This function activates the UDP socket to receive status messages via the UDP protocol.

void stop();

This function deactivates the UDP socket to stop receiving status messages via the UDP protocol.

webserver.h, webserver.cpp

The *WebServer* class implements a multi-threaded HTTP-Server, that is listening on port 8080.

void start();

This function starts the web server to listen on port 8080.

void stop();

This function stops the web server to listen on port 8080.

void incomingConnection(qintptr handle);

This function creates a new Thread as soon as there is a new connection incoming. The Thread is then initialized and started.

QString formatTime(qint64 msecs);

This function takes milliseconds and converts them to a time format which can then be displayed as the “remaining render time”, “elapsed render time” and so on in the webinterface.

websocketrunnable.h, websocketrunnable.cpp

The *WebSocketRunnable* class implements a *QRunnable* which is used by the *WebServer* for multi-threading. It also reads the html file from the /htdocs folder and replaces placeholders with data from the server.

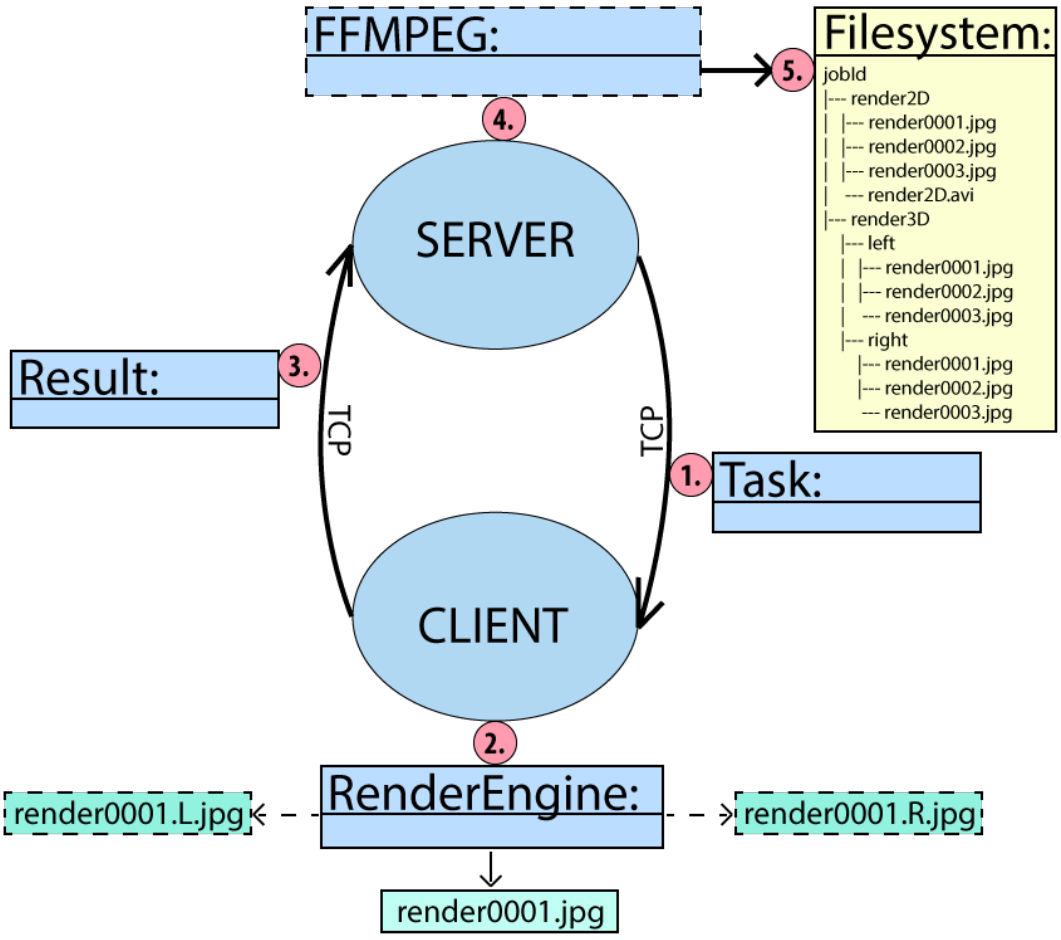
void ***run();***

This function is responsible for displaying a html website in the web interface. First all bytes on the network are read, then a proper http response is generated according to the http request. Depending on what type of media the browser requested either the template html file (with replaced placeholders for dynamic data) or a media file is served.

Operation

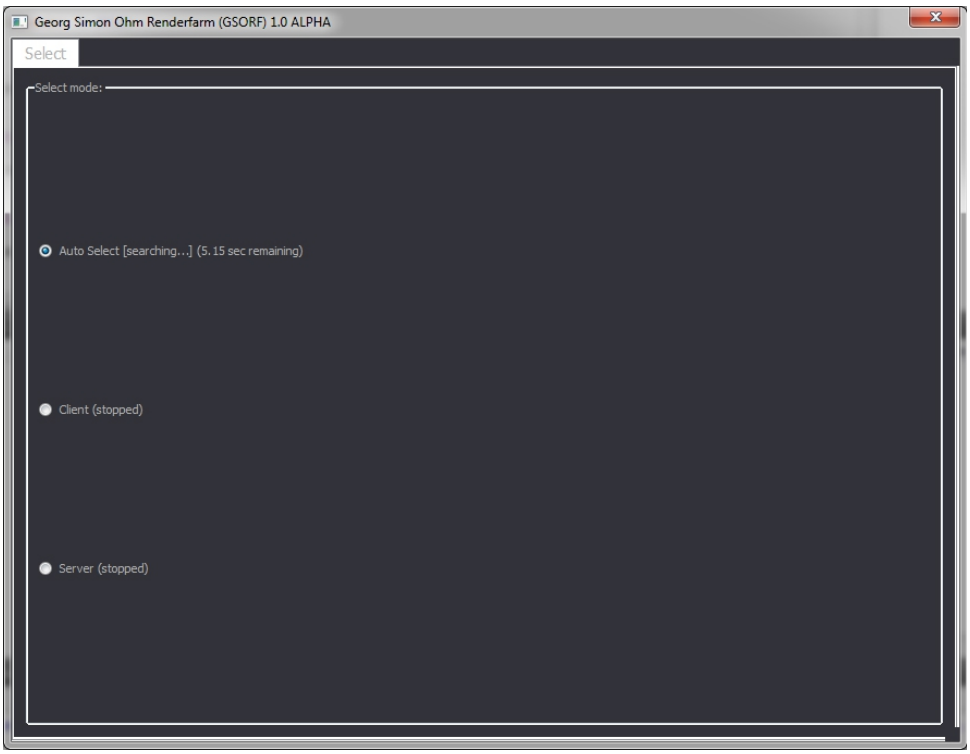
1. Overview

GSORF consists of two program components: The server and the client.
As it's the brain of your renderfarm the server controls data transmission and allows you to start a new rendering. This in turn is then sent to all other computers which are running the client. Clients will render the scene and send back the finished image to the server. You can then collect the image files for video editing and compositing.

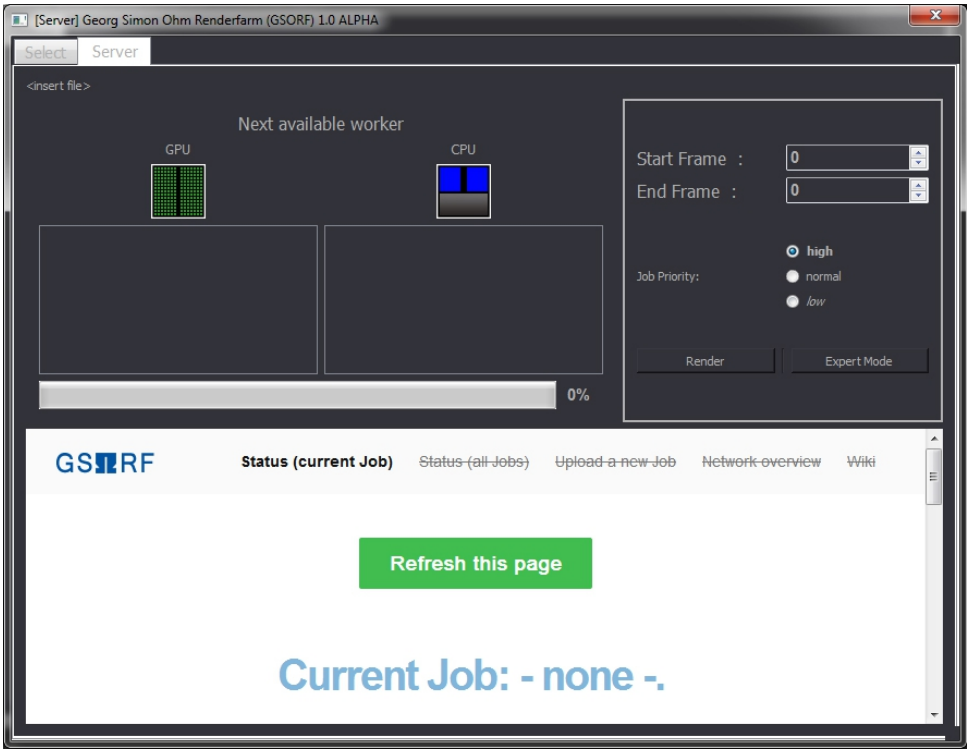


2. Starting the Server

Start GSORF on a machine where you want to store all your finished renderings. It will automatically search for a server in the network. This should fail since there are no servers on your network right now.



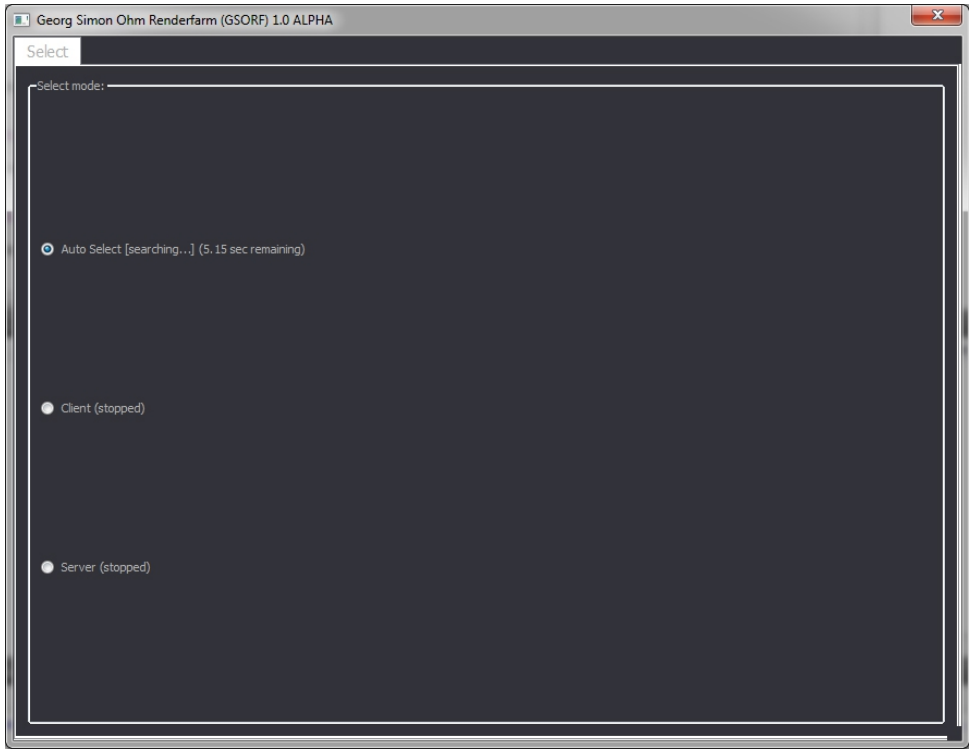
Select „Server“ from this list. You are now presented with the server perspective.



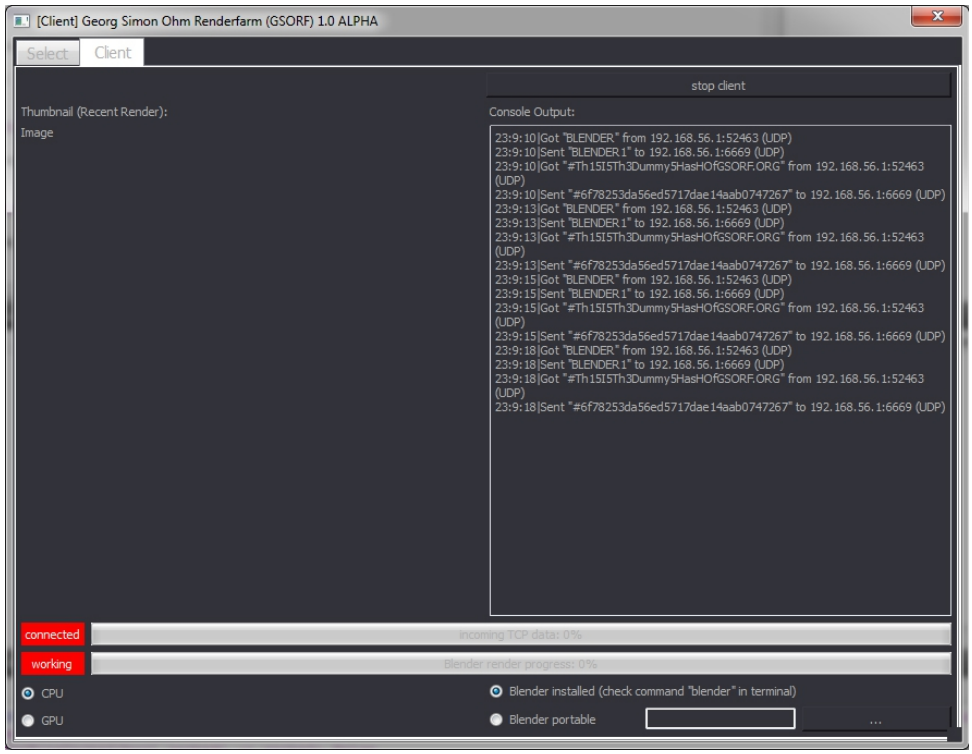
For now we will leave the server open and return here again as soon as some clients are running.

3. Starting the Client

Start GSORF on the remaining machines as well.



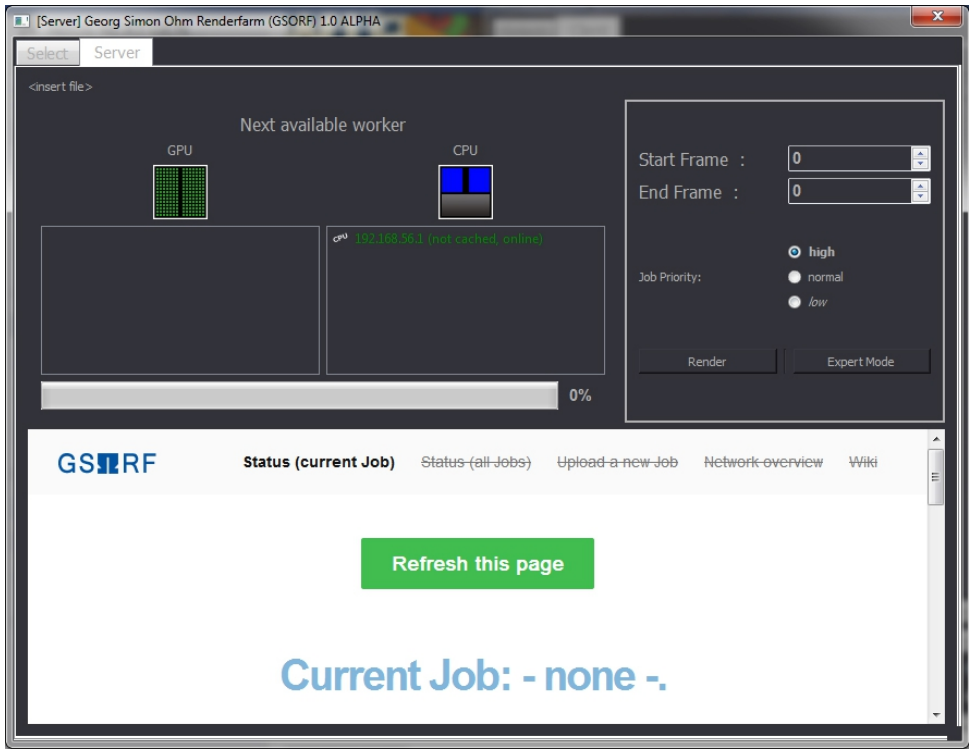
It should now automatically recognize the server on the network and switch to the client perspective.



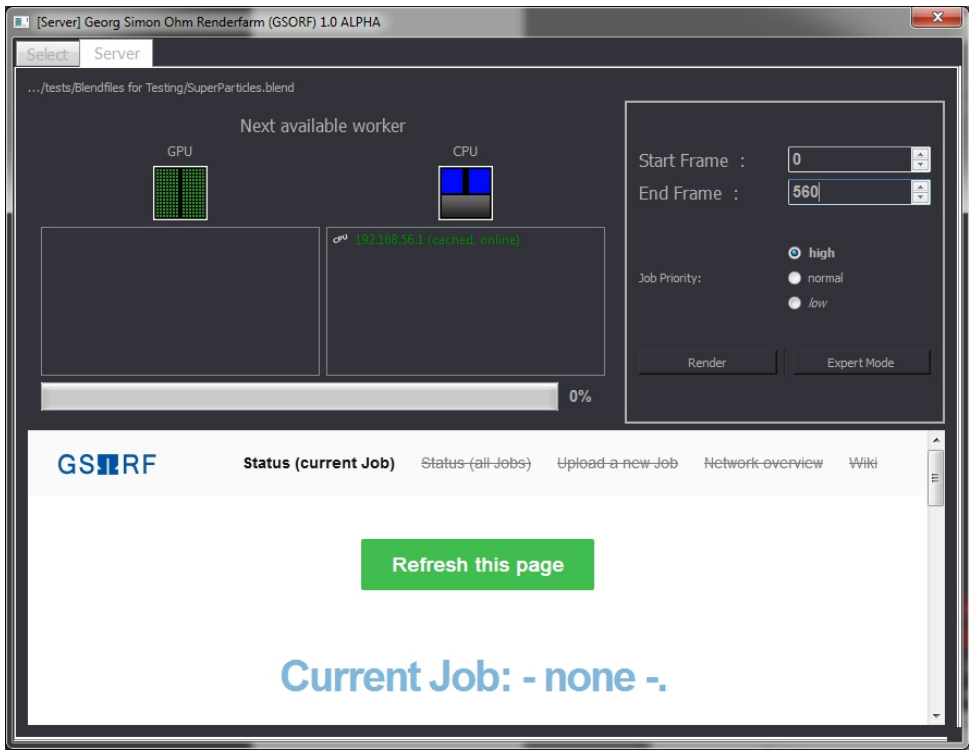
Repeat this step on every machine you want to use for rendering.

4. Rendering a .blend file

You have now set up everything to start rendering your .blend file on the network. Take a look at the server. It should now display all available render clients.



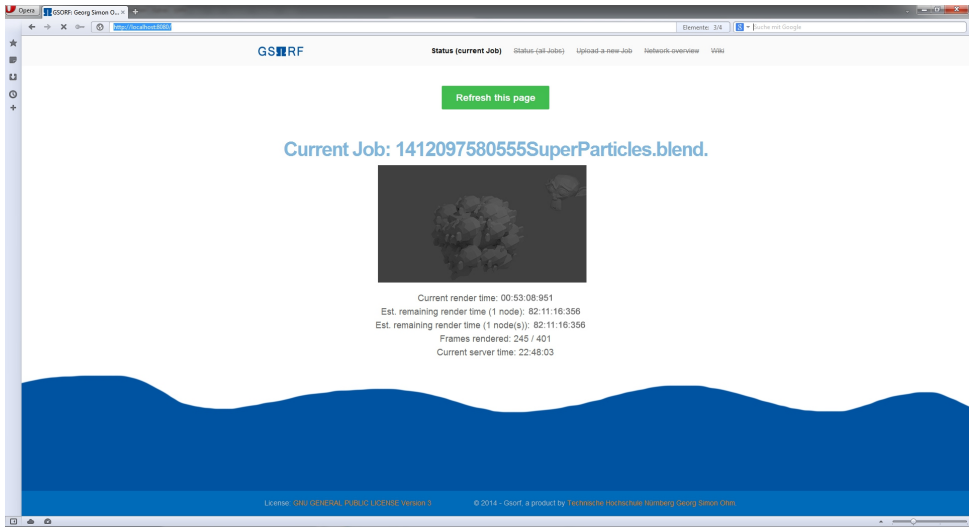
To define a new render job, drag and drop your .blend file in the server. The name of your file is now displayed in the window top. Next, set the start and end frame. You can now press „render“ to send the job on the network.



Depending on the complexity of your scene this may take a while, please keep the server running while the rendering is in progress.

5. Monitoring the rendering progress

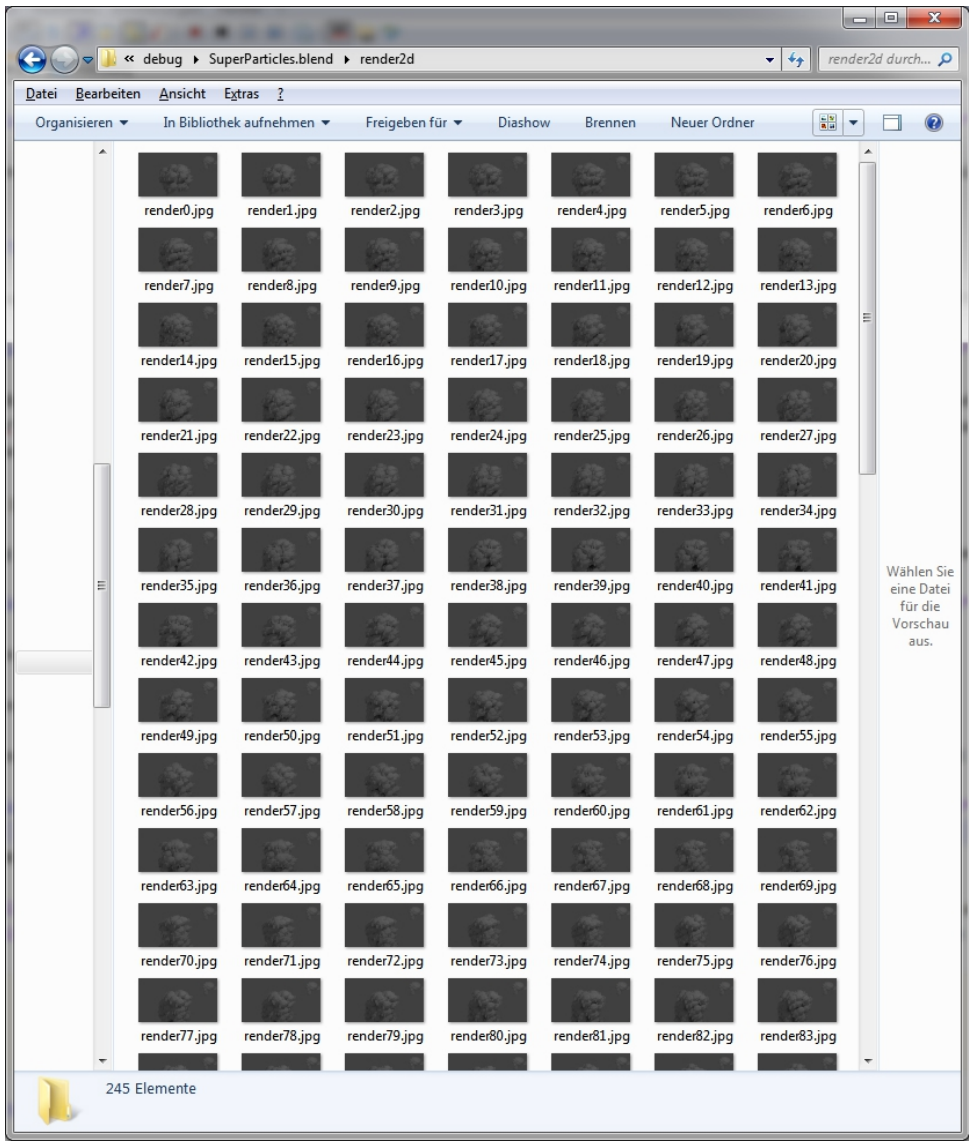
You can monitor the rendering progress in a web browser. Navigate to the url „http://localhost:8080“ to open the GSORF web interface:



As soon as the rendering is complete, you can collect the images.

6. Collecting the finished renderings

The animation frames are stored in a separate directory on the server, which is called according to your .blend file.

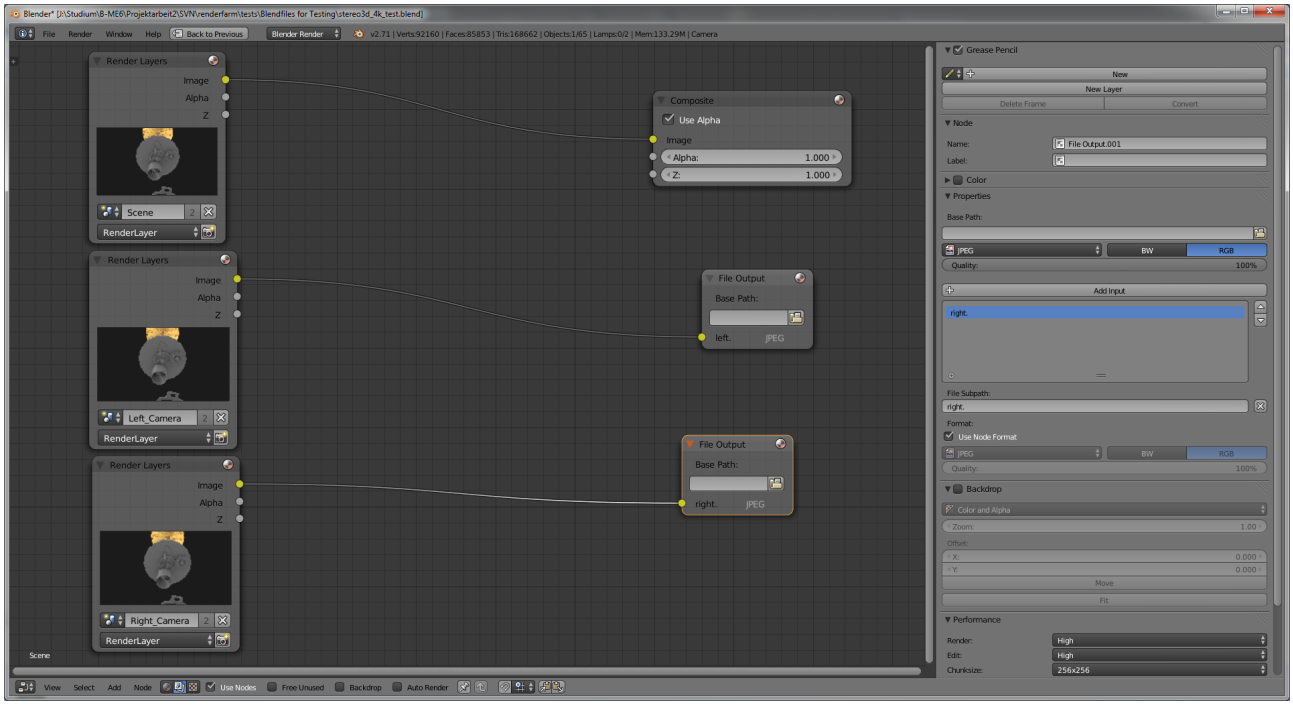


Use Case: Stereo 3D

To render a stereo 3d file, proceed as usual, but pay attention to your .blend scene file.

1. Prepare your .blend file

Creating a stereo 3D file is easy when using the s3d addon for instance. You can download it from http://www.noeol.de/s3d/.
Use it to create a stereo rig and compositing nodes. To make it work with GSORF we need to make some slight adjustments to the compositing nodes:



Connect the RenderLayer node for the main scene to the composite output. The RenderLayer nodes for the left and right scenes need to be connected to separate FileOutput nodes, respectively. Change the file subpath of the FileOutput node from “Image” to “left.” or “right.” (case sensitive) according to their respective RenderLayer.

With this flexible method you are still able to add post effects like e.g. motion blur or depth of field to your renderings and the renderfarm will still be able to recognize the correct eye.

2. Check the stereo 3d option in the GSORF Server

When you have prepared your .blend, you can drop it in the GSORF Server as usual. But before sending it on the network, you need to tick the checkbox “This is a stereo 3d .blend” in the expert mode:

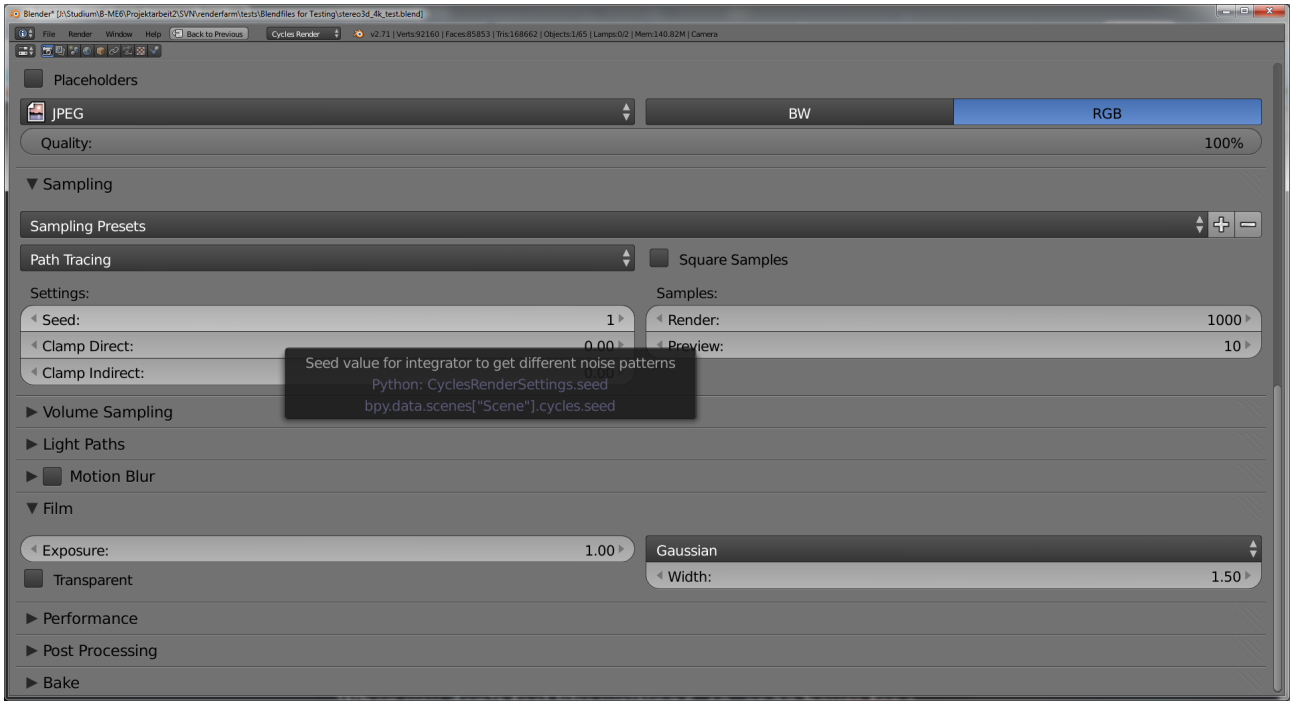
[image]

Use Case: Distributed Tile Rendering with Cycles

1. Prepare your .blend file

Although GSORF doesn't provide distributed tile rendering out of the box, it is still very easy to use it for rendering a single animation frame on several computers. We can use different seed values with lower samples, render them with GSORF and then combine the images later. This method is explained perfectly in this tutorial: https://cgcookie.com/blender/2014/07/31/rendering-single-frame-faster-multiple-computers/

Open your .blend in Blender, change the seed and sampling and save different .blend files. For example if I want to render a scene with 5000 samples on five computers, I'd set the first .blend to 1000 samples with a seed of 1:



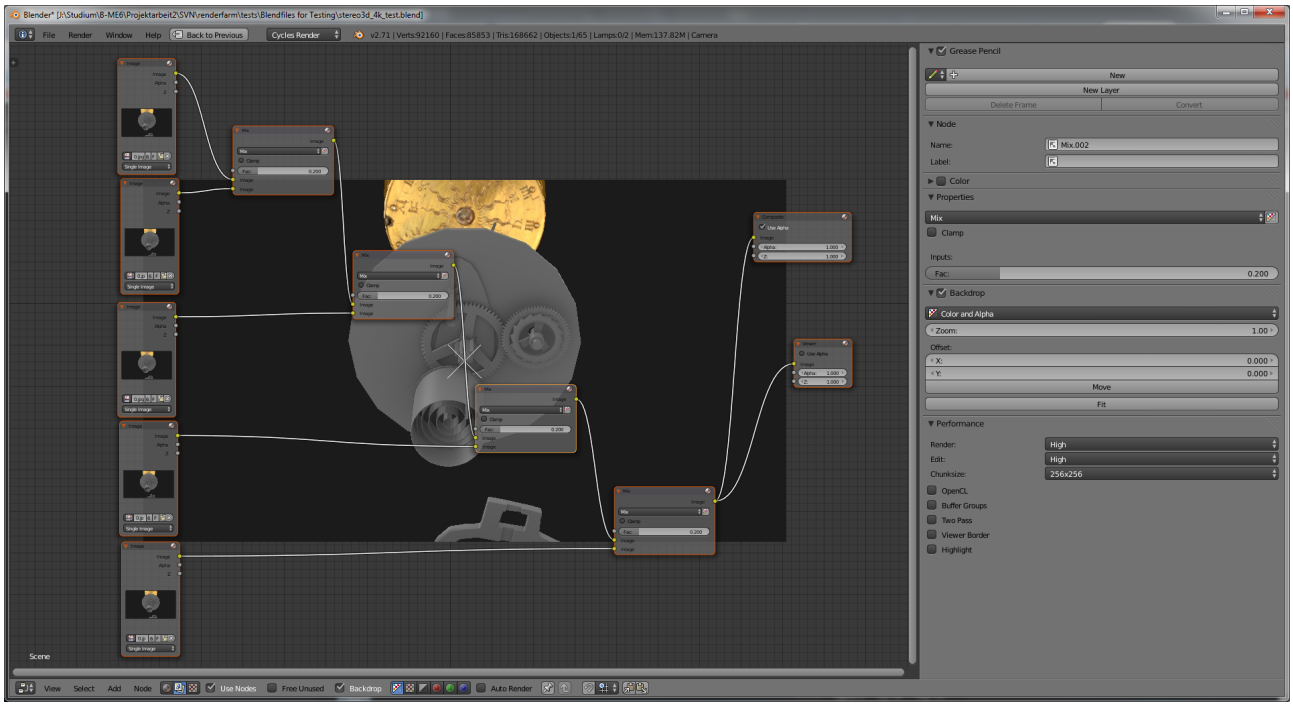
The next file would be set to 1000 samples with a seed of 2 and so on.

2. Send it to the Renderfarm

You now have several .blend files. Let them render one by one with GSORF as usual.

3. Combine the tiles

When finished, take the images and load them up into Blender's compositor (or any other layer based graphic tool). In Blender, connect the image nodes in a way that they get mixed with each other. The mixing factor is simply 1 / (Number of images) – if every image has the same sampling setting. In the end this might look like this:



Developer contact
Adam Kalisz:
Tel: +49 (0)175 7034443
Email: gsorf@lwtv.de

Developer contact
Dominik Rusiecki:
Tel: +49 (0)172 9614 69
Email: rusieckido46966@th-nuernberg.de

Product owner
Prof. Dr. Stefan Röttger

Nuremberg, Germany
© Copyright 2014. Georg-Simon-Ohm Technische Hochschule Nürnberg. All rights reserved.
All trademarks are the property of their respective owners.

GSORF Documentation 1.0v1 AK