# RenderPal V2

The professional Render Management System

**Manual version:** 2.6.1



SHORAN SOFTWARE

# X. Contents

**X. Contents** 2

**0. About this manual** 5

## Part I - Introduction

**1. Installation & Setup** 7
- 1.1 Guidelines 7
- 1.2 Installation 7
- 1.3 Linux and Macintosh installation 8
- 1.4 Configuration basics 8
- 1.5 Updating 9
- 1.6 Advanced installation topics 9

**2. Fundamentals** 12
- 2.1 Render farm basics 12
- 2.2 The three components of RenderPal V2 12
- 2.3 Client Pools 13
- 2.4 User Groups & Accounts 13
- 2.5 Renderers 13
- 2.6 Render Sets 14
- 2.7 Net Jobs 14
- 2.8 Path Maps 14

**3. Workflow** 15
- 3.1 Net job creation I: Filling out the render set 15
- 3.2 Net job creation II: Creating the net job 16
- 3.3 Monitoring and controlling net jobs and clients 17
- 3.4 Other things to do 17

**4. The renderer system** 18
- 4.1 The basics 18
- 4.2 The renderer editor 19
  - 4.2.1 General 20
  - 4.2.2 Groups 20
  - 4.2.3 Renderers and versions 21
  - 4.2.4 Script templates 29
    - 4.2.4.1 Template variables 29
    - 4.2.4.2 Template blocks 30
  - 4.2.5 Renderer API 32
    - 4.2.5.1 Callback functions 32
    - 4.2.5.2 Object types 35
    - 4.2.5.3 Functions 36
- 4.3 Renderer license management 38
  - 4.3.1 License settings 39

## Part II - Features

**5. General** 41

**6. Clients** 42
- 6.1 Parallel rendering 42
- 6.2 The rendering process 42
- 6.3 System control commands 43

**7. Client pools** 44
- 7.1 Dispatch modes 44
- 7.2 Idle client shutdown 44

**8. User groups and accounts** **45**
8.1 User rights 45
**9. Client management** **47**
9.1 Client pool settings 47
9.2 Client settings 48
9.3 Import clients 48
**10. User management** **49**
10.1 User group/account settings 49
**11. The server tab** **50**
11.1 The client pool list 50
11.2 The net job list 51
11.3 The net job chunk list 51
11.4 View filters 51
**12. The render set tab** **53**
12.1 The renderer selector 53
12.2 The render settings list 54
12.3 The file and directory settings 54
12.4 The scene file list 54
12.5 Dynamic scene and output file names 54
**13. The net job editor** **55**
13.1 Main settings 55
13.1.1 Frame splitting 56
13.1.2 Image slicing 56
13.1.3 Additional splitting 57
13.2 Advanced settings 57
13.2.1 Automatic frame checking 58
13.3 Net job events 59
13.3.1 Event variables 59
13.3.2 Program events 60
13.3.3 System command events 60
13.3.4 Python script events 61
13.4 Notes & Tags 61
13.5 Net job presets 62
**14. Renderer management** **63**
**15. Manual frame checking** **64**
**16. Path map management** **65**
16.1 Path map entries 65
**17. Update management** **66**
**18. Autostart** **67**
18.1 Network shares 67
**19. Logging** **68**
**20. Options** **69**
20.1 General options 69
20.1.1 General settings 69
20.1.2 Miscellaneous 70
20.2 Server options 71
20.2.1 Basic settings 71
20.2.2 Advanced settings 72
20.2.5 VNC integration settings 75
20.2.6 Client management settings 75
20.2.7 IP cache 76
20.3 Remote Controller options 77
20.4 Client options 77

20.4.1 Renderer configuration 78

**21. Remote client configuration 80**
21.1 Client settings 80
21.2 Renderer settings 81

**22. Miscellaneous 83**
22.1 The scheduling dialog 83
22.2 The render to print calculator 83
22.3 External tools 83

**23. The console client 84**
23.1 Configuration 84
23.2 Autostart 85
23.3 Command-line options 85

**24. The console remote controller 86**
24.1 Configuration 86
24.2 Command-line options 86
24.2.1 Advanced scene filename syntax 90
24.2.2 Default values 90
24.2.3 Net job presets 91
24.2.4 Controlling net jobs, net job chunks and client pools 91
24.2.5 Data queries 92

## Part III - Advanced Topics

**25. Dynamic scene and output file names 95**
25.1 List parameters 95
25.2 A common example 96

**26. User-created updates 99**
26.1 Update description file reference 100
26.2 Update script API 101

**27. Net job event examples 103**
27.1 Program example 103
27.2 System command example 103
27.3 Python script example 103
27.4 Further ideas 104

**28. The RenderPal V2 database 105**

**29. Command-line switches 106**

**30. Renderer instructions 108**
30.1 After Effects 108

**31. Using RenderPal V2 with Virtualization and Wine 109**

**32. Tips & Tricks 110**
32.1 General 110
32.2 RenderPal environment variables file 110

**Z. Appendix 111**

# 0. About this manual

This manual will explain all aspects of RenderPal V2, from its very basics to more advanced topics. The easiest and fastest way to get used to RenderPal V2 is by simply “playing around“ - you‘ll be surprised how easy and intuitive to use RenderPal V2 is.

This manual does not cover any renderer or compositing application related topics. You should be familiar with the tools you use. RenderPal V2 also requires a certain amount of general computer knowledge; a basic understanding of networking is also necessary.

The first chapter will give you detailed information about the installation and setup process of RenderPal V2 - first things first. The following chapters will deal with the various features RenderPal V2 has to offer, as well as showing some general workflow guidelines.

When reading this manual, it‘s best to have RenderPal V2 running (so you can directly check out what we are talking about).

For any suggestions, feel free to contact us; contact information can be found on the last page of this manual and on our website at http://www.renderpal.com.

**Note:** Screenshots do not always reflect the most recent version.

# Part I - Introduction

The first part of this manual covers all the basics about RenderPal V2. Its installation will be explained, as well as its general usage and workflow. It is highly recommended to read it thoroughly - most questions that arise when starting to use RenderPal V2 will be answered here.

# 1. Installation & Setup

The first thing to do when using RenderPal V2 is to, you guessed it, install and setup everything. The entire installation, setup and configuration process is easy, fast and straightforward - even for large render farms. Due to the automatic update deployment, the installation is an one-time process: once installed on a machine, RenderPal V2 will take care of everything. In the final part of this chapter, we will give some useful information on how to install and setup the RenderPal V2 Client via command-line/batch files; this is especially useful for larger render farms.

## 1.1 Guidelines

You should always install the RenderPal V2 Server first. This is the heart of RenderPal V2, and both the clients and the remote controllers require a running server to operate. Once up and running, you should configure it to your needs (this will be explained in a later chapter). We also recommend to create the pool structure before installing any clients.
Next, the clients should follow. The RenderPal V2 Client supports so-called heartbeats, which allow the clients to inform the server about their existence. This way, you won't have to add the clients manually. Clients found via heartbeats can also be automatically assigned to a pool.
Last but not least, the remote controllers should be installed. In most cases, the server administrator will create various user groups and accounts before or after this step.

Your farm is ready to start rendering! We will now explain how exactly the above steps look like.

## 1.2 Installation

The installation differs for the various components and operating systems. All Windows components come as a graphical setup; the Linux and Macintosh components come in the form of archive files. Please note that we will not cover the various configuration topics here; they will be covered later.

### RenderPal V2 Server

The RenderPal V2 Server, which is only available for Windows, can be installed using the RenderPal V2 Setup. The installation is straightforward and shouldn't need any further explanation.

#### *Autostart*

The server can be configured to be automatically started on system startup. This will be covered in a later chapter.

### RenderPal V2 Client

The RenderPal V2 Client is available for Windows, Linux and Macintosh. The RenderPal V2 Setup can be used to install the client on Windows; this process is even easier than installing the server.

**Usage Tip:** Copying clients

The easiest way to deploy the RenderPal V2 Client across your network is to do a full installation and configuration on one system and just copy the entire RenderPal V2 Client directory to all other systems.

#### *Autostart*

The client can be configured to be automatically started on system startup. This will be covered in a later chapter.

### RenderPal V2 Remote Controller

The RenderPal V2 Remote Controller is available for Windows; a "lite", console based version is also available for Linux and Macintosh. The installation process for Windows is just the same as for the server and client... you already know that it is just plain easy.

### RenderPal V2 and virtualization

RenderPal V2 works very well when used with virtual machines. This means, for example, that you can run the RenderPal V2 Server on a Macintosh as well - just use a virtualization software (like VirtualBox or VMWare), create a virtual Windows machine and install the server on it.

## 1.3 Linux and Macintosh installation

For Linux and Macintosh, both the client and the remote controller come in the form of archives. Simply extract the corresponding archive to a location of your desire. The Linux/Macintosh client and remote controller are a non-graphical applications, so it requires you to tweak some text-based configuration files. Do not worry though, as there are only very few settings you have to change. The next section will describe this in detail.

The client can then be started using the file “rpclientcmd” located in the client directory; the remote controller can be started using the file ”rprccmd” located in the remote controller directory. It is also possible to set up the client to be automatically started during system startup. This will be described in a later chapter.

**See also:** Console client configuration, Console client autostart, Console remote controller configuration

## 1.4 Configuration basics

This section only briefly covers the most basic configuration tasks you have to do to get everything running. All default settings have been carefully chosen and should be sufficient in most cases. However, there are some settings that always need to be made. This is just a guideline on what to do first when configuring everything; in-depth information will be given later.

### RenderPal V2 Server

The server is very easy to configure, and there are only a few tasks that should be done in the beginning:

- Create your client pools (groups of clients)
- Create your user groups and accounts (used by the various remote controllers)
- Edit your path maps (usually only when you have a mixed operating system environment)
- Change the various settings to your needs (like automatic e-mail notifications, client management settings...)
- Configure the server to be automatically started when Windows starts

All of the above tasks are actually optional - RenderPal V2 Server comes with a default configuration that is fully operational (but rather generic). Client pools and user groups/accounts represent the “skeleton” of your render farm, so it is always a good idea to create them first.

### RenderPal V2 Client

Clients should always be configured using either the RenderPal V2 Server or Remote Controller. This allows for batch configuration of all your clients, which is just way more convenient than configuring every single client. While most defaults should be just right for most users, you’ll need to configure the various renderers the clients should use.

#### *Configuring the heartbeat feature*

Heartbeats can be sent from the clients to the RenderPal V2 Server to inform the server about their existence. To get this feature to work, you’ll have to configure it first, though. There are two ways to do so. You can either configure the clients by hand, or use the command-line switch **/heartbeat** (e.g. `/heartbeat ”192.168.0.100”`); this switch will automatically activate heartbeats and set the RenderPal V2 Server address.

#### *Configuring renderers*

The clients need to know the executables of the renderers you are going to use. By default, most renderers will provide useful information about which executable files to use.

#### *Console clients*

All console clients can be configured remotely as well. However, you might want to edit the heartbeat options by hand, so that the clients will be automatically be added to the server. To do so, open the file **RpClientCmd.conf** (located in the client directory) with a text editor of your choice. All settings in this file are commented. Just search for the section **[Client.Heartbeat]** and edit the settings to your needs.

### RenderPal V2 Remote Controller

This is actually the easiest component of RenderPal V2 to configure. The only thing that you have to set is the server's address and your user credentials.

***Console remote controllers***

In order to use the console remote controller, you'll have to configure it first. To do so, open the file **RpRcCmd.conf** in a text editor of your choice. All settings in this file are commented. The section **[RemoteController.Server]** contains all settings you'll need to change. The file **RpRcDefaults.conf** can also be edited to set default values for the various command-line switches of the console remote controller; this topic will be covered in a later chapter.

## 1.5 Updating

As mentioned before, the installation of RenderPal V2 is a one-time process. This means that manual updating or maintenance will not be necessary once the installation has been done. While it is most certainly a lot of fun to update 250+ clients by hand, RenderPal V2 offers a powerful automatic update deployment system. Simply put, all you have to do is to add a new update to the server and RenderPal V2 does the rest for you. All clients and remote controllers will be automatically updated - no matter on what operating system they are running.

## 1.6 Advanced installation topics

### Silent setups

The Windows Setup allows you to create and use so-called silent setups. These are command-line driven setups that require no user interaction. They can become very useful when installing many clients on large networks. Using silent setups is a two step process:

**1. Create a response file.** In order to run the setup in silent mode, you must first run the setup executable with the /r command-line switch to create a response file, which stores information about the data entered and options selected by the user at run time. You must also specify the response file to create using the /f1 command-line switch and specifying a fully qualified filename. Example:

```
RenderPalV2Setup.exe /r /f1"C:\Setup.iss"
```

**2. Run the setup in silent mode.** Once the response file has been created, you can run the setup in silent mode using the /s command-line switch. The setup will then be executed with the data entered and options selected during step 1. The response file has to be specified using the /f1 command-line switch. Example:

```
RenderPalV2Setup.exe /s /f1"C:\Setup.iss"
```

Note that the syntax has to be exactly as shown above (no space between /f1 and the filename, the filename must be fully qualified and should be surrounded with quotes).

When performing a silent setup for the RenderPal V2 Client, a default configuration will be created which also automatically searches for any installed renderers. When starting the client for the first time, the configuration dialog will not be shown.

### Copying the client directory

RenderPal V2 stores all configuration files inside its program folder. This way, you can just copy an existing client to a new machine and it will work right away (though you might need to modify some settings to fit the new computer). If you want to run two clients on the same machine, you also have to create a copy of the existing client directory and change the client's listening port.

There are no drawbacks to this method. All features, including automatic updating, will work without any problems. It is a very convenient and time-saving way to deploy clients on large networks. We recommend to have the computer name placeholder (`%COMPUTERNAME%`) in the client alias, so that you can still distinguish the copied clients. This is also the default setting, so you should just leave it the way it is.

## Useful command-line switches

There are a few command-line switches that are especially helpful when it comes to command-line setups (often in combination with an installation batch file).

***/service:install***

This switch (Windows only) allows you to install the RenderPal V2 Client as a service, so that it will be started during Windows startup. The syntax of this switch is as follows:

```
/service:install(<user>,<pwd>,<dep>,<autorestart>,<scanshares>)
/service:install(Tak,MyPassword,,yes,yes)
```

`<user>` is the user name/account to run this service under, `<pwd>` is the user's password and `<dep>` states the name of a service the RenderPal V2 service should be started after. `<autorestart>` determines whether the application should be automatically restarted if it has been terminated (set to **yes** or **no**). If `<scanshares>` is set to **yes**, RenderPal V2 will scan your existing network shares and will create the necessary mappings. All values are optional. More details about the RenderPal V2 service will follow in a later chapter.

***/heartbeat***

This switch allows you to set the address of the RenderPal V2 Server to use for heartbeat sending. The syntax is as follows:

```
/heartbeat:<server>
/heartbeat:"192.168.0.100"
```

`<server>` denotes the address of the RenderPal V2 Server. You can also specify a port (using the `address:port` syntax) if it differs from the default one.

***/silent***

This switch will suppress any message boxes during execution of other command-line switches. Should be specified for any batch file/silent installations.

***/quit***

Forces the executable to quit after execution of other command-line switches. Should be specified when performing installation related tasks.

## Bringing it all together

Silent setups, simply copying the client directory and the various command-line switches are powerful tools, especially when used in a batch file. There are many ways to combine these features, but here is one example of how such a batch file might look like:

**Code:** Batch file example

```
REM Windows Installation Batch file example

REM Step 1
xcopy "\\MyServer\RenderPal V2 Client" "C:\RpClient" /E

REM Step 2
copy "\\MyServer\RpConfigs\New\RenderPalCl.cfg" "C:\RpClient\Config"

REM Step 3
"C:\RpClient\RenderPalCl.exe" /service:"install(Tak,MyPassword,,yes)" /silent /quit

REM Step 4
"C:\RpClient\RenderPalCl.exe" /heartbeat:"192.168.0.100" /silent /quit

REM Done! Note that step 3 and 4 could also be combined into one single call.
```

1. Copy an existing client directory (from a network path) to a local disk or perform a silent setup
2. Optionally copy a different configuration file
3. Install the client as a service using **/service:install** (Windows only) or use the autostart daemon scripts for Linux/

Macintosh

4. Set the RenderPal V2 Server address using **/heartbeat**

With this simple batch file, you can install the client, set up the service and configure the heartbeat address with a single click.

# 2. Fundamentals

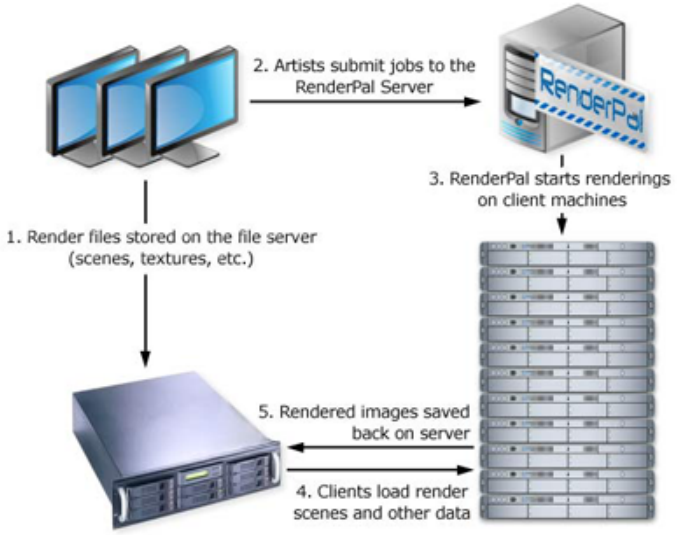
Before we get into the various features of RenderPal V2, we will give you an overview of its fundamental building blocks. Understanding the fundamentals is key to using RenderPal V2 efficiently. The following topics will be covered:

- Render farm basics
- The three parts of RenderPal V2
- Client Pools
- User Groups & Accounts
- Renderers
- Render Sets
- Net Jobs
- Path Maps

This chapter is all about the question ”What are ...?”. The following sections will describe what the above components are, what function they serve and how everything works together.

## 2.1 Render farm basics

The following diagram shows the basic setup of a render farm:




A render farm usually consists of a file server, where all scenes and scene-related files (textures etc.) reside on, the render management server running RenderPal V2, the artists’ workstations, as well as various rendering machines. The workflow of a render farm is pictured in the diagram, but there are still a few tips that can help you when setting up your farm:

- It is highly recommended to have a dedicated file server running a server OS
- The RenderPal V2 Server can run on the file server or on a separate machine
- The artists’ workstations can also be used for rendering when they are idle
- No scene-related files should ever reside on the workstations
- RenderPal V2 can shut down unused machines to save energy; it can also wake up computers when needed
- The RenderPal V2 Remote Controller can also be used over the Internet

## 2.2 The three components of RenderPal V2

RenderPal V2 consists of three individual programs: the server, the client and the remote controller. While the server and the client are crucial for RenderPal V2 to operate, the remote controller is not essential for RenderPal V2 to run and can be considered an ”optional” component (but you won’t want to miss it).

### RenderPal V2 Server

The server is the heart of RenderPal V2. It's main tasks are to manage all clients and logged in users, distribute the various net jobs across the network, and to keep an eye on everything. It also keeps all clients and remote controllers up-to-date via its automatic update deployment.

### RenderPal V2 Client

The client is the workhorse of RenderPal V2. It receives its jobs from the server and executes (renders) them. It also keeps track of the (textual) output of every job it has rendered; this output can also be viewed from within the server or remote controller.

### RenderPal V2 Remote Controller

The remote controller is used to remotely control the RenderPal V2 Server (hence the name). It offers almost all features found in the server, except for the update management and server related settings. The features available to a logged in user can be restricted via user permissions in the user account management.

All of the above components have a lot to offer, but this briefly shows you what their main tasks are. During the following chapters, you will learn all details about their various features.

## 2.3 Client Pools

A client pool represents a group of clients. For a client to do some work, it has to be assigned to a pool; a client can be assigned to multiple pools at once. When creating net jobs, you will also select which pools should work on the job. A client pool can be seen as a separate, encapsulated working unit. Each pool will be processed independently, one after another; this is important when it comes to job dispatching (see below). Pools also offer other features including scheduling and automatic client shutdown.

### Job dispatching

One of the main tasks of a client pool is to dispatch its assigned net jobs to the clients. Without going into details for the moment, it is important to know that one pool cannot ”see” any other pools that might also work on a job - each pool is encapsulated. This has one important consequence: when dispatching a job across multiple pools, the clients assigned to the pools will **not** be grouped together. This means the order of clients is not necessarily ”priority pure” (if pool A contains high priority clients, but pool B only contains low priority clients, both A and B will still alternately dispatch a job).

## 2.4 User Groups & Accounts

When using the remote controller, one has to specify a login for the server. These user accounts are created using the user account management. A group/account specifies the permissions for an user; it also restricts the account to certain client pools. This allows for a very detailed user management for your render farm: create guest accounts that can only ”view but not touch”, normal users or administrators who can modify the client pools or even create new user groups and accounts.

## 2.5 Renderers

Renderers (note that ”renderers” refers to ”true” renderers as well as compositing tools and that alike) build one of the core components of RenderPal V2 - what would a render management system be without support for renderers anyway? A renderer consists basically of various parameters which can be set by the user in a render set (see below), rules that define how RenderPal V2 should compile a command-line from these, how it should start the renderer and so on.

RenderPal V2 comes with support for quite a lot of renderers ”out-of-the-box”, but it is also possible to create your own renderers as well; you can also edit these renderers to your own needs. This will all be covered in a later chapter.

The renderer system of RenderPal V2 is highly advanced and very rich, though it is still very easy to use, as you will see.

## 2.6 Render Sets

A render set is a collection of one or more scene files (or any other type of source file), along with various render settings (like frame ranges, image name and so on). It builds the base for any net job you create. Each render set contains at least one scene file, but multiple scenes in a single set are supported as well. The settings made in a render set will be applied to all scenes in the set. If you need to make different settings for a scene, you have to create a new render set. When multiple scene files are added to a render set and an output filename is specified, a special renaming rule will also be applied (the scene name will be added to the output filename).
The render settings available depend on the selected renderer; all settings are **overrides** and may be left out. In this case, RenderPal V2 will use the settings made in the scene. Some features of RenderPal V2 require certain settings to be filled out; this includes the frame range(s) for frame splitting and the image dimensions for image slicing.

## 2.7 Net Jobs

Net jobs represent the various jobs that will be distributed and rendered across your render farm. A net job consists of two parts: the render set on which the net job is based and the numerous settings which control how the job is divided into smaller pieces (called net job chunks), the renderer to use, net job events and more. Net jobs offer plenty of features, which will be covered in full detail in a later chapter.

### Net job chunks

RenderPal V2 divides a net job into smaller chunks, which are then sent to the clients for processing. The count and size of these chunks is determined by the frame splitting and image slicing settings. Frame splitting allows you to divide an animation into packets of smaller frame ranges; image slicing allows you to cut up a large image into smaller pieces.

A net job chunk can have one of three priorities: low, normal and high. Chunks with high priority will be rendered before chunks with normal priority, which itself will be rendered before chunks with low priority.

**Important:** Frame splitting, image slicing and additional splitting

Without the use of frame splitting, image slicing or additional splitting, the job will only have one single piece (the entire scene); this will barely be what you want. In order to be picked up by multiple clients at the same time, you'll need to apply frame splitting, image slicing and/or additional splitting.

## 2.8 Path Maps

Path maps are used to translate paths and filenames to something else. Path maps can be defined globally, for certain client pools or for specific clients; they will be applied to all paths and files inside a render set and net job (including net job events). Path maps are especially useful for render farms with multiple operating systems, but can be used in other situations as well. A path map entry consists of an in-path and an out-path; the path to translate will be parsed for the in-path, which will, when found, be replaced by the out-path. Path map entries can also be made specific to a certain OS, which makes managing a render farm with multiple operating systems even easier.

**Important:** Paths and filenames inside scenes

Paths and filenames inside a scene will **not** be translated (e.g. textures). Path maps only affect paths set inside RenderPal V2.

**See also:** Path map management

# 3. Workflow

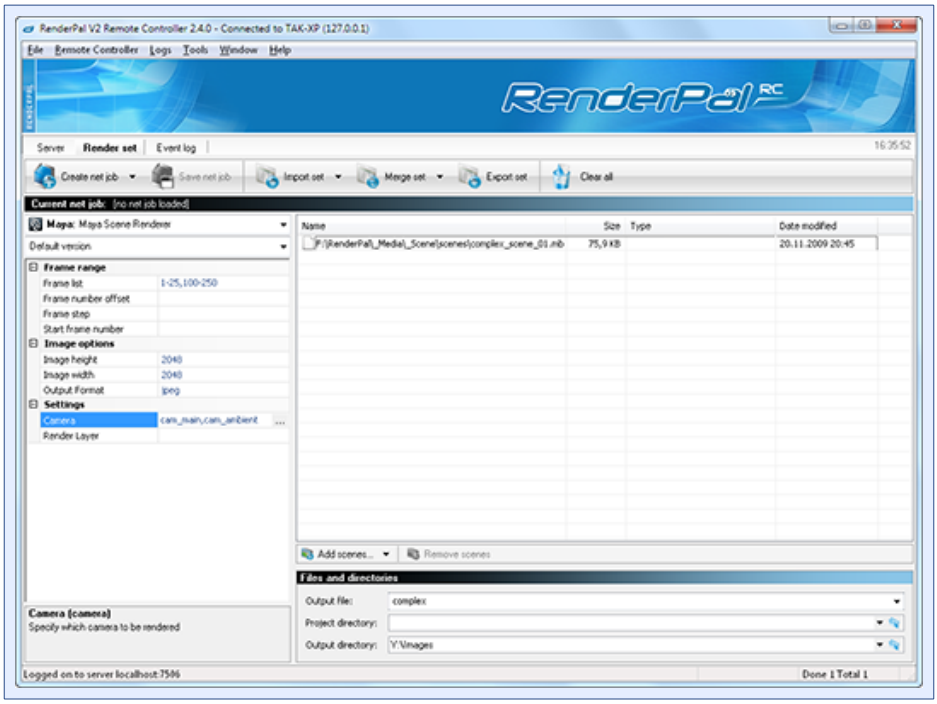
Once RenderPal V2 has been installed and configured, the most common tasks are the creation and monitoring of net jobs and their chunks, as well as looking after your various render clients. This section describes the general workflow of RenderPal V2. The tasks described in this chapter will all be shown using the remote controller, but the steps are exactly the same for the server. Details about the various features presented here will follow in later chapters.

The following tasks will be covered:

- Net job creation I: Filling out the render set
- Net job creation II: Creating the net job
- Monitoring and controlling net jobs and clients
- Other things to do

## 3.1 Net job creation I: Filling out the render set

Every net job is based on a render set, so the first step is to fill out one. The following screenshot shows the Render set tab:




The first thing you should do is to select the renderer you want to use. This will also determine the available render settings, as well as the file filter when adding new scene files - which is the second step. Once you have added the scenes you want to render (you will most likely have one scene in each set), it is time to fill out the various render settings (remember that all settings are **overrides**). The frame list, as well as the image dimensions are the most common ones and should always be filled out. Filling out the output file and output directory is also recommended (the output directory can also be translated using path maps). When you have finished filling out the render set, click Create net job to start the actual net job creation.

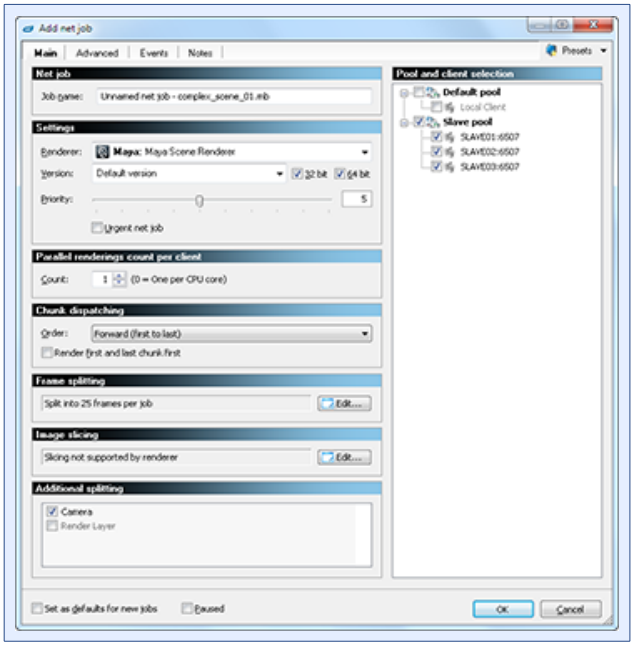
You've probably noticed the Save net job button already; this is used when editing the render set of an existing net job. If you want to make any changes to an existing job, don't forget to save the job afterwards.

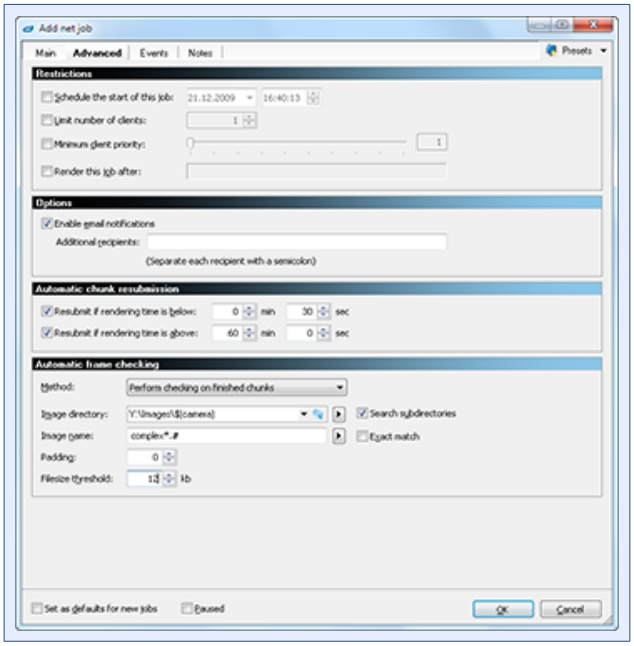
All settings filled out will be preserved on exit; to create a render set from scratch, click Clear all first. It's also possible to export (and later import) a render set to a file. This can become handy if you want to create presets for your renderings.

**See also:** The render set tab

## 3.2 Net job creation II: Creating the net job

After filling out the render set, you can create the actual net job. The following screenshots show the net job dialog Main and Advanced tabs:




The first thing to fill out is the net job name; the default name pattern can also be changed in the RenderPal V2 options. The priority determines how ”important” this job is, and how high the job in the processing queue should be. Urgent jobs will be prioritized above all other (non-urgent) jobs.

**Important:** Urgent net jobs

When setting a net job to urgent, clients (except those that are already rendering urgent net jobs) will drop their current job in favour for the new urgent net job, so be careful when submitting urgent jobs.

You can furthermore change the renderer to be used, as well as the pools and clients that should work on this job. As mentioned earlier, a net job without any applied frame splitting, image slicing or additional splitting will only be picked up by one client, so you should always activate at least one of these options. Other useful options include the ability to automatically resubmit jobs that are either taking too long or were finished too quickly, the automatic search for missing frames (called **Frame checking**) and net job events. These and all other options will be covered later.

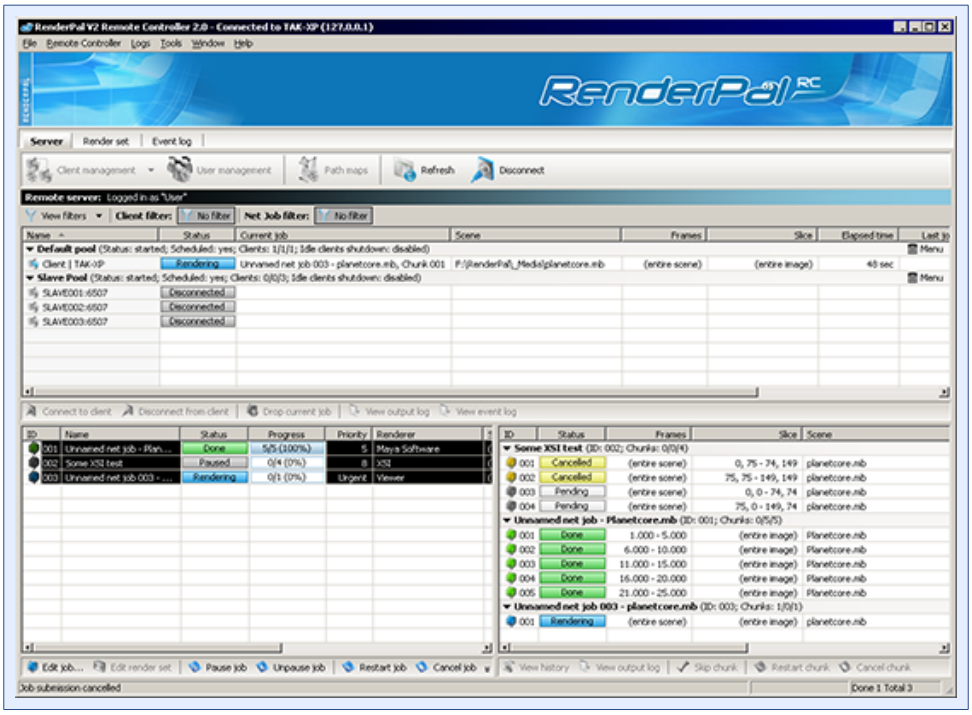
All settings can be changed at a later time. If certain settings, like the renderer, frame splitting, image slicing or additional splitting, are changed, the job might need to be restarted. A warning will be displayed in this case.

Net job presets allow you to save various settings for later use; they can also be used as the base for newly created net jobs. They will be covered in greater detail later. If you just want to make the current net job settings the default ones (which will be used for any newly created net job), simply check the Set as defaults for new net jobs box before creating the job.

**See also:** The net job editor

## 3.3 Monitoring and controlling net jobs and clients

Besides the creation of net jobs, one of your main tasks will be to monitor the various net jobs and clients in your render farm. This is done in the server tab (or often referred to as the **main view** of RenderPal V2), which is shown in the screenshot below. The main view consists of the client pool list, the net job list, as well as the net job chunk list (which can be hidden).



### Client pool list

The client pool list shows you all client pools along with their assigned clients. The list contains information about the clients' status, details about their current jobs and so on. You can control the clients and pools in various ways, as well as remotely take a look at the output and event logs of the clients.

### Net job list

This list shows all the net jobs in your render farm. It includes several details about the various jobs, like their status, the submitter and their progress. Controlling net jobs is also possible from within this list. When selecting one or more jobs, their individual chunks will be shown in the net job chunk list.

### Net job chunk list

This list shows the individual chunks of all selected net jobs. Besides controlling the chunks, it is also possible to retrieve the (textual) output for each chunk, or to view a small history (which client rendered the chunk, which clients failed in doing so and so on).

**Important:** User restrictions

Remote controller users can be restricted to certain client pools, which will not be visible to them. The same applies to net jobs. The ability to control pools, clients and net jobs can also be limited.

**See also:** The server tab

## 3.4 Other things to do

While the main tasks in managing a render farm are the creation of net jobs and the monitoring of everything, there are some other tasks that have to be done from time to time. This includes adding new updates, changing some client configurations, maintaining the pools and users or updating the path maps.

# 4. The renderer system

Since renderers are such a fundamental feature of RenderPal V2, this chapter is dedicated to the renderer system, how to use its editor (for creating your own renderers or editing existing ones) and so on. The first part will cover the basic concepts behind the renderer system, while the second part will cover the built-in renderer editor; note that you will only come into touch with the editor when creating or editing renderers (if you are just beginning to use RenderPal V2, we recommend you to skip the second part for now and come back here later).

**Note:** When we refer to ”renderers”, we usually mean the interface to the renderers, not the renderers themselves; when we refer to ”configuring renderers” in this manual, this usually refers to setting up the executables paths for renderers.

## 4.1 The basics

Renderers in RenderPal V2 are ordered hierarchically into **groups**, **renderers** and (renderer) **versions**:

### Groups

A group is the highest entity in this hierarchy, and all renderers are ordered into groups; it is solely an informative and organizational unit. A group usually represents a set of similar renderers (a single renderer might support various plugin renderers, for example, and these renderers could all be put into a single group).

### Renderers

The actual renderers form the core of the renderer system. A renderer defines the various parameters the user can set, how the command-line should be compiled, it can filter the textual output of the renderer (and react accordingly) and it can even extend the default behavior (which is to compile a command-line according to the define command-line switches and launch the renderer) using a Python script.

### Versions

Every renderer can have various versions. A version offers basically the same options as a renderer and further extends its parent renderer. Versions can be used to further refine an existing renderer, which is especially useful when a new renderer version (of the actual renderer, not the RenderPal V2 renderer) comes out which supports new features and you want to have support for both the new version and the old one. The name ”version” should be seen rather loose, as a renderer version doesn’t necessarily need to represent a different renderer version but can also be used in any other way you want.

To sum things up, all renderers are sorted into various groups, and every renderer can have multiple versions. When using a renderer, you will actually always use a specific renderer version, and a renderer needs at least one version (in many cases, a renderer will only have a single, ”one fits all” default version). When a renderer has more than one version, one of these is set as the default version, which will be automatically used when selecting that renderer.

The number of renderers can become quite large, so every group, renderer and version can be set to invisible; renderers you don’t use can be hidden this way (instead of simply deleting it). The invisible setting will affect the renderer selection (in the render set and net job dialog), as well as renderer configuration. The renderer editor, however, will always show all renderers.

We have talked about a lot about parameters, command-line switches and so on. Even though you won’t have to deal with these things directly when just using renderers, it is still of advantage when at least roughly knowing what they mean. So here is a short rundown of the various components a renderer (or a renderer version) consists of:

### Parameters

The settings a renderer support are called its parameters. Most of these parameters can be edited in the render set, but there are also some parameters which are computed and cannot be edited by the user. You can also think of parameters as values the renderer can use. Parameters (or to be more precise, their values) can then be used in the command-line or the Python script. There are plenty of different parameter types, like strings, numbers, files and so on.

### Command-line switches

Renderers are always launched with a certain command-line which is compiled by RenderPal V2. The command-line switches define how exactly this command-line should look like. Renderers define the various switches, along with their values (usually in the form of a parameter placeholder), which should be put together to form the command-line.

### Executables

Every renderer needs to know which executable file should be launched. While these are usually configured in the clients, it is also possible to define defaults in the renderers and the renderer versions (this is especially handy when all your machines have the same paths for the renderers). Executables can be defined for all operating systems, for both 32 and 64 bit.

### Output filters

Renderers have no other way to report what they are doing (especially if something goes wrong) than printing text to the console. RenderPal V2 can parse this output by using so-called output filters. Such a filter consists of a text to look for and an action to take (report something to the server or cancel/abort the job).

### Return codes

When finished with their work, processes return a numeric value to indicate whether an error occurred or not. This code can then be checked by RenderPal V2 to report whether a job succeeded or not. Note that not every renderer makes use of this feature (they usually just always indicate ”success”).

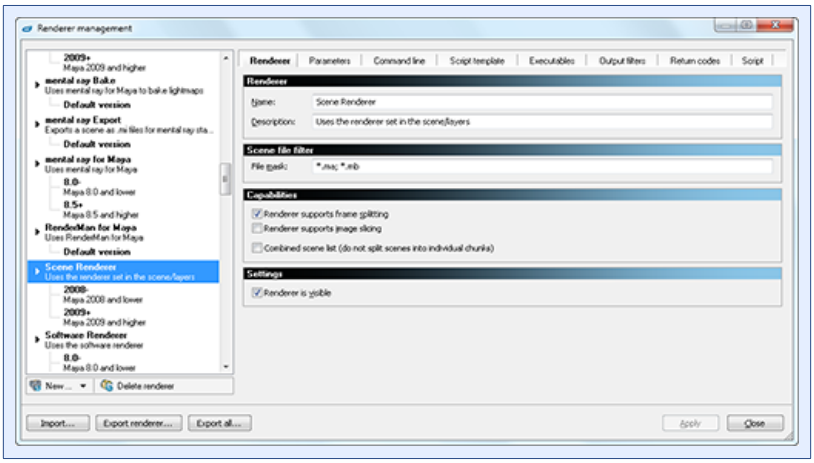
### Python script

By default, RenderPal V2 will build the command-line based on the command-line switches that are configured for the renderer. It will process every switch, one by one, and add it to the resulting command-line (as long as all necessary parameters have been filled out in the render set). It will then start the renderer with that command-line and wait until it has finished. This ”default behavior” might not always be sufficient (especially when using renderers that execute scripts), so a renderer can redefine nearly all aspects of a renderer using Python.

All this might seem like quite a lot of information, and you might even think that the renderer system is rather complicated. Rest assured that it isn’t - it is actually quite easy to use. You won’t have to deal with all these concepts most of the time, they only become important when creating or editing renderers. The renderer system is complex and offers a lot of features, but you will see that working with renderers is very easy and straightforward.

## 4.2 The renderer editor

As you’ve surely already noticed, the renderer system offers a lot of features. Creating and editing renderers is done using a comfortable, easy to use graphical editor - there’s no need to edit any source files by hand. This chapter will show you the renderer editor in detail. Even though the editor is easy to use, creating renderers still is an advanced topic, and you should first get a decent knowledge about RenderPal V2 in general before trying to create or edit renderers. Here is a screenshot of the editor:




Before we start, here is a word about the differences between a renderer and a renderer version: A renderer builds the base and contains most of the settings, while a version can further extend its parent renderer; a version inherits all settings from the renderer and extends those. You can, for example, add additional parameters in a version or modify its command-line. The differences will be described in detail in the following sections.

**Note:** Most aspects of the editor are the same for renderers and versions, so we will only describe them separately when necessary.

### 4.2.1 General

The renderer editor consists of a list of all groups, renderers and versions, as well as several tab pages where all settings can be made. Using the editor is rather straightforward: selecting an item will automatically set it into edit mode, adding new items or deleting existing ones can be done using the toolbar or context menu.

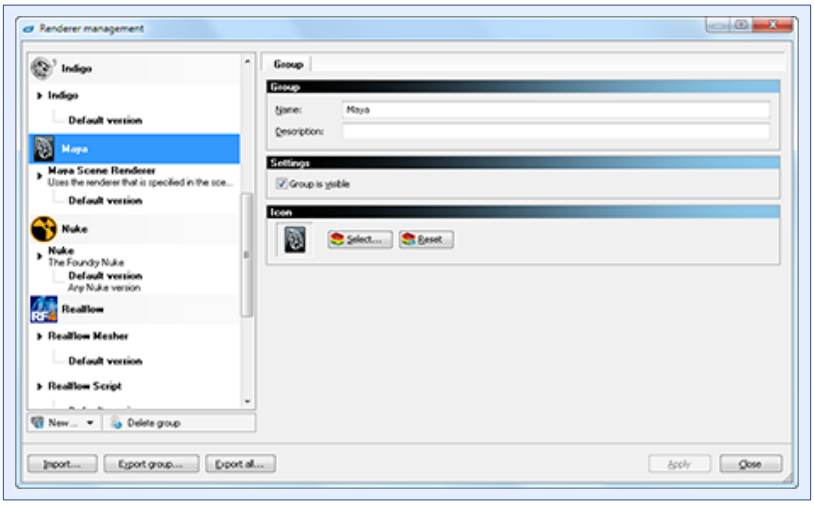
It is also possible to duplicate renderers and versions (using the context menu). When duplicating, you will be asked to give the new renderer/version a new name, as well as to choose a target group and renderer (only for duplicated versions).

Groups, renderers and versions can be exported, as well as the entire renderer pool (using Export all). The data is always exported to a single XML file, which can then be easily exchanged. When importing such a file, an options dialog will appear where you can decide whether existing items should be overwritten or should be renamed, and whether to merge new items with existing ones (if you are importing a renderer that already exists and the existing renderer has versions which aren't present in the imported file, these versions will be kept if merge is enabled; the same applies for groups and their renderers).

**Note:** It is possible to edit renderers which are in use, but be aware that net jobs using that renderer will be restarted (a warning will be displayed, though). Renderers in use can't be deleted. Also note that changes will be immediately sent to the RenderPal V2 Server (if the pool was edited using a Remote Controller) and all Remote Controllers; there is no need to update anything by hand.

### 4.2.2 Groups

Renderers are always ordered into groups. A group has no effect on the renderer, it's sole task is to organize renderers. Groups only have a single tab, the group settings:



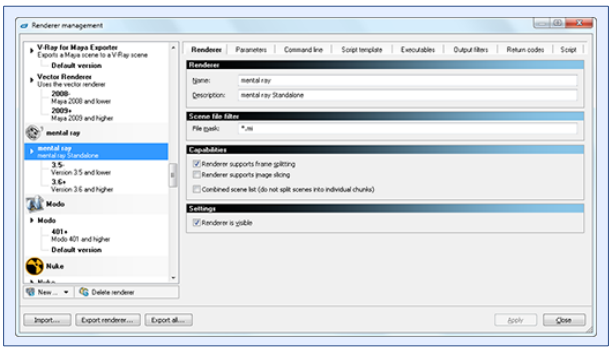

| Name | Description |
|---|---|
| **Group** | |
| Name | A unique name for this group. |
| Description | An optional description for this group. |
| **Settings** | |
| Group is visible | Sets the visibility of this group. If set to invisible, the group won't appear in selections etc. (including all its renderers). |
| **Icon** | |
| Every group can have a representative icon. You can either directly select an icon file (.ico), or choose one from an executable (.exe) or library (.dll); in all cases, the icon will be copied automatically to the correct place, so the original icon doesn't need to be kept. | |

## 4.2.3 Renderers and versions

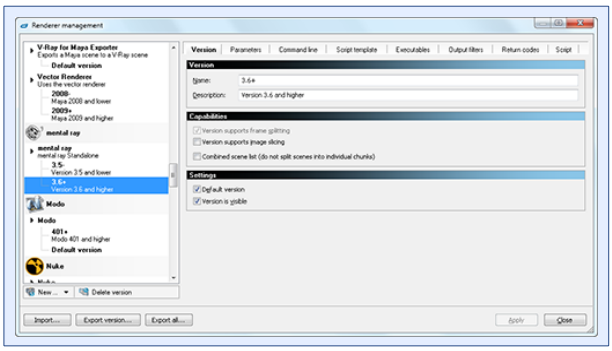
Since renderers and versions have nearly identical settings, we will describe them both in a single chapter. We will describe the differences between them where necessary.

### Renderer/Version settings

The general settings for a renderer/version include its name, description, a file mask, as well as its capabilities. The name of a renderer has to be unique among all renderers (no matter in which group the renderer is), the name of a version has to be unique among all versions of its parent renderer. The file mask (only available for renderers) is used in the ”Open file” dialog when adding scenes in the render set as a file filter. The capabilities determine whether the renderer/version supports frame splitting and image slicing; if the renderer doesn’t support a feature, it can still be enabled in a descendent version, but a feature supported by the renderer can’t be disabled in a version.



Renderer settings



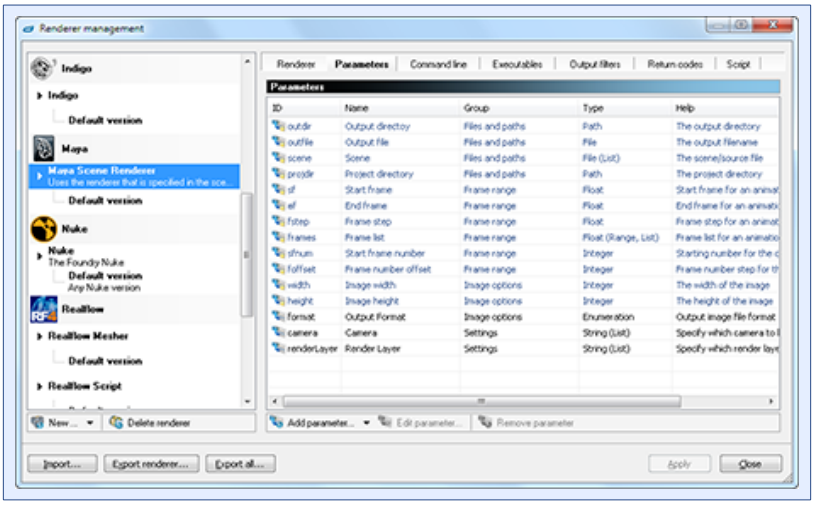
Version settings

| Name | Description |
|---|---|
| **Renderer/Version** | |
| Name | A unique name for this renderer/version. |
| Description | An optional description for this renderer/version. |
| **Scene file filter *** | * Renderer only |
| File mask | Used in the ”Open file” dialog when adding scenes in the render set as a file filter. Enter file types in the form of `*.ext`; separate multiple extensions with a semicolon. |
| **Capabilities** | |
| Renderer/Version supports frame splitting | Determines whether frame splitting is supported. For frame splitting to work, the frame list parameter must be included in the renderer or version. |
| Renderer/Version supports image slicing | Determines whether image slicing is supported. For image slicing to work, the image width and height parameters must be included in the renderer or version, as well as the output file parameter. |
| Combined scene list | Usually, if a render set contains multiple scenes, it will be split into individual chunks for each scene. With this option, this behavior can be disabled, and each chunk will contain all scenes (useful for converters, for example). |
| **Settings** | |
| Renderer/Version is visible | Sets the visibility of this renderer/version. If set to invisible, the renderer/group won’t appear in selections etc. |

### Parameters

Parameters represent the values (or settings) which can be set for a renderer and its versions in a render set. Most of these can be edited by the user, but some are computed and won’t appear in the render set. A parameter consists of its value type (like string or number), name and group, as well as several type-specific options.

Parameters come in two flavors: standard (or built-in) parameters and custom parameters. The standard parameters are parameters which RenderPal V2 uses for certain features, like frame splitting, image slicing and so on. Standard parameters can either be added by using the context menu of the parameters list, or by using the dropdown button next to the Add parameter button in the toolbar. Standard parameters will be shown in blue in the list.

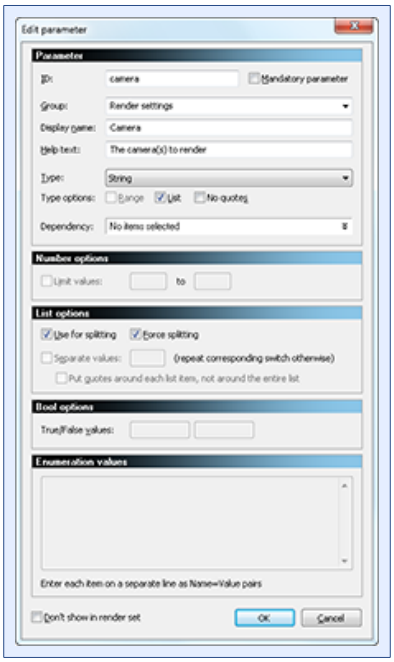



When editing a version, inherited parameters will be shown in gray. Inherited parameters can't be modified, but you can overwrite an inherited parameter by adding a new one with the same ID - parameters in the version will always have priority over inherited ones; it is also possible to disable parameters for certain versions.

**Important:** Frame parameters

While most standard parameters should be self-explaining, the frame parameters are a bit special. There are three frame related parameters: the **frame list**, the **start frame** and the **end frame** (only the frame list can be added manually to the list). The frame list is what the user will be able to edit; this parameter allows him to enter frame ranges to be rendered (like 1-100;200;250-500). However, this parameter isn't useful for command lines, so two more parameters will be generated automatically: the start and end frame, which can then be easily used in a command-line. When adding the frame list parameter, the start and end frame parameters will be automatically added to the parameter list.

There are only a small number of standard parameters; RenderPal V2 gives you full freedom over the parameters your renderer supports and imposes as little constraints as possible on the renderer architecture. When adding or editing a parameter, the following dialog will appear:




A parameter consists of a unique ID (this is also used in the command-line switches and Python script; for standard parameters, this cannot be modified), its group, display name and help text, which are used in the render settings edi-

tor etc. and its type along with several type-specific settings (which can only be partly edited for standard parameters). Parameters can also be hidden from the render set editor (useful for command-line-only parameters).

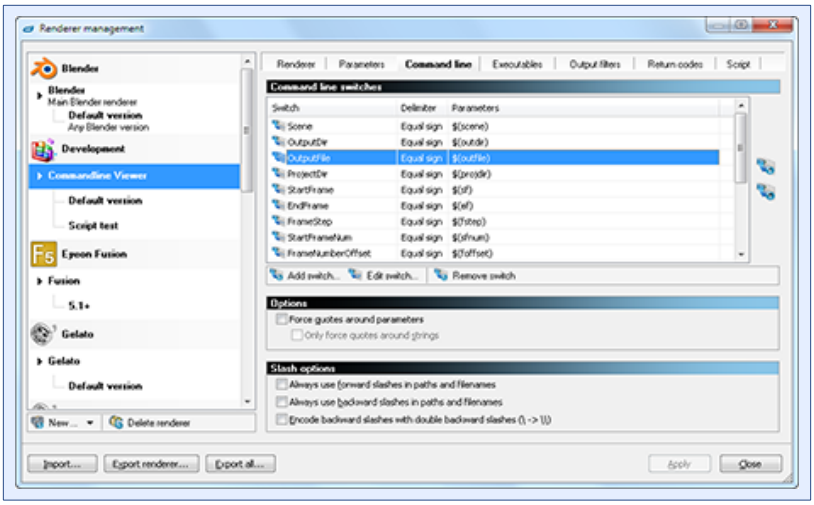
| Name | Description |
|---|---|
| **Parameter** | |
| ID | A unique ID for this parameter; parameters of a renderer can be overwritten in a descendent version. |
| Mandatory parameter | If set, this parameter must be filled out in the render set; if not filled out, an error will be shown when trying to create/save a net job. |
| Display name | The ID alone is usually not very explanatory, so you can also specify a more meaningful display name, which is then used in the render settings editor, the command-line Remote Controller and so on. If no display name is provided, the ID will be used instead. |
| Help text | The render settings editor also can show a small help text for every setting, which is provided in this field. |
| Type | The following parameter types exist:<br><br>**String:** A simple single-line string<br>**Integer:** A numeric, non-floating value<br>**Float:** A numeric, floating value<br>**Bool:** A true/false value<br>**Switch:** An on/off value; the user just selects whether to set the switch or not. A switch parameter doesn't have a value and is used for command-line switches without a value<br>**Text (encoded):** A multi-line text; newlines, backslashes etc. will be encoded with (a newline will become \n, a backslash \\ and so on)<br>**File:** A file value<br>**Path:** A path value<br>**Enumeration:** A "selection" value; the user will be presented a list of values to choose from |
| Type options | **Range:** If either Integer or Float are selected as the type, this option allows the user to not just enter a single value but a range (e.g. 50-100)<br>**List:** If set, the user can enter not just a single value but as many as he wants (supported for String, Integer, Float, File and Path). Such lists can also be used for splitting.<br>**No quotes:** Usually, values that contain spaces will be enclosed with quotes; if this option is set, no quotes will be added, and the value will be passed "as-is" . |
| Dependency | Some parameters only make sense if they are specified along with some other parameters (like image width and height, for example). Such dependencies can be specified for every parameter; if the parameter is filled out, but a dependent parameter is missing, an error will be shown when saving the render set. This is usually used when a command-line switch requires two or more parameters. |
| **Number options** | |
| Limit values | Specifies the range (minimum and maximum value) for numerical values; if the entered value is out of this range, an error will be shown when trying to create/save a net job. |
| **List options** | |
| Use for splitting/Force splitting | If set, the parameter can be used for additional splitting in the net job (every entry of a list parameter will then be used to create net job chunks). When Force splitting is enabled, the parameter will always be used for additional splitting. |
| Separate values | If the parameter isn't used for splitting, this defines how the parameter should be passed to the command-line. If set, the given separator is used to combine the values into one; if not set, the given command-line switch will be repeated for every value. |

| Name | Description |
|---|---|
| Put quotes around each list item, not around the entire list | If this option is specified for list parameters, quotes will be put around each item when the parameter isn't used for splitting. This means that a compiled switch using this parameter will look, for example, like this: `-switch "val 1", "val 2", "val 3"` instead of `-switch "val 1, val 2, val 3"`. |
| **Bool options** | |
| True/False values | These define what should be passed to the command-line for boolean values (since some renderers expect true/false, others 1/0 or on/off and so on). |
| **Enumeration values** | |
| The Enumeration type gives the user a selection of various values to choose from (the output file type selection is a good example for this). These values can be entered in the form of `Name=Value` into this list, each entry on a single line; the user will see the `Name` entry and `Value` will be passed to the command-line. | |

This might all sound very theoretical; the easiest way to get a feeling for parameters and their various options is to take a look at existing renderers and to play around with them a bit. You should just create a new renderer and experiment with the various settings.

## Command-line switches

The main task of a renderer (in most cases) is to create a command-line based on the various parameters the user has filled out in the render set. How this should be done is defined by the command-line switches. The command-line switches list contains several switches which will be processed one by one (the order of switches can be changed via drag and drop, by using the move up/move down buttons or using Ctrl+Up/Down arrow); a switch consists of the switch itself (e.g. -camera), a delimiter (to separate the switch from its value) as well as the value.




When editing a version, inherited switches will be shown in gray. Inherited switches can't be modified, but you can overwrite an inherited switch by adding a new one with the same switch - switches in the version will always have priority over inherited ones. You can also rearrange inherited switches, as well as disable them for certain versions.

When adding or editing a switch, the dialog seen on the right will appear. A command-line switch consists of the switch itself, a delimiter (a space ( ), equal sign (=) or a colon (:)) and its parameters (the passed value(s)).


A compiled command-line switch will basically look like this:
`<switch><delimiter><parameter values>`

Renderer parameter values are passed to a command-line switch in its parameters field using the following syntax: `$(param-id)` (e.g. `$(threads)`). RenderPal V2 will replace the parameter placeholder with the value set in the render set. If no value is given and the parameter isn't enclosed with square brackets (making the value optional), the entire switch will be omitted from the resulting command-line. It is possible to add multiple parameters for the value, and "fixed" values can be added as well. If a value for a switch is optional, enclose it with square brackets (e.g. [$(sf)]); if the value

for the optional parameter is missing, the switch won't be omitted - the value will just be left out.

**Examples:** Command-line switches

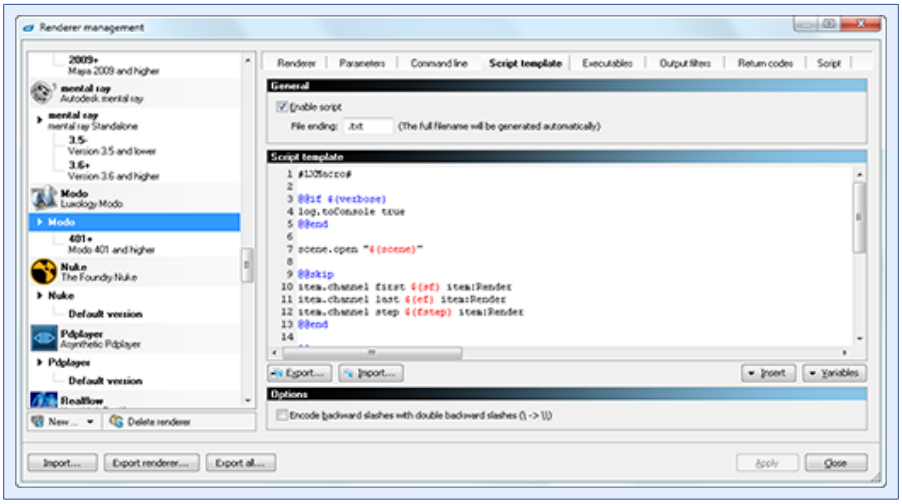
```
-threads $(threads)
In: threads = 2; Out: -threads 2

-imagesize $(imgwidth) $(imgheight)
In: imgwidth = 100, imgheight = 250; Out: -imagesize 100 250

-test1 $(val)
In: val = nothing; Out: switch omitted (value for parameter 'val' missing)

-test2 [$(val)]
In: val = nothing; Out: -test2

-test3 [$(val)] $(val2)
In: val = nothing, val2 = Hello; Out: -test3 Hello
```

There are also several options available which further define how the resulting command-line should look like. When adding or editing a version, it is possible to override these options or use the inherited ones. Here is a list of all options:

| Name | Description |
|---|---|
| **Options** | |
| Force quotes around parameters | By default, only parameters whose values contain spaces will be enclosed with quotes; setting this option will force quotes around any value. |
| Only force quotes around strings | If enabled, the above option will only apply to parameters of a string type. |
| **Slash options** | |
| Always use forward slashes in paths and filenames | If enabled, all slashes will be converted to forward slashes (/) before being passed. |
| Always use forward slashes in paths and filenames | If enabled, all slashes will be converted to backward slashes (\) before being passed. |
| Encode backward slashes with double backward slashes | If enabled, backward slashes will be replaced by double backward slashes (\ becomes \\); this might be necessary for renderers which interpret backward slashes as an escape character. |

## Script template

Some applications only offer command-line support through script files: You write a small script file with settings and commands for the application, and the application executes the script. RenderPal V2 makes writing renderers based on such scripts very easy with its script templates.

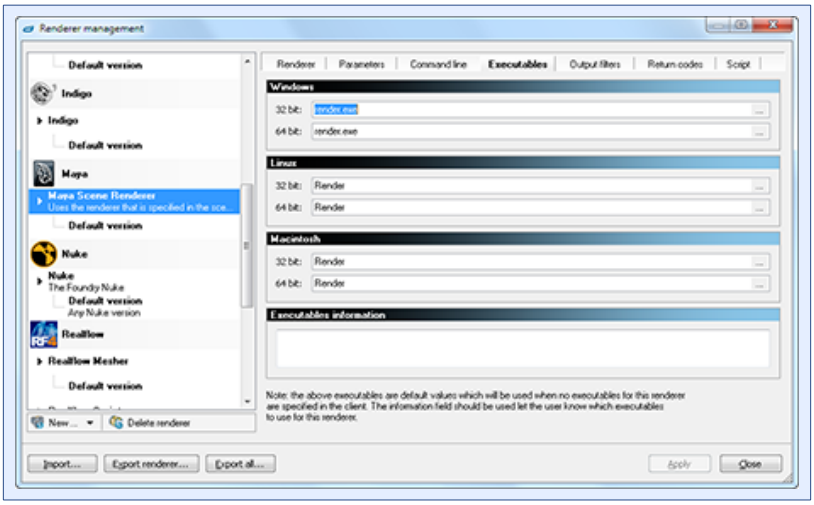


A script template consists of two things: A file ending for the resulting script file (since some applications will expect a specific file ending) and the template itself. The editor allows you to quickly insert the most commonly used block-types, as well as any variable from the parameters list (you can also bring up the variables menu by pressing Ctrl+Space). Script templates and the simple (but very effective) script template language will be explained in detail in a separate chapter.

**See also:** Script templates

## Executables

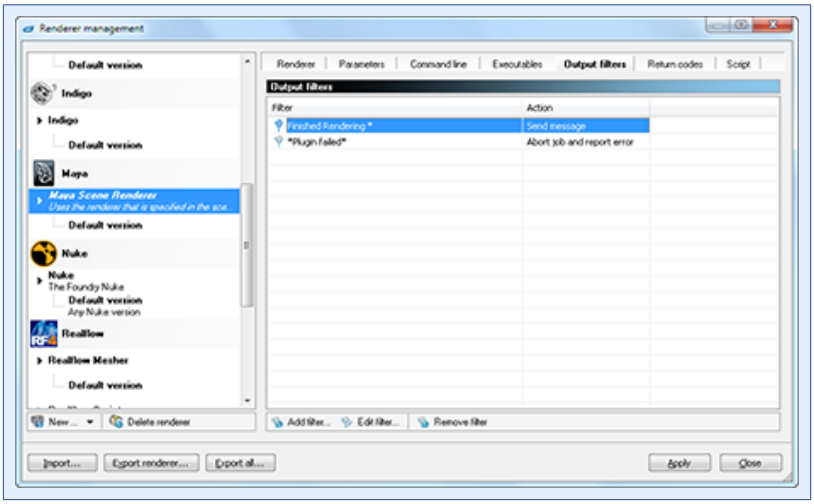
Executables can be set directly in the renderer or version (version has precedence over renderer, as usual). These executables are defaults which will be used when nothing is configured in the client for the version and its parent renderer. Executables can be configured for all operating systems (Windows, Linux and Macintosh) and for both 32 and 64 bit.



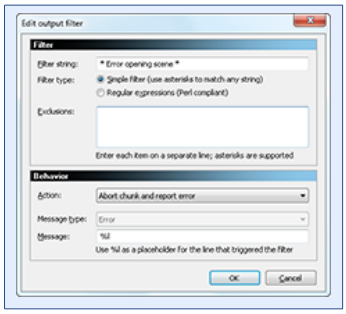

The executables information is a text field where notes about which executables to choose for this renderer/version should be entered, so the user knows which executable to actually select; this information is also displayed when configuring clients.
Setting default executables is especially useful when renderers on all (or at least most of) your clients have the same path; this makes it unnecessary to configure them on the clients.

## Output filters

Most renderers will print text about what they are doing to the output window in the RenderPal V2 Client. This text includes information about the rendering progress, notes about finished images, and also error reports. The output filters of a renderer/version allow you to react to specific strings and act accordingly. Actions include sending a message to the server (which will then appear in the chunk's history and Last message column), restarting or cancelling the current chunk or aborting it. This allows you to react to specific status events and errors with ease.



An output filter consists of a text string to look out for, as well as an exclusion list, the action to take and the message and message type to send to the server. Every line of output is then sent through these filters and matched against them (for simple filters, the asterisk (*) is supported as wildcard to match anything; for more complex filters, you can use regular expressions (Perl compliant) as well). If a match is found, the action configured in the filter is executed. Version filters have precedence over renderer filters (and thus renderer filters can be overwritten in a version).




When adding or editing an output filter, the dialog seen on the right will appear. The filter string is the ”triggering” text of the filter; use the exclusion list to exclude certain strings. The asterisk (*) will match any text, so you could enter something like **”Plugin * failed to load”**, for example. There are five different action types:

**Send message**
This will send a text message to the server, which will then be added to the net job chunk’s history log (and shown in the Last message column of the net job chunk list as well). This is useful to report the rendering’s progress to the server. You can also select the type of the message to report: Notice, Info, Warning and Error. The message type only affects the way the message will appear in the history log, not the rendering itself.

**Restart chunk**
The current chunk will be cancelled and immediately restarted.

**Cancel chunk**
The current chunk will be cancelled and the chunk’s status will be set to cancelled. This is the same as when the user manually cancels the chunk, so it will not be automatically picked up again for rendering.

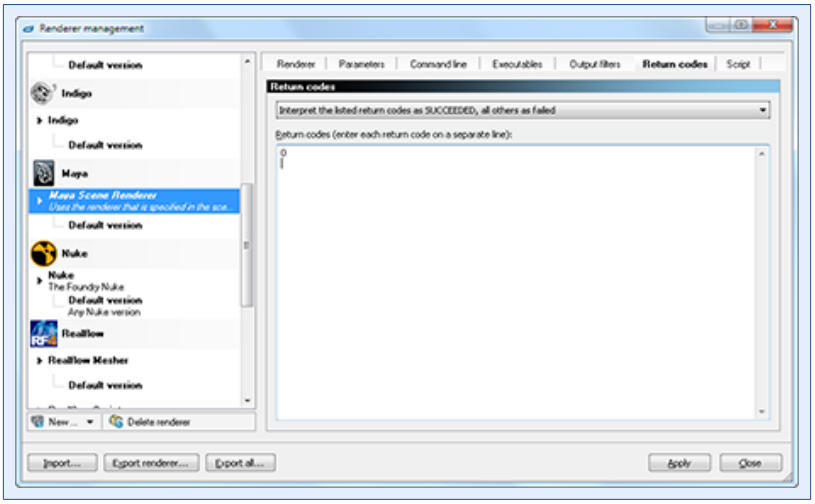
**Abort chunk and report error**
The current chunk will be aborted and the chunk’s status will be set to erroneous; an error will be reported to the server as well. The chunk will be automatically picked up again for rendering.
The message field is used to set the message that should be sent to the server (this is optional for Cancel chunk). The `%l` placeholder can be used to include the entire line that triggered the filter.

**Callback function**
If you want to react to a triggered output filter in a more advanced way, you can also let it call a custom Python script callback function. These callbacks have two parameters (the line that triggered the filter and the render set) and should return `rpDefault`. When adding a new output filter that uses a callback function, this function will be automatically added to the renderer’s Python script.

**See also:** Perl regular expressions

## Return codes

When the renderer process has finished, a numerical value will be returned. This value usually indicates whether a rendering was successful... or not; in most cases, 0 means success, while anything else means error. Please note that not all renderers make use of this feature and just return 0 (so they always report success).



Since not all renderers follow the ”0 for success, anything else for error” paradigm (or you might want to ignore specific error codes, for example), RenderPal V2 allows you to specify your own return codes and how they should be interpreted.

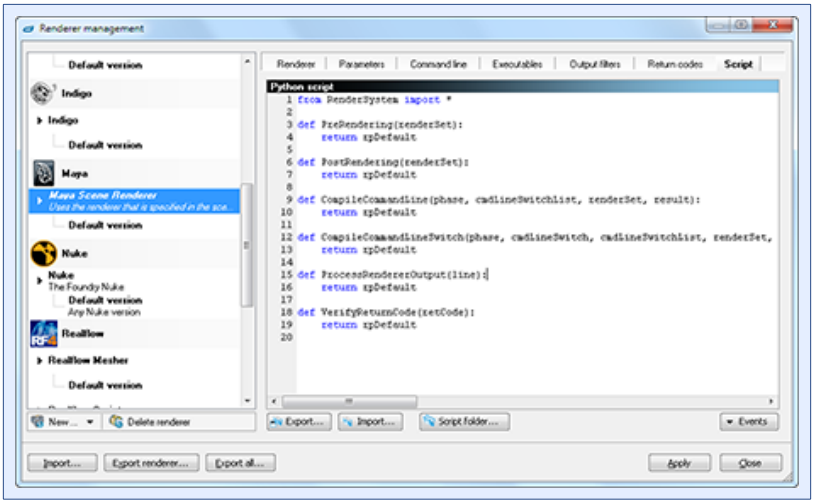
There are three ways in which RenderPal V2 can handle return codes. It can either entirely ignore it, interpret only certain codes as ”success” while all others will be seen as errors or interpret only certain codes as ”error” while all others will be seen as successes.

The list of return codes contains numerical values (each entry on a separate line) that should be either seen as a success or as a failure, depending on the selected interpretation type.

A version cannot change the interpretation type; this is set in the renderer. However, a version can further extend the return code list by adding new codes.
To use the common paradigm described above, set the interpretation type to Interpret the listed codes as SUCCEEDED, all others as failed and enter a 0 in the return codes list. If you aren’t sure whether a renderer properly makes use of return codes, you should simply turn return code checking of and use output filters to catch errors.

## Script

While for most renderers the default behavior of RenderPal V2 will be sufficient, there still might be the need to modify certain aspects of the renderer that cannot be achieved with what we described so far. For this, it is possible to write a Python script which contains callback functions that will be called by the renderer system (to add a callback, use the Events button).




Using this feature requires at least basic knowledge of the Python scripting language. This manual will not teach you Python; we will assume that you know how to code in Python. But do not worry: most renderers will not require to be extended in this way, and even if this is the case, Python is a rather easy to learn language. For more information, visit http://www.python.org.

The renderer Python API will follow in the next chapter.

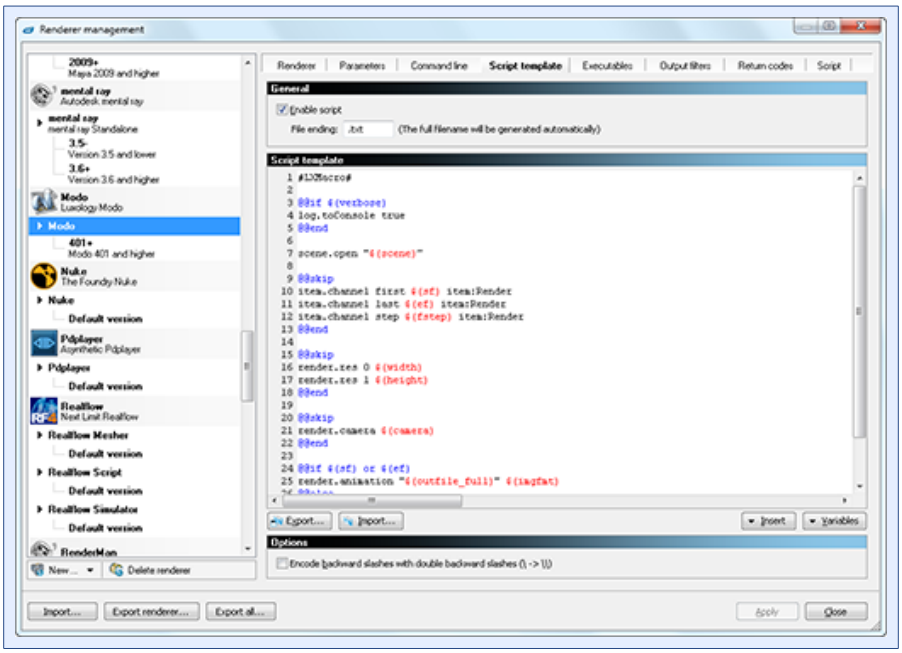
**Important:** External scripts

Care should be taken when external script files (which are not part of the standard Python library) are used within the renderer script. These scripts will **not** be automatically transferred to the clients. However, since the script will be executed by the clients, they will also require any used external script files. So before using any external resources, make sure the clients also have them available.

### 4.2.4 Script templates

Many applications can be (or even have to be) controlled via external script files; these scripts are then passed to the application with a simple command-line statement. RenderPal V2 makes creating such script files extremely easy with its script templates: Every renderer can automatically create a script file based on a template that uses a simple but very effective template language; the filename of the generated script is then placed in a standard renderer parameter (called scriptfile) and thus can be used like any other parameter in the renderer command-line.

Script files are always generated by the clients, and besides specifying a file ending, the template itself and adding the needed command-line switch to the renderer, there is not much for you to do. Below is a screenshot of the script template editor:




After activating the script, you have to specify a file ending: Some applications will expect a specific ending for their scripts, so it can be freely set. The template can be imported and exported, so you can backup them easily if you need to. You will also find buttons to quickly insert variables (i.e. renderer parameters) and blocks; these build the heart of every template and will be explained in the following chapters. The variables menu can also be brought up by pressing Ctrl+Space; blocks can be inserted by pressing Ctrl+1 to Ctrl+5.

A template basically consists of all those lines of commands, settings and so on that build the final script. There are two features that make templates dynamic: variables, which stand for the various parameters the renderer has (or, more precisely, for the values of those parameters set in the render set), and template blocks (like conditional blocks, loops and so on).

**Usage Tip:** Tolerant syntax and template errors

The template syntax is very tolerant: It is case-insensitve, you can usually add spaces wherever you want, and if you make a mistake, the template will (in most cases) still be compiled. If the template compiler encounters an error (and thus the template doesn't behave as you might expect), you should check the output log of the corresponding net job chunk (or the output window of the client); any errors will be shown there.

#### 4.2.4.1 Template variables

Templates without being able to somehow put the values specified in the render set into it would be of little to no use, so variables form one of the core pieces of template scripts. Basically, you can use every parameter of the renderer as a variable in a template. The general syntax for variables is: `$(<parameter>[:format])`, where `<parameter>` is the ID of a renderer parameter (like `sf` for the start frame) and `format` is an optional format specifier.

A simple variable could look like this: `$(sf)` or `$(camera)`; these would then be replaced with the value taken from the render set. To get more control about how the value is formatted, you can also specify a format specifier, which follows

more or less the common printf-syntax: `%[padding]<type>`, where `padding` is an optional padding (e.g. `%04` will pad with up to four zeros; note the leading zero) and the type can be either `i` or `d` for integers or `f` or `e` for floating-point numbers. If you do not specify a format, the value will be formatted "as-is". Here are a few examples:

**Examples:** Template variables

```
The given values are:
sf = 1; ef = 5; camera = SphereCam

A few template script lines and their results:
Scene.StartFrame = $(sf)       => Scene.StartFrame = 1
Scene.EndFrame   = $(ef:%04d)  => Scene.EndFrame   = 0005
Scene.Camera     = "$(camera)" => Scene.Camera     = "SphereCam"

As you can see in the last example, quotes will never be added automatically but have
to be manually where necessary.
```

If there is no value provided for a certain parameter in the render set, the variable will be replaced with an empty string; since this will usually not be what you want, you have to use blocks, which will be described in the following chapter.

### 4.2.4.2 Template blocks

The true power behind the template system are the various blocks you can use; these make the whole system truly dynamic. Each block starts with an `@@`, followed by a keyword plus some arguments (for most block types, at least); each block is then closed with an `@@end` statement. There are three types of blocks: Conditionals, Loops and Skips.

#### Skip blocks

In most cases, you will want to have lines skipped where at least one variable is not set, so that they won't appear in the generated script. A skip-block does exactly that:

```
@@skip
...
@@end
```

Lines within a skip block that contain at least one variable that is not set in the render set will be skipped (lines without missing variables or without any variables at all will not be skipped; note that not the entire block will be skipped if one variable is missing, only that particular line will be). The skip-block is the most common block type you'll use.

#### Conditional blocks

Sometimes, you might want to "execute" some lines only if a certain condition is true. The template language offers the three most common conditional statements:

```
@@if condition1
...
@@elseif condition2
...
@@else
...
@@end
```

First, `condition1` will be evaluated, and the following lines will be executed if it is true; otherwise, `condition2` (the elseif-block) will be evaluated; if it fails again, the else-block will be executed. Note that there can be more than one elseif-block, but only one else-block at the end of the entire if-block.

The conditions can check if a certain variable is set or not set; you can also combine multiple variables with an `and` or an `or`:

```
@@if $(sf) and not $(ef)
```

This will check if a value for `sf` is given `and` a value for `ef` is `not` given; only if both statements are true, the follow-

ing lines will be executed.

### Loop blocks

There can be parameters in a renderer which can contain more than one value. To iterate over all these values, you can use a loop block:

```
@@foreach $(var) in $(list-var)
...
@@end
```

The lines within this block will be repeated for every value in `$(list-var)`; the current value will be temporarily saved to `$(var)` (the name var is just an example, you can use whatever you want here).

It is also possible to nest one ore more blocks, like this (the tabs are not necessary, they were just added for clarity):

```
@@skip
    @@foreach $(cam) in $(camera)
    Render.AddCamera($(cam))
    @@end
@@end
```

If this all sounds too theoretical, take a look at the following example:

**Example:** Modo script template

```
01. @@if $(verbose)
02. log.toConsole true
03. @@end
04.
05. scene.open "$(scene)"
06.
07. @@skip
08. item.channel first $(sf) item:Render
09. item.channel last $(ef) item:Render
10. item.channel step $(fstep) item:Render
11. @@end
12.
13. @@skip
14. render.res 0 $(width)
15. render.res 1 $(height)
16. @@end
17.
18. @@skip
19. render.camera $(camera)
20. @@end
21.
22. @@if $(sf) or $(ef)
23. render.animation "$(outfile_full)" $(imgfmt)
24. @@else
25. render "$(outfile_full)" $(imgfmt)
26. @@end
```

The first if-block (1-3) executes only if the `$(verbose)` switch is set; this will then set the log.toConsole value of to true, so that more information will be printed. The next line (5) will always be executed (`$(scene)` never is empty). The skip-blocks from 7-11, 13-16 and 18-20 just set some values that should be skipped if no value is provided in the render set (this has been split into three skip-blocks for readability only). The final if-block (22-26) checks whether the start- or end-frame (`$(sf)`, `$(ef)`) are set, and if this is the case, it will render an animation (23) or a single image otherwise (25). This example also shows why you sometimes need an if-block instead of a simple skip-block: If the following lines do not contain any variables (or their execution depends on other variables), they would always be executed in a skip-block.

### 4.2.5 Renderer API

Any renderer can be extended by writing a Python script which overrides (or extends) its default behavior. The script consists of one or more callback functions which are called by the renderer system. Each callback function performs a specific job, like performing some pre-rendering tasks or compiling the command-line. This way, it is possible to change nearly every part of the rendering system.

The entire renderer API can be split into three parts: the callbacks, the various objects used in those callbacks and a bunch of helper functions. The entire script is loaded and executed by the clients before starting the actual rendering. There are a few general things to know about the renderer scripts:

- The script is only ”alive” during the execution of the current job; also, every script is completely independent
- Code outside of the callback functions is perfectly fine, so you can add additional functions, variables and so on
- To print text to the output window, use the built-in `print` statement
- The entire renderer API is available through the `RenderSystem` module, so this has to be always imported (when using the editor to add an event, the necessary code will be automatically added)
- The `RenderPal` module also has some functions and constants useful for writing renderers; these will be mentioned here as well
- It is completely up to you which callbacks to use; they can all be seen as optional and independent
- The declaration of the callbacks may not be changed in any way
- If an error occurs in your callbacks (like syntax errors etc.), the default behavior will be used and the error will be printed to the output window
- You can use all built-in modules of Python, but care should be taken when using any other external scripts, as these won’t be automatically passed to the clients

**See also:** Script-based renderers

#### 4.2.5.1 Callback functions

Currently, there are 6 callbacks you can override. All callbacks have in common that they can return one of three values:

- `rpTrue`
- `rpFalse`
- `rpDefault`

All values are constants found in the `RenderSystem` module. In most cases, returning `rpTrue` indicates that everything went fine and that the callback took care of processing; `rpFalse` indicates that an error occurred and that the rendering should be stopped. Last but not least, returning `rpDefault` will trigger the default built-in behavior to be called; this does not mean, however, that any changes you made will be lost - it only means that RenderPal V2 will continue with its default behavior, but in most cases, your changes will be used when continuing (this is especially useful when compiling the command-line).

A few functions will be called in different phases by the renderer system: an execution phase, which represents the actual operation and a post-phase, which is called once the operation has been finished. These functions will have a phase parameter, which can take the value of either `phaseExec` or `phasePost`. To override or modify the default behavior, use the execution-phase; to modify the result of the operation afterwards, use the post-phase.

Several callbacks will also use a Result object; this is used for functions that don’t just result in a simple true/false value, but which also ”produce” something, like the command-line compilation. The Result object can also be used to query the current value of an operation (this is especially useful in the post-phase).

##### *CompileCommand-line*

This function is called when the entire command-line is compiled. It can be used to completely take control over the command-line compilation or to modify it further.

```
def CompileCommand-line(phase, cmdLineSwitchList, renderSet, result)
```

**Parameters**

phase [Integer]: CompileCommand-line is called two times by the renderer system: one time for the actual compilation (`phaseExec`) and another time after the compilation has been done (`phasePost`).

cmdLineSwitchList [CmdLineSwitchList]: A list of all command-line switches for this renderer.

renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

result [Result]: The result object that holds the resulting command-line.

**Usage**

This callback is used to take control over the entire command-line compilation. It is especially useful for completely dynamic command lines that can't be put together by the normal command-line switch system. CompileCommand-line is called twice, so you should always check the value of phase. The post-phase is useful to further extend or modify a compiled command-line (which is hold by the result object).

**Remarks**

When returning `rpTrue` during the execution-phase, the post-phase will not be called.

### *CompileCommand-lineSwitch*

This function is called for every command-line switch that is being compiled by the renderer system. It can be used to react on the compilation of specific switches.

```
def CompileCommand-lineSwitch(phase, cmdLineSwitch, cmdLineSwitchList, renderSet, result)
```

**Parameters**

phase [Integer]: CompileCommand-lineSwitch is called two times by the renderer system: one time for the actual compilation (`phaseExec`) and another time after the compilation has been done (`phasePost`).

cmdLineSwitch [CmdLineSwitch]: The current command-line switch being compiled.

cmdLineSwitchList [CmdLineSwitchList]: A list of all command-line switches for this renderer.

renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

result [Result]: The result object that holds the resulting string that should be added to the command-line.

**Usage**

This callback is used to control the compilation of individual command-line switches. It is especially useful for conditional compilation or extending/modifying switches. CompileCommand-lineSwitch is called twice for each switch, so you should always check the value of phase. The execution-phase is good for conditional processing, the post-phase is useful to further extend or modify a compiled command-line switch (which is hold by the result object).

**Remarks**

Returning `rpFalse` will cause the entire command-line compilation to fail. If you want to force the renderer system to ignore a certain switch, set the result object to an empty string: `result.Set("")`.

### *GetExecutable*

This function is called to retrieve the executable for the renderer.

```
def GetExecutable(renderSet, result)
```

**Parameters**

renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

result [Result]: The result object that holds the resulting executable as a string.

**Usage**
This callback is used to use a different renderer executable than is set in the configuration. This allows for conditional executables, for example. The result object holds the original executable file when this function is called.

**Remarks**
Returning `rpTrue` will use the new executable set via result.Set("...") (as long as it isn't empty); returning `rpDefault` or `rpFalse` will use the original executable.

### *PreRendering*

This function is called before the actual rendering starts.

```
def PreRendering(renderSet)
```

**Parameters**
renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

**Usage**
This callback is called before the rendering process is launched. It is useful for writing script files which can then be used by the renderer, for example. This is an easy way to write script-based renderers.

### *PostRendering*

This function is called after the actual rendering is done.

```
def PostRendering(renderSet)
```

**Parameters**
renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

**Usage**
This callback is called after the rendering process has finished. It is useful for cleaning up.

### *ProcessRendererOutput*

This function is called whenever a line of text is printed by the renderer.

```
def ProcessRendererOutput(line, renderSet)
```

**Parameters**
line [String]: The line of text.

renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

**Usage**
This callback can be used for more advanced output processing which cannot be achieved with the standard output filters (when regular expressions are needed, for example).

**Remarks**
Do not use the print statement in this function (at least not without some kind of condition)! Any text printed during this callback will again be passed to it, so an endless-loop will be the result.

When returning `rpDefault`, the text will be passed on to the output filters; when returning anything else, they will be skipped.

### *VerifyReturnCode*

This function is called when checking the return code.

```
def VerifyReturnCode(retCode, renderSet)
```

**Parameters**

retCode [Integer]: The renderer process return code.

renderSet [RenderSet]: The render set object containing all parameter values for the rendering.

**Usage**

This callback is called after the rendering process has finished and its return code should be checked. You will rarely ever need to override this function.

### 4.2.5.2 Object types

This chapter will show you the various object types used by the callback functions. Most of these objects are used to pass data to the callbacks, but the Result type is also used to return values to the renderer system.

#### *RenderSet*

This object represents a render set. It is used to retrieve all parameter values provided for the current job. Values are accessed through their parameter ID. All values come in the form of a tuple (even if there is only a single value for the parameter).

**Methods**

[Integer] GetValueCount([paramNames [Tuple]]): Returns the number of parameter values in the render set. If an optional tuple containing one or more parameter names is passed to the function, only values of these parameters will be counted.

[Tuple] GetValueNames(): Returns a list (as a tuple) of the value names (parameter IDs) provided in the render set. This is useful when looping through all available values.

[Tuple] GetValue(valName [String]): Returns the values for the given parameter (specified by valName) as a tuple or `None` if no value with the name exists. Even if there's only a single value, a tuple will be returned.

#### *CmdLineSwitch*

This object represents a command-line switch. It is used during compilation of the command-line.

**Methods**

[String] GetSwitch(): Returns the switch of this command-line switch.

[String] GetDelimiter(): Returns the delimiter of this command-line switch.

[String] GetParameters(): Returns the parameters of this command-line switch.

[Bool] VerifySwitchDependency(paramNames [Tuple], depOnAll [Bool]): This function can be used if a command-line switch depends in some way on one or more parameters that need to be specified; paramNames contains a list of parameter names that should be checked. If depOnAll is set to `True`, the function will return `True` only if all parameters were specified; otherwise, it will return `True` if at least one is specified.

[String] Compile(): This function compiles the command-line switch and returns the resulting string or `None` if an error occurred; CompileCmdLineSwitch will **not** be called when calling Compile.

#### *CmdLineSwitchList*

This object represents a list of CmdLineSwitch objects.

**Methods**

[Integer] GetSwitchCount(): Returns the number of switches in this list.

[CmdLineSwitch] GetSwitch(index [Integer]): Returns the switch at position index as a CmdLineSwitch object or `None` if the index is out of bounds.

[CmdLineSwitch] FindSwitch(name [String]): Returns the switch with the specified name or `None` if no such switch exists.

#### *Result*

This object is used to exchange an operation result.

**Methods**

[None] Set(value [Anything]): Sets the result to value.

[Anything] Get(): Returns the current value.

[None] Clear(): Clears the result.

### 4.2.5.3 Functions

There are several useful functions provided in the `RenderSystem` and `RenderPal` modules.

#### *RenderSystem.*SetErrorMessage

This function sets the error message of the renderer system. It should be used before returning `rpFalse` from a callback to describe the error that occurred.

```
[None] SetErrorMessage(errMsg)
```

**Parameters**

errMsg [String]: The error message.

#### *RenderSystem.*Message

Sends a message to the server which will then appear in the chunk's history and Last message column. This function is usually used from within ProcessRendererOutput, but can also be used in any other callback.

```
[None] Message(msg, msgType)
```

**Parameters**

msg [String]: The message to send.

msgType [Integer]: The type of the message. The following values (all in the module `RenderPal`) are possible: `msgNotice`, `msgInfo`, `msgWarning`, `msgError`.

#### *RenderSystem.*RestartRendering

Restarts a current rendering. This can be only used from within ProcessRendererOutput.

```
[None] RestartRendering([reason])
```

**Parameters**
reason [String]: An optional reason to send to the server.

### *RenderSystem.*CancelRendering

Cancels a current rendering. This can be only used from within ProcessRendererOutput.

```
[None] CancelRendering([reason])
```

**Parameters**
reason [String]: An optional reason to send to the server.

### *RenderSystem.*AbortRendering

Aborts a current rendering. This can be only used from within ProcessRendererOutput.

```
[None] AbortRendering([reason])
```

**Parameters**
reason [String]: An optional reason to send to the server.

### *RenderSystem.*SendEmail

Sends an email (using the email queue of the RenderPal Server).

```
[None] SendEmail(subject, body, [recipients])
```

**Parameters**
subject [String]: The email subject.

body [String]: The email body (in plain text).

recipients [String]: An optional list (separated by semicolons) of additional email recipients.

### *RenderPal.*GetBuildNumber

Returns the current RenderPal V2 build number, which can be used to check which version the user is running.

```
[Integer] GetBuildNumber()
```

### *RenderPal.*GetDirectory

Returns the RenderPal V2 directory.

```
[String] GetDirectory()
```

### *RenderPal.*GetTempDir

Returns the temporary directory. This should be used for writing scripts for renderers, for example.

```
[String] GetTempDir()
```

Writes an event entry to the event log.

[None] LogEvent(msg, msgType)

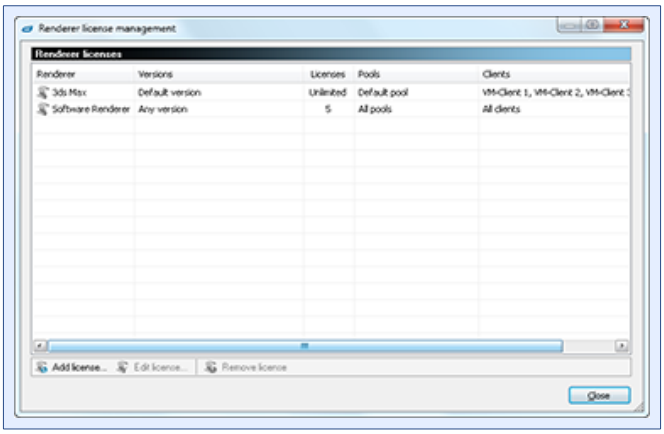
**Parameters**

msg [String]: The event message.

msgType [Integer]: The type of the message. The following values (all in the module `RenderPal`) are possible: `msgNotice`, `msgInfo`, `msgWarning`, `msgError`. These types correspond to the message types in the event log.

## 4.3 Renderer license management

Most renderers and compositing applications use some form of license management, usually in the form of node-locked licenses, where licenses are given to specific machines in a render farm, or in the form of a floating license, which only limits the number of concurrent machines at a given time.




RenderPal V2 offers a way to easily reflect your renderer licenses: For every renderer and its versions, you can specify how many licenses are available and which pools and clients own (have access to) a license. Using the license management, which is shown in the following screenshot, you can avoid to have new jobs sent to clients even if there currently is no free license slot available, eliminating common license problems.

Licenses can be added for every renderer and its versions. If no license for a specific renderer exists, no limits will be applied when rendering with that renderer. If, however, a license has been created, its limitations will be applied when a job is about to be rendered. For a license count limited license, RenderPal V2 will look if there still is a ”license slot” available; if the license is restricted to specific pools and clients, it will look if a client has access to the renderer license (either if the client itself has been selected or if it is assigned to a pool that has been selected in the license settings).

### 4.3.1 License settings


The screenshot on the right shows the various license settings. For every license, you have to select its renderer and the versions it should apply to (or all versions; this is especially useful if you intend to create further versions of the renderer in the future).

If you specify a license count, you tell RenderPal V2 that there are only X licenses total available in your render farm. This is useful to define floating licenses, which are not bound to specific nodes.

You can also restrict the license to specific pools and clients; only clients that are selected directly or are assigned to a selected pool will pick up jobs which use the renderer. Pools always have priority over clients: Even if a client is unchecked in a specific pool, it will still be used since its pool is selected. This is also reflected by clients in a selected pool being drawn in gray. Restricting licenses this way is useful to define node-locked licenses.

Usually, you will not specify both a license count and pool/client restrictions, but it can be useful in some cases.

# Part II - Features

This part is dedicated to all the numerous features found in RenderPal V2. All important features will be covered in detail, with in-depth information on their functions and usage. We will also give useful tips on how to use the features in special (sometimes not so obvious) ways.

All features presented here have a certain scope which they apply to (server, client, remote controller and so on). Many features appear in multiple instances in RenderPal V2 (especially in the server and remote controller, since they offer almost the same set of features), and we don’t want to explain them two or more times.

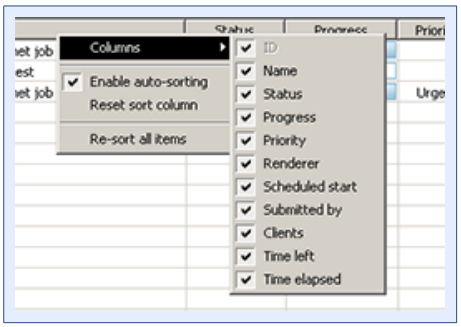
# 5. General

**Applies to:** Global

The interface of RenderPal V2 has been designed to be intuitive and easy to use; there are no hidden features (that can only be accessed via keyboard, for example). RenderPal V2 extensively makes use of context menus (the menus that pop up when right-clicking somewhere), especially in lists. If you can't find a certain function, try to look into the main menu or the corresponding context menu first.

## List controls

In most lists (like the client pool list, net job list and so on) it is possible to select which columns to display; this can be useful to save screen space. Showing or hiding certain columns can be done using the list header context menu; the menu also offers some sorting related functions.




Lists also allow you to search for certain text in a specific column. List search can be accessed either via the list's context menu or keyboard.

Column visibility and list sorting are preserved on exit.

## Common keyboard shortcuts

There are a few shortcuts that are used all over the place in RenderPal V2, most of them in lists. Here is an overview of all common keyboard shortcuts:

| Scope | Shortcut | Function |
|---|---|---|
| **Lists** | Ctrl+F | Find |
| | F3 | Find next |
| | Ctrl+A | Select all |
| | Ins | Create new entry |
| | Del | Delete entry |
| **Tabbed dialogs** | Ctrl+Tab | Next tab |
| | Ctrl+Shift+Tab | Previous tab |
| **Floating windows** | Ctrl+W | Close window |
| **Miscellaneous** | F5 | Refresh (where applicable) |

There are plenty of other global shortcuts; these can be found when taking a look at the various menus (main and context) of RenderPal V2.

# 6. Clients

**Applies to:** Global

It is important to distinguish between the clients in the server and the actual client program, the RenderPal V2 Client, which executes the rendering and keeps track of all rendering events.

A client consists of the connection address (IP or NetBIOS/computer name), as well as a few other settings. The most important setting is the client priority, which determines the hierarchy of clients inside a client pool. The RenderPal V2 Client itself offers numerous options, most of which are renderer related.

**Usage Tip:** Client addresses

For networks with static IPs, it is recommended to add clients using their IP address. For dynamic networks, clients should always be added using their NetBIOS name.

**Usage Tip:** Client priorities

Client priorities should be assigned according to how powerful a certain computer is. Powerful machines should get the highest priorities, while weaker ones should get lower priorities.

Clients can be accessed and controlled in several ways using the server or remote controller. This includes pausing clients, job control, output retrieval and system control commands. It is also possible to remotely configure the clients.

**See also:** Client management

## 6.1 Parallel rendering

RenderPal V2 supports parallel renderings on a single client. This can be especially useful if the used renderer or compositing application doesn't support multi-threaded rendering and you want to harness the full power of your client computers.

Parallel rendering is a net job specific setting; all net jobs have a setting that sets the number of parallel renderings that should be executed on a client. If you don't want any parallel renderings, simply set this setting to 1; if you want to have one rendering per CPU core, set it to 0.
When starting multiple renderings on a machine, heavy system and network traffic can occur. Due to this, each client has a certain start delay between each consecutive job; this should be set to an appropriately high value to reduce loading times and network traffic. The default for this is 15 seconds, which should be fine in most cases.

**Important:** Renderers and parallel rendering

While technically all renderers can be used for parallel renderings, some might behave erroneously when being executed more than once at the same time. If your renderings keep on failing, try to first increase the start delay; if they still keep failing, don't use parallel rendering for that specific renderer.

A client can only render chunks of the same net job in parallel; rendering chunks of different jobs is not possible. You will need to run multiple clients on the machine in order to achieve this.

**See also:** The net job editor, Client options, Remote client configuration

## 6.2 The rendering process

When a client receives a new job from the server, it will first execute any client pre-execution chunk events. After these have been processed, the actual rendering will start. Once rendered, any client post-execution chunk events will be executed.

For every job, the client creates a new output buffer, which will receive all textual output from the renderer, as well as the net job events. This output can later be retrieved using the server or remote controller. The number of output buffers can be limited to restrict the amount of memory the client consumes. Also, each output buffer has a maximum size of 3

MB to prevent excessive flooding (if the limited is exceeded, old text will simply be removed).

During the rendering process, RenderPal V2 will keep track of any events that occur during the entire rendering process, including the job status, errors and so on. These events will be shown in the client's event log (also remotely retrievable) and the net job chunk history; this is especially useful when certain clients fail to render a job.

RenderPal V2 allows you to select which system architecture to use: 32 and/or 64 bit. This only determines which executable should be used (different executables can be configured for 32 and 64 bit), but has no other effect. If 64 bit is selected, but no executable is configured, the renderer system will try to use the 32 bit executable.

**Usage Tip:** Failing renderings

When renderings keep on failing on certain clients, take a look at their output and event log first. In most cases, the renderer will tell you what the exact problem was. It is also possible to exclude a client from a job after a certain number of chunk errors.

## 6.3 System control commands

It is possible to reboot or shutdown a machine running the RenderPal V2 Client using the server or remote controller. The timeout and the reason for the shutdown/reboot can be specified; it is also possible to force an immediate shotdown/reboot by specifying a timeout of 0. In the graphical client, this process can be aborted; the command-line client will execute system control commands immediately. RenderPal V2 can also ”wake up” (turn on) clients using Wake-on-LAN if the computer supports it (either manually or automatically).

It is also possible to execute an arbitrary command-line on one or more clients (or all clients of an entire pool). This allows you to, for example, install software remotely on all machines running the RenderPal V2 Client. Due to the power of this function, it has its own user right.

**Important:** Linux/Macintosh client

System control commands (except **Wake up**) require the client to be ran as root under Linux and Macintosh to work.

# 7. Client pools

**Applies to:** Global

Client pools are responsible for the dispatching of net job chunks to their assigned clients. The way these chunks are dispatched is determined by their dispatch mode, which will be described shortly. Just like clients, pools are also ordered in a hierarchical way, determined by their priority.

A client pool can be either in started or stopped state; only started pools will be processed. When stopping a pool that contains clients which are currently rendering, these renderings will be aborted. It is also possible to define a schedule when a pool (or, to be more precise, their assigned clients) is available for rendering.

Client pools, as well as their assigned clients, can be controlled in various ways, including job assignment, system control commands, remote client configuration and automatic shutdown of idle clients. Most client control functions can also be executed for an entire pool.

**Usage Tip:** Grouping clients

The organization of client pools should be carefully planned. Common ways of organizing clients is to create separate pools for user machines and render machines or pools for various operating systems. Pool priorities should be used to support the pool hierarchy (render machine pools should receive a higher priority than user machine pools, for example).

**See also:** Client management

## 7.1 Dispatch modes

RenderPal V2 supports three dispatch modes, which determine how the net job chunks of their assigned net jobs will be distributed across all clients in the pool. The three dispatch modes are:

**Job sequential**
In this mode, all jobs will be dispatched in a sequential manner. This means that the job with the highest priority will get all available clients first. Each job is processed one after the other; parallel rendering only occurs if there are still free clients left once the highest job has been processed.

**Priority grouped parallel**
This mode will group net jobs with the same priority together and will dispatch all available clients equally among these jobs.

**Balanced parallel**
In this mode, all available clients are distributed in a balanced manner across its assigned net jobs. The priority of a net job determines how many clients a net job will receive: jobs with a priority of 10 will get a few more clients than jobs with a priority of 9 and so on.

In most cases, **Balanced parallel** will be the mode of choice; it is also the default mode for new pools.

**Usage Tip:** Mixing dispatch modes

It is possible to have different dispatch modes for your client pools. Since all pools are encapsulated, independent units, this allows for an even finer control of the dispatching process. One possible scenario is to set one of your pools to **Job sequential** mode, while all other pools use **Balanced parallel**. This way, net jobs with the highest priority will receive some extra clients from the ”Job sequential pool”.

## 7.2 Idle client shutdown

A pool can be configured to automatically shut down any idling clients when they have reached a certain idle time. This setting is useful to shut down unused clients to save energy, for example. Note that this setting is **not** preserved on server exit.

# 8. User groups and accounts

**Applies to:** Global

User groups and accounts, which come to use when using the RenderPal V2 Remote Controller, specify the permissions of a user, which pools the user has access to, as well as when jobs submitted by an user may be rendered.

There are various user rights which define what a user may do and what not; these rights will be described in the next section. A group/account can also be limited to certain client pools; the user will only be able to see the pools he has been granted access to (this does not affect Client management).

It is also possible to specify a schedule for user groups and accounts. Jobs submitted by the user will only be rendered during the scheduled times.

A user account can be part of a user group. User groups are an easy way to define global settings which will then be inherited by all assigned accounts. It is also possible to override specific group settings in a user account.

**See also:** User management

## 8.1 User rights

The following tables show all available user rights:

### General

| Name | Description |
|---|---|
| Show net jobs from all users | Net jobs from all other users, as well as jobs created directly in the server, will be shown if this right is granted. Otherwise, only jobs submitted by this user will be shown. |
| Grant full rights for own net jobs | This right grants the user full control over his own net jobs; all other jobs will obey to the granted permissions. |

### Net Jobs

| Name | Description |
|---|---|
| Create net jobs | This right allows the user to create new net jobs. |
| Edit net jobs | This right allows the user to edit existing net jobs. |
| Edit net job events | Net job events can be used in potentially dangerous ways, so this permission has been added. If granted, the user can add and edit net job events. |
| Allow submission of urgent net jobs | This right allows the user to submit urgent net jobs. |

### Net Job control

| Name | Description |
|---|---|
| Pause/unpause net jobs | This right allows the user to pause and unpause net jobs. |
| Cancel/restart net jobs | This right allows the user to cancel and restart net jobs. |
| Delete net jobs | This right allows the user to delete net jobs. This also affects the removal of finished net jobs. |
| Change chunk priority | This right allows the user to change the priority of net job chunks. |

### Pool & Client control

| Name | Description |
|---|---|
| Control client pools | This right allows the user to control client pools. |
| Control render clients | This right allows the user to control render clients (excluding system control commands). |
| Allow client system control | This right allows the user to reboot, shutdown and wake up clients. |

| Name | Description |
|---|---|
| Configure clients | This right allows the user to remotely configure clients. |
| Remote program execution | This right allows the user to perform remote execution on clients. |

### Administrative

| Name | Description |
|---|---|
| Access client management | This right allows the user to access the client management. The shown pools are **not** affected by the pool access of this user. |
| Access user management | This right allows the user to access the user management. An user cannot edit or remove his own account. |
| Access path maps | This right allows the user to access the path maps. |
| Access renderer management | This right allows the user to access the renderer management to create and edit renderers |

Which net jobs are shown and which may be edited and controlled is determined by the rights **Show net jobs from all users**, **Grant full control over own net jobs** and the various net job related rights. If **Grant full control over own net jobs** is set, all rights will be granted for net jobs from the logged in user; otherwise, own net jobs will obey to the granted rights.

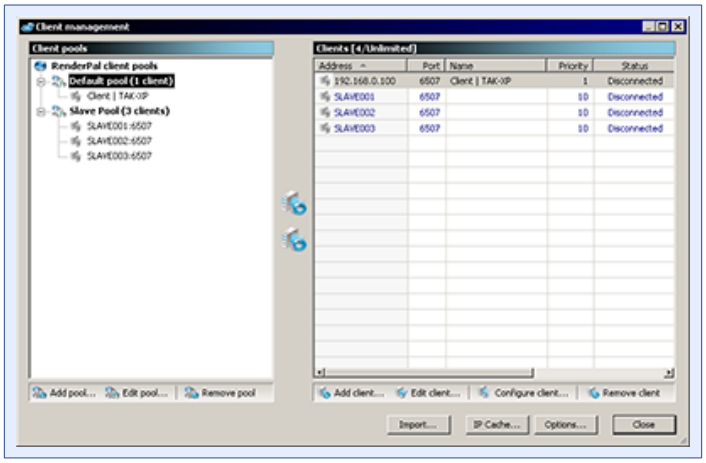
**Usage Tip:** ”View but don’t touch” accounts

There are many ways to combine the above user rights. One common example is the creation of guest accounts, which may look at everything, but not change anything. To achieve this, remove all user rights except for **Show net jobs from all users**. The logged in user will see everything, but has no right to modify anything.

# 9. Client management

**Applies to:** Server, Remote Controller

All client pools and clients are managed using the client management. The client management is divided into two parts: the client pools, as long as their assigned clients, and the known clients list. The screenshot below shows the client management dialog:




Clients can be assigned to new pools either by simply dragging clients to a pool, via context menu or by using the Move left/Move right buttons. It is also possible to import clients from either the HOSTS or LMHOSTS file; the IP Cache and Options buttons allow you to quickly access various client management related options (only available in the server).

**Usage Tip:** Configuring clients

One benefit of the known client list is that it shows all clients in your render farm; it furthermore shows various details about all clients, like their priority and operating systems. This makes it easy to perform remote configuration for a specific ”type” of clients: sort the list accordingly, select the clients you want to configure and click Configure client.

**See also:** Clients, Client pools

## 9.1 Client pool settings

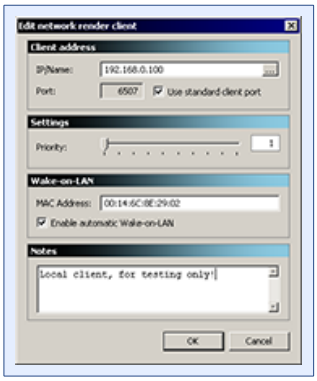
The screenshot on the right shows the client pool settings dialog, which will come up when adding or editing a client pool. In this dialog, you can edit all pool related settings, including its name, priority, dispatch mode, which groups and users have access to this pool, automatic idle clients shutdown scheduling, as well as some notes.


When activated, automatic idle clients shutdown will be activated for the pool during the scheduled times. This is especially useful to let an entire render farm be shut down after working hours once all renderings have been finished, for example.

The pool schedule can be edited by clicking Edit schedule.

**Note:** The client pool access for user groups and accounts can also be changed in the user group/account settings.

## 9.2 Client settings

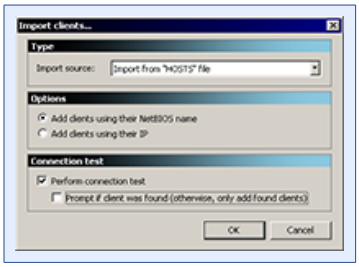



When adding or editing clients, the dialog shown on the right will come up. In this dialog, you can set all client related settings, including their connection address, priority, Wake-on-LAN options and some notes.

RenderPal V2 can turn on clients using Wake-on-LAN (if the computer supports it). For this to function, RenderPal V2 needs to know the MAC Address of the client; when using the graphical client under Windows, this setting will be filled automatically (for Linux and Macintosh, it has to be filled out by hand). Clients can be woken up manually, but also automatically whenever the client is needed for rendering; if Enable automatic Wake-on-LAN is turned on, the server will try to wake up the client if it can't connect to it otherwise.

By default, RenderPal V2 will use a standard port for each client; you can override this by unchecking Use standard client port. The default client port can be changed in the server settings.

## 9.3 Import clients

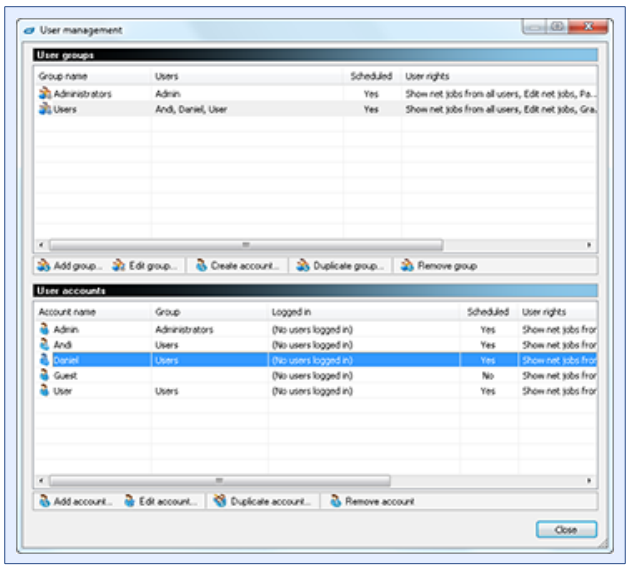



RenderPal V2 allows you to import clients from either the HOSTS or LMHOSTS system files. You can select if the imported clients should either be added using their IP address or their NetBIOS name. It is also possible to perform a connection test before adding the clients. If Prompt if client was found is checked, RenderPal V2 will show a confirmation for each found client; otherwise, only clients that actually exist will be added.

# 10. User management

**Applies to:** Server, Remote Controller

User groups and accounts are created and modified via the user management, which is shown in the following screen-shot:




You can quickly create new accounts belonging to a group by selecting that group and pressing the Create account button (or by using the context menu/keyboard shortcut). To quickly assign a user account to a group, drag the account onto the group. Instead of creating groups or accounts from scratch, it is also possible to quickly duplicate them.

**Important:** Active groups/accounts

Groups and accounts that are currently in use cannot be edited or removed. However, it is still possible to modify the pool access of a group or account in the pool settings (client management). You can also disconnect all active users of a user account.

**See also:** User groups and accounts

## 10.1 User group/account settings


Settings of an user group or account are modified using the dialog shown on the right (the dialog shows the user account settings - the group settings are pretty much the same). Besides its name and optional password (accounts only), an account can also be given an e-mail address, which is used to send e-mail notifications about net jobs created by this user.

If an account is assigned to a group, it will inherit and use all settings of that group. It is, however, also possible to override certain settings in an account.

The Job priority limit specifies the highest priority limit for net jobs this user can submit. If Allow multiple logins is not checked, only one user can log in using this account at a time. The default net job color is used when the user creates a new net job; the color is only set as the default for new net jobs and can always be changed in the net job dialog.

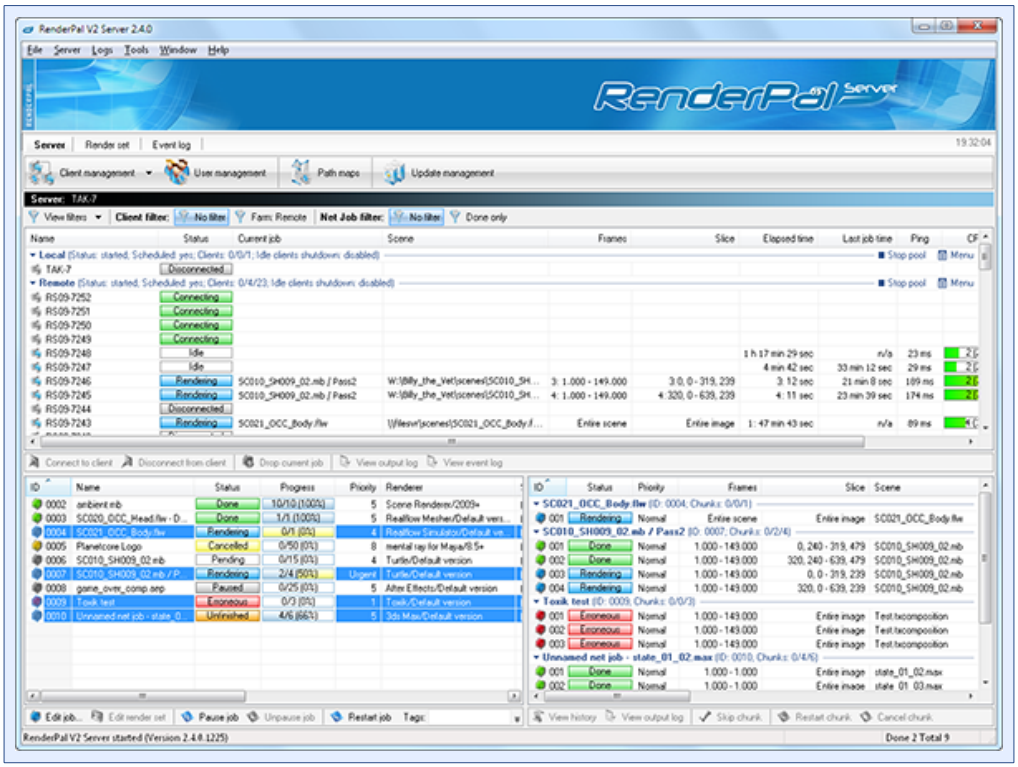
The user rendering schedule can be edited by clicking Edit schedule. Jobs from users will only be rendered if they are on schedule.

**See also:** User rights

# 11. The server tab

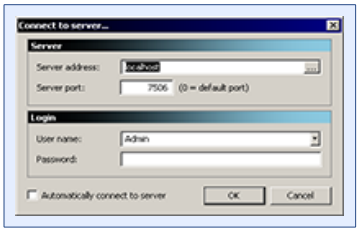
**Applies to:** Server, Remote controller

The server tab is the center of your render farm, showing all your pools, clients, net jobs and their various chunks. All monitoring and controlling is performed here. The server tab is present in both the RenderPal V2 Server and Remote Controller, with a small number of differences, which will be discussed later. To limit the shown pools, clients and net jobs, the server tab also offers so-called view filters, which are especially useful in larger farms. The following screenshot shows the server tab (often referred to as the **main view**) of the RenderPal V2 Server:



The top toolbar allows you to quickly access the Client management, User management and the Path maps. In the RenderPal V2 Server, you can also access the Update management; the RenderPal V2 Remote Controller allows you to refresh the main view, as well as to connect to/disconnect from the server.

### RenderPal V2 Remote Controller



To use the remote controller, you have to connect to the RenderPal V2 Server first; this can be achieved by clicking on Connect on the top toolbar (which will become Disconnect once connected). This will bring up the dialog shown on the right. Enter the server address, as well as your user credentials, here. If you want to let the remote controller automatically connect on startup, check Automatically connect to server.

#### *Data refreshing*

The data presented in the remote controller will usually be updated automatically by the server. The Refresh button on the top toolbar can be used to manually refresh the main view; this can be useful before performing critical tasks.

**See also:** Monitoring and controlling net jobs and clients

## 11.1 The client pool list

The client pool list shows all pools as ”header-like” items, as long as their assigned clients. The toolbar below the client pool list contains several client related functions (buttons in the toolbar will only be active when a client is selected).

### Client pool items

Client pool items show information about the pool's status, the number of rendering/active/total clients and whether idle shutdown is active. The pool context menu offers numerous control functions, like starting and stopping, changing the dispatch mode, several job assignment related functions and more.

### Client items

Client items show the status and activity of the various clients in your render farm. A client can be controlled in many ways via either the client pool list toolbar or context menu. This includes functions for connecting and disconnecting, job and job assignment control, system commands and output and event log retrieval.

## 11.2 The net job list

The net job list contains all jobs of your render farm. The list shows various details about every job, including its status, renderer, overall progress, the time already spent on the job and an estimation on the remaining time. Net jobs can be controlled in many ways, including editing the job or the underlying render set, pausing/unpausing and cancelling/restarting. It is also possible to perform manual frame checking, which will be explained in a later chapter, on a job.

As long as the net job chunk list is visible, the chunks for all selected net jobs will be shown.

It is possible to quickly filter the net job list using net job tags (which can be freely defined for every net job). Simply type one or more tags (separated by commas or semicolons) into the small Tags edit field in the net job toolbar, and only jobs which contain at least one of the entered tags will be shown. Clear the edit field to show all net jobs again.

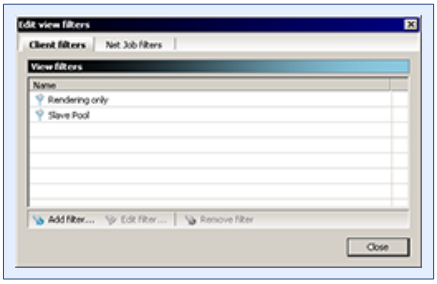
## 11.3 The net job chunk list

The net job chunk list shows the various chunks of all selected net jobs, grouped into their corresponding net jobs. This list can be shown or hidden using the main Window menu or the net job list context menu. Chunks can, just like jobs, be controlled in many ways. You can change their priority, skip, cancel or restart certain chunks, view their history (a small summary of all chunk-related events) and retrieve the output from the last client that has processed this chunk.

**Usage Tip:** Hiding the chunk list

It is recommended to hide the chunk list if you don't use it. This saves both CPU and network traffic (in the case of the RenderPal V2 Remote Controller). You can also use **Ctrl+L** to quickly show or hide the chunk list.

## 11.4 View filters

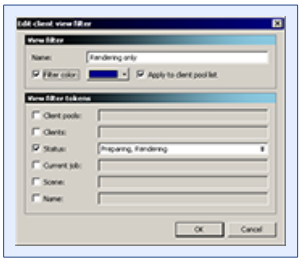



View filters can be used to only show pools, clients and net jobs that match certain, user-defined criteria. View filters come in two flavours: **client filters** and **net job filters** (client filters can also be used to filter client pools). In larger render farms, the number of clients and jobs can become quite overwhelming, so view filters offer a mechanism to minimize the number of shown items to only those you are interested in. This could include clients that are only rendering, jobs that are finished and so on. The screenshot below shows the view filter dialog. Filters can also be added, edited and removed directly in the view filter bar using context menus.

A view filter consists of a name, an optional color that is applied to the filter bar button and the corresponding list and the various filter tokens, which determine the criteria an item has to fulfill to be shown. If multiple tokens are selected, **all** have to be fulfilled for an item to be shown. Textual tokens also support the use of wildcards (*) to match any text.

View filters always have priority over tag filtering: First, all net jobs that don't match the current view filter will be removed, then those that do not match any of the entered tags.

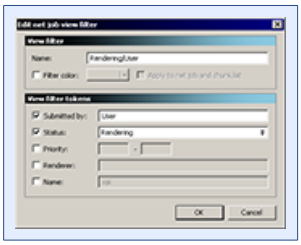
### Client view filters



Client view filters are used to filter the shown client pools and clients; the screenshot on the right shows the client view filter editor. The following table lists all available filter tokens:

| Name | Description |
|---|---|
| Client pools | Only the selected client pools will be shown |
| Clients | Only the selected clients will be shown |
| Status | Only clients which match the selected statuses will be shown |
| Current job | Only clients rendering the selected net jobs will be shown |
| Scene | Only clients rendering a scene containing the entered text will be shown |
| Name | Only clients whose name contains the entered text will be shown |

### Net job view filters



Net job view filters are used to filter the shown net jobs; the screenshot on the right shows the net job view filter editor. The following table lists all available filter tokens:

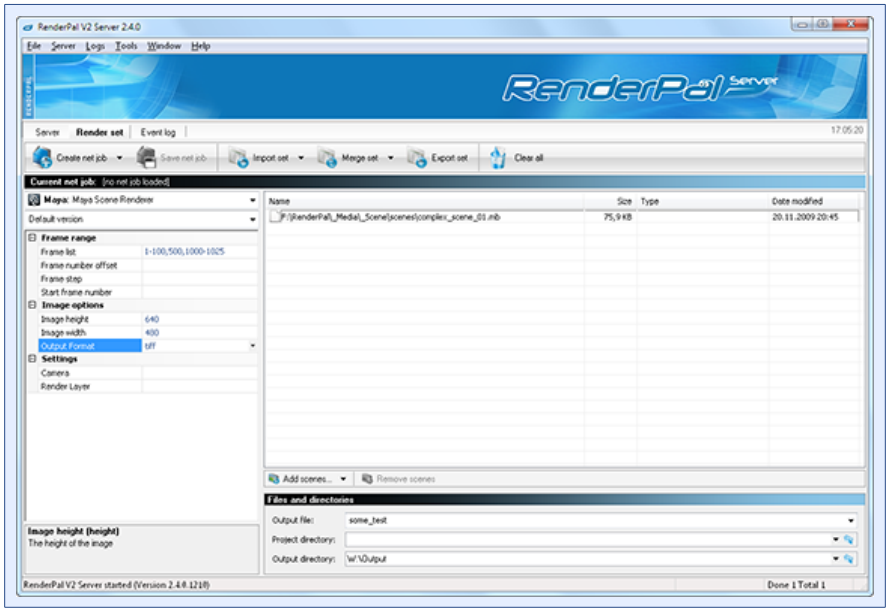
| Name | Description |
|---|---|
| Submitted by | Only jobs whose submitter name contains the entered text will be shown |
| Status | Only jobs which match the selected statuses will be shown |
| Priority | Only jobs that lie within the entered priority range will be shown |
| Renderer | Only jobs using one of the selected renderers will be shown |
| Name | Only jobs whose name contains the entered text will be shown |

**Note:** You can enter multiple text items by separating them with a semicolon.

# 12. The render set tab

**Applies to:** Server, Remote controller

The render set tab is used to write and edit render sets, which build the base for every net job. The render set tab consists of the renderer and renderer version selector, the render settings list, the scene file list and a few output file and directory settings. The following screenshot shows the render set tab in action:




### Creating and saving net jobs

Since render sets are the base of every net job, the most important functions found in the top toolbar are Create net job, which will create a new net job based on the current render set, as well as Save net job, which will be only active when you have chosen Edit render set from within the net job list of the main view; if you make any changes to an existing render set, you will have to apply them by saving the net job.

**Important:** Saving net jobs

When saving an edited render set, the net job has to be restarted for the changes to take effect. RenderPal V2 will show a confirmation in this case.

### Import, merge and export

A render set can be exported to a file to be later imported again. This can be useful if you frequently use specific render settings. It is also possible to merge an exported render set with the current one; this will only import the render settings of the selected set without overwriting any existing settings. Scenes and the files and directories settings won't be affected at all.

See also: Net job creation I: Filling out the render set

## 12.1 The renderer selector

The available render settings, as well as the file and directory options, depend on which renderer and version is currently selected. When changing the active renderer or version, settings will be preserved where possible. The active renderer will also affect the file filter when adding new scenes.

When creating a new net job, the active renderer will be used.

When adding scenes with a different ending, RenderPal V2 can automatically choose the correct renderer; this behavior can be changed in the options.

## 12.2 The render settings list

All render settings available for the selected renderer will be shown in the render settings list. All settings entered here will act as **overrides**; if left empty, the renderer will use the settings made in the scene. The entire settings list is dynamic and depends on the selected renderer and version. Below the settings list, a small box is shown which will show information about the selected setting, including its parameter ID. You can also search for a specific setting using the context menu.

## 12.3 The file and directory settings

You can specify the output file, project directory and output directory in every render set. It is recommended to always specify an output directory. If a project directory is specified, this will be the default path when adding scene files.

## 12.4 The scene file list

The scene file list allows you to add one or more scene files (or any other kind of source file) to the render set. While all other settings are optional in a render set, at least one scene has to be always added. Scenes will be rendered in the same order as they appear in the set; the order can be changed via drag and drop.

**Usage Tip:** Adding manual entries and editing entries

If you can't (or just don't want to) browse for a scene, you can simply add a new entry by hand using Add manual entry. This also allows you to add Linux or Macintosh paths directly. Entries can also be edited by simply double-clicking them or by pressing F2.

**Important:** Multiple scenes

When a render set contains multiple scenes and an output file is specified, the scene filename will be prepended to avoid file collisions. It is also important to understand that all settings made in the render set will be equally applied to **all** scenes in the set. If you need different settings for each scene, you will have to create a new render set.

## 12.5 Dynamic scene and output file names

RenderPal V2 features an advanced syntax to create dynamic scene and output file names (this also includes the output directory). Such file names contain one or more variables from the underlying render set and will be processed dynamically. Here are two simple examples of what can be done with this feature:

### Dynamic scene name

One scenario where dynamic scene names are useful is to create complex render setups where jobs use files that do not even exist yet (usually file sequences); for example, one job could render several consecutive files that will then be converted in a batch by another job. In such a case, the second job would only have one scene file that contains a dynamic variable (in this example, the frame list); the resulting job will automatically create the correct file names.

### Dynamic output path

Imagine that your scene contains various cameras, but the renderer does not put the resulting images into sub-directories. Instead of creating multiple net jobs for each camera, you can use a dynamic variable for the output path and split the job into each camera. Each chunk for each individual camera will then automatically use a different output path, depending on the current camera.

Since this is an advanced feature you won't need every day, it is explained in full detail in the advanced section.

**See also:** Dynamic scene and output file names

# 13. The net job editor

**Applies to:** Server, Remote controller

The net job editor is used to create and edit net jobs; it consists of four tabs: the main and advanced settings, net job events and notes. To use the current settings as the default settings for newly created net jobs, check Set as defaults for new jobs before clicking OK. The paused state of a net job can also be quickly changed in the net job editor.

When editing multiple net jobs at once, you can select which settings to apply; a small checkbox will appear in front of each header, and only those settings in a checked group will be applied.

### The life cycle of a net job

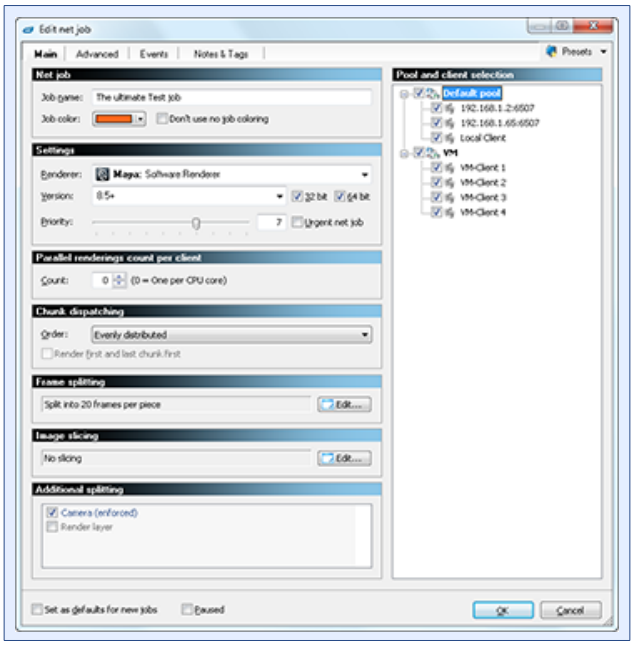
During its life time, a net job goes through many different phases. To fully understand the various net job features and how they affect these phases, we will now briefly describe them in the order they will appear (after the job has been created):

1. **Server pre-execution net job events:** Just after the creation of the net job, any server-side pre-execution events will be executed.
2. **Chunk dispatching:** The chunk dispatching begins.
3. **Server pre-execution chunk events:** Before a chunk is dispatched to a client, any server-side pre-execution chunk events will be executed. After this, the job is sent to the client.
4. **Client pre-execution chunk events:** The client will execute any pre-execution events before rendering starts.
5. **Rendering:** The client renders the job.
6. **Client post-execution chunk events:** Once the job has been rendered, any client-side post-execution events will be executed.
7. **Server post-execution chunk events:** The client has finished working on the job, so the server will execute any server-side post-execution chunk events.
8. **Frame checking:** If frame checking is set to be performed on finished chunks, this will be done now.
9. **Finishing the job:** Once all chunks have been successfully rendered, the job will be finished. First, any server-side post-execution net job events will be executed. If frame checking is set to be performed on finished jobs, this will be done at the end.

**See also:** Net job creation II: Creating the net job

## 13.1 Main settings

The most important settings of a net job are made in the main settings tab, which is shown in the screenshot below:



The main settings include the net job name (the default name for new net jobs can be changed in the options), an optional net job color (which will be used to color the net job in the main net job list), the priority, as well as the urgency flag, the renderer and version to use, as long as the architectures to utilize, and frame splitting, image slicing and additional splitting. The parallel renderings count per client specifies how many chunks should be rendered on a single client in parallel. You can also select which pools and clients should work on this job using the pool and client tree.

**Note:** Whether frame splitting, image slicing and additional splitting are available depends on the selected renderer.

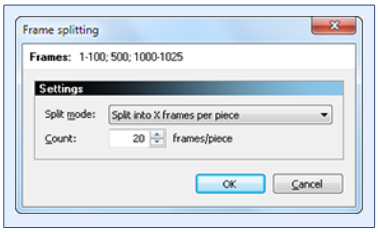
### Chunk dispatching

The chunk dispatching options allow you to specify the order in which the various chunks of the net job should be rendered. The following table lists all available dispatch orders:

| Mode | Description |
|---|---|
| Forward | Renders the chunks from start to end. |
| Backward | Renders the chunks from end to start. |
| Evenly distributed | Renders the chunks evenly, resulting in a more and more detailed overview of the rendering sequence. |
| Random | Renders the chunks randomly. |

The option render first and last chunk first can be used to always render the start and end chunk before any other chunk.

## 13.1.1 Frame splitting




Frame splitting is used to divide the frames of an animation into multiple, smaller pieces, which will then be individually rendered by your clients. The frame splitting dialog is shown on the right. RenderPal V2 supports two splitting modes:

### Split into X total pieces

The entire animation will be split into a total of X pieces. With this method, you have control over the number of resulting chunks.

### Split into X frames per piece

Each resulting piece will have the same number of frames (depending on the divisibility of the frame range). With this method, you have control over the ”size” of each chunk.

**Note:** The ”By frame” value (if specified) will be taken into account when splitting is performed.

**Important:** Render set values

For frame splitting to work, a frame list has to be specified in the render set.

## 13.1.2 Image slicing

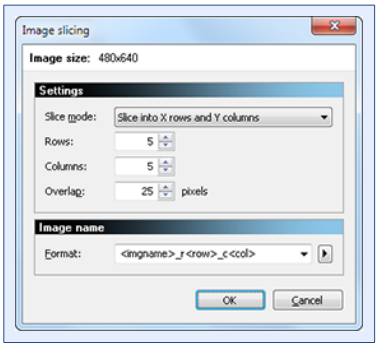



Image slicing cuts down a large image into smaller pieces, which will then be individually rendered by your clients. The image slicing dialog is shown on the right. RenderPal V2 supports two slicing modes:

### Slice into X rows and Y columns

The image will be sliced down into the specified number of rows and columns. With this method, you have control over the number of resulting pieces.

### Slice into pieces of a constant size

The image will be sliced down into pieces of the specified size. With this method, you have control over the dimensions of each piece.

Both methods offer an Overlap value, which can be used to enlarge each piece by the specified amount of pixels. This

can be useful when using effects like glow (to avoid errors at the borders when assembling the image).

### Image name format

Since the image will be sliced down into multiple pieces, and thus resulting in multiple files, each file has to be named properly; the image name format defines how the new filenames should look like. RenderPal V2 offers several placeholders that can be used: `<imgname>` is the original output file specified in the render set (adding this to the image name format is mandatory); `<current>` and `<total>` represent the current piece number and the total amount of pieces; `<row>` and `<col>` represent the current row and column numbers.

**Example:** Image slicing

```
Image name format:  <imgname>_<current>_of_<total>
Total image size:   640x480 pixels
Slicing mode:       Slice into pieces of constant size
Piece size:         320x240 pixels
Output filename:    MyImageName

Result:
  MyImageName_1_of_4
  MyImageName_2_of_4
  MyImageName_3_of_4
  MyImageName_4_of_4
```

**Note:** If an image region is specified in the render set, this will be taken into account when slicing.

**Important:** Render set values

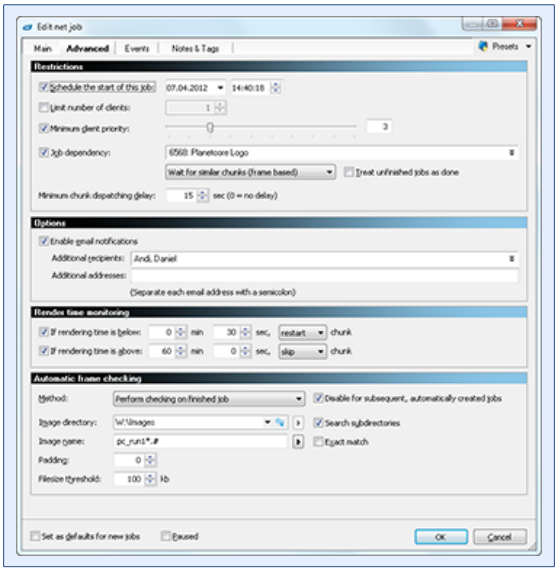
For image slicing to work, the image width and height, as well as the output file, have to be specified in the render set.

## 13.1.3 Additional splitting

Renderers can also split a job by further settings (like cameras or layers). Simply check a setting which should be used for splitting in the additional splitting list.

## 13.2 Advanced settings

The advanced net job settings offer to control net jobs, further refine how they should be dispatched and to perform certain actions if the render times of a chunk are too long or not long enough. It is also possible to let RenderPal V2 automatically check for any missing frames, which will then be resubmitted for rendering. The following screenshots show the advanced settings tab:



### Net job restrictions

A net job can be restricted in various ways. It is possible to schedule the start of the job, to limit the number of clients that will simultaneously work on the job or to set the minimum client priority (this can be used to only let powerful machines render the job, for example). A job can also be rendered only after certain other jobs or chunks of those jobs have been finished; this way, you can make the job dependent on other jobs. The dependent job can either wait for the entire job to be finished or for similar chunks. The latter supports three modes: ID-, frame- and image slice-based matching; the net job will match its own chunks based on the selected mode against its dependent jobs and render only those chunks that have been finished on all dependent jobs. A minimum delay between chunk dispatches can also be set.

### Automatic chunk resubmission

Often, renderings will either take too long (because they got trapped in an endless loop, for example), or they just finish way too quickly, indicating that something went wrong. RenderPal V2 can automatically resubmit the corresponding job in these cases.

## 13.2.1 Automatic frame checking

Since renderers do not report whether they could render a specific frame or not, RenderPal V2 offers a powerful, yet easy to use mechanism to automatically check for any missing frames. Automatic frame checking can either be performed after every chunk or after the entire job has been rendered. When new net jobs are created due to missing frames, the original net job will be set to be dependant on the newly created jobs. This way, the original job will only be finished when all missing frames have been rendered properly. By default, these newly created jobs will inherit the frame checking settings of the original net job. To disable this behavior (to avoid a large amount of automatically created net jobs if the new jobs will fail again), simply turn on the option Disable for subsequent, automatically created jobs. However, RenderPal V2 will limit the ”depth” of such net job recreations to avoid endless loops.

To use automatic frame checking, only a few settings have to be made. First, RenderPal V2 needs to know where the images will be located (Image directory); an option is also available to let the frame checker search in all subdirectories. The most important option is the Image name, which tells the frame checker how the filenames should look like. Other options include a frame number padding and a filesize threshold (files that are smaller than the specified size will be considered as missing). In most cases, the default settings, which are extracted from the underlying render set, should be sufficient.

### The image directory

When using automatic frame checking for every chunk, it is possible to use placeholders for parameters used for additional splitting in the image directory, as well as the scene name. These placeholders will then be replaced by the value of those parameters in the current chunk.

**Example:** Image directory

```
Add. splitting enabled for parameter Camera
Image directory: W:\Output\$(camera)
Cameras in the render set: cam1, cam2

The frame checker will replace $(camera) with either cam1 (W:\Output\cam1) or cam2
(W:\Output\cam2) - depending on which camera is used in that particular chunk - and
search for images in that directory.
```

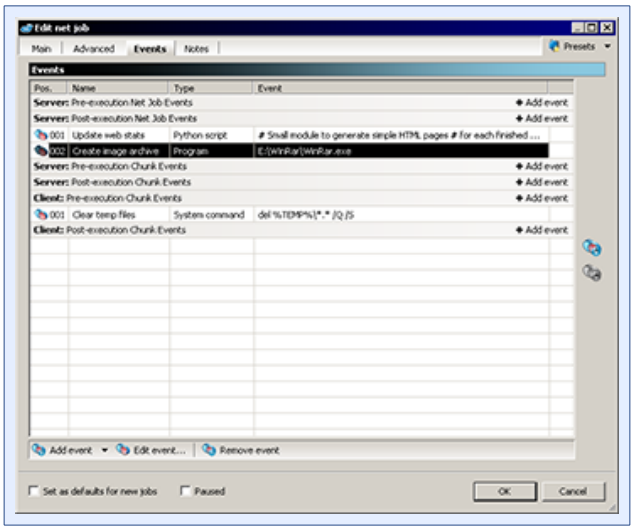
### The image name

To see whether a file is missing or not, the frame checker needs to know how the filenames look like. This is where the image name comes into play. The image name contains the actual filename, as long as a placeholder for the frame number (`#`). This placeholder tells the frame checker where in the filename the frame number should be. The common asterisk (`*`) can also be used in the image name. An option whether the frame checker should search for partial (the default behavior) or exact matches also exists.

This might sound confusing at first, but is easier than it seems. All you need to know is how your resulting files will be named. Imagine your files will be called `MyImage_001.jpg`, `MyImage_002.jpg` and so on. `MyImage` is the actual output file, `001` is the frame number and `.jpg` is the file ending. The resulting image name format is `MyImage_#`. All we did was to replace the frame number with the frame number placeholder (`#`). The file ending can be left out as long as Exact match is unchecked (which is always recommended).

## 13.3 Net job events

Net job events can be used to execute a program, a system command or a Python script when certain events occur; RenderPal V2 currently supports six different events. Net job events are a powerful feature, and they can be used for a huge variety of tasks, ranging from the more obvious ones, like cleaning temporary files, copying images around or creating HTML reports using Python to the not so obvious ones. The net job events tab can be seen in the following screenshot:




Net job events come in two flavours: server-side events, which will be executed on the server machine and client-side events, which will be executed on the client machine. The following tables list all available events:

### Server-side events

| Scope | Description |
|---|---|
| Pre-execution Net Job events | Executed before the net job is started (in other words, this event will be executed immediately after the job has been created or restarted). |
| Post-execution Net Job events | Executed after the entire net job has been rendered. |
| Pre-execution Chunk events | Executed before a chunk is sent to the client. |
| Post-execution Chunk events | Executed once a chunk has been rendered. |

### Client-side events

| Scope | Description |
|---|---|
| Pre-execution Chunk events | Executed before the client renders the job. |
| Post-execution Chunk events | Executed after the client has rendered the job. |

All post-job events offer options whether they should be executed if the job/chunk succeeded and/or if it failed. For every event type (programs, system commands and Python scripts), you can create presets that can be later reused. This way, you can quickly store your most frequently used events.

**Note:** Path maps are applied to all event contents.

### 13.3.1 Event variables

There are several variables/placeholders you can use in events. These include variables from the net job, the net job chunk (for chunk events) and the underlying render set. The basic syntax of a variable is as follows:

```
$(<Type>.<Name>)
```

`<Type>` can either be `NetJob`, `NetJobChunk` or `RenderSet`, denoting the scope of the variable. `<Name>` is the actual variable name. Here are two variable examples:

```
$(RenderSet.SceneFile)
$(NetJob.Renderer)
```

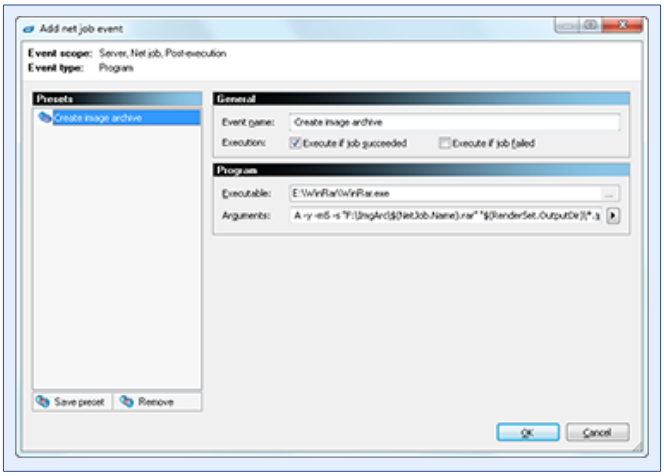
**Important:** Variables are only placeholders

Before executing the event, all variables will be replaced by their actual values, meaning that they act as placeholders. This has the consequence that you need to add quotes (if necessary) manually. For example, if you want to use the scene file in a command-line, you should write something like this: `-someswitch "$(RenderSet.SceneFile)"`. This also applies to Python scripts.

A list of all available variables can be found using the drop-out button found in the net job event dialog.

### 13.3.2 Program events

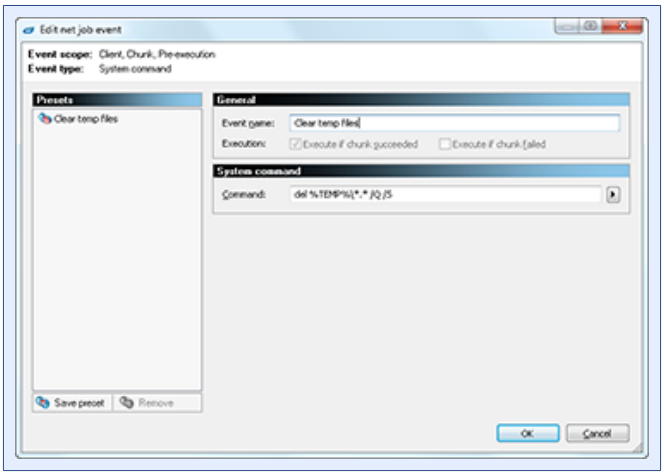
Program events allow you to launch external applications with an additional command-line. The following screenshot shows the program event dialog:




A program event consists of the actual program to launch (Executable), as well as an additional command-line (Arguments). Event variables can be used in the arguments only.

### 13.3.3 System command events

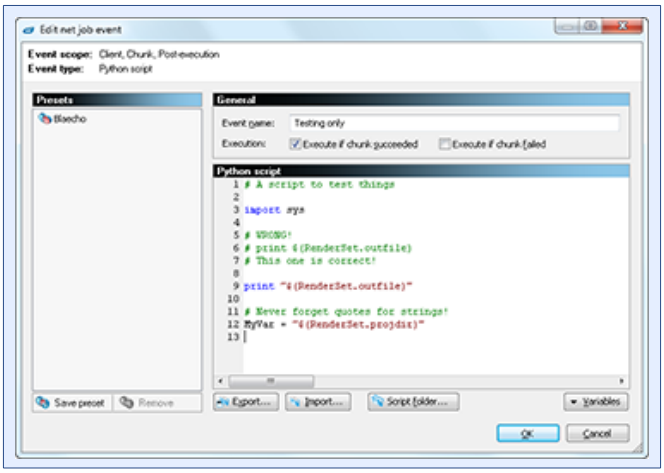
System command events can be used to execute simple system commands. The following screenshot shows the system command event dialog:




A system command event consists of just the Command itself. You should always keep in mind that system commands differ from operating system to operating system.

### 13.3.4 Python script events

Python scripts are without doubt the most powerful and flexible event type. Almost any task can be achieved via Python script events, but, as a matter of fact, they also require a decent knowledge of the Python scripting language. Here is a screenshot of the Python script event dialog:




The Python script solely consists of the script itself; just like in all events, event variables can be utilized in Python scripts as well. It is also possible to load and save script files. Script files should be stored in the user script folder, which can be opened by clicking Script folder. It is recommended to keep the entered script as short as possible and rather use external files (via `import`) instead.

**Important:** Event Variables & Python variables

As mentioned earlier, event variables are only placeholders; this also holds true for Python script. If you want to use a string variable, like the output directory, somewhere, you have to enclose the variable with quotes. If you frequently use a certain event variable, you should create a ”real” variable like this:

```
MyPalVar = ”$(RenderSet.OutputDir)”
```

Leaving out the quotes would result in errors since `$(RenderSet.OutputDir)` is a string. For other variable types, like integers and floats, quotes have to be left out:

```
MyOtherVar = $(NetJobChunk.StartFrame)
```

Python scripts have an nearly endless potential; a few examples and ideas will be given in a later chapter. To be able to use this feature of RenderPal V2, you’ll need a good understanding and knowledge of the Python scripting language. Teaching anything about Python would be beyond the scope of this manual, but the internet is full of good resources about Python. The first place to start is the Python website itself, which can be found at http://www.python.org. There are also many good books about this language.

Due to its immense popularity, libraries for almost everything are available for Python (most of them already included). Another great benefit of Python scripts is their operating system independence.

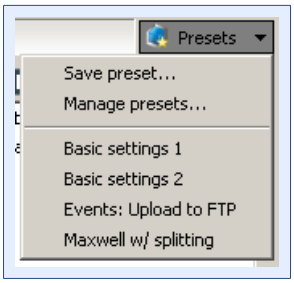
**Note:** A Python installation is not required to use Python script events; RenderPal V2 comes with all Python runtimes and libraries.

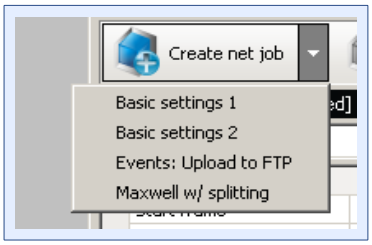
## 13.4 Notes & Tags

A net job can have some arbitrary, user-defined textual notes and tags. The notes are purely informational, while the tags can be used to quickly filter the main net job list. When entering tags in the main net job list, only jobs that contain at least one of the entered tags (tags are not case-sensitive) will be shown. Tags are an easy and fast way to further organize your net jobs.

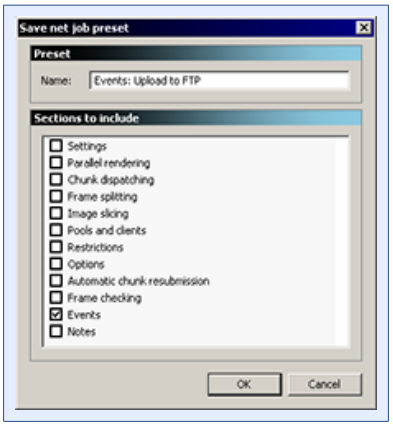
## 13.5 Net job presets

Net job presets are an easy way to save various net job settings for later use. All settings of a net job can be saved in a preset, including the pool and client selection, all settings and net job events. Net job presets are created and managed using the Preset button in the net job editor, as seen in the first screenshot below. New net jobs can also be created based on an existing net job preset, which can be seen on the second screenshot:

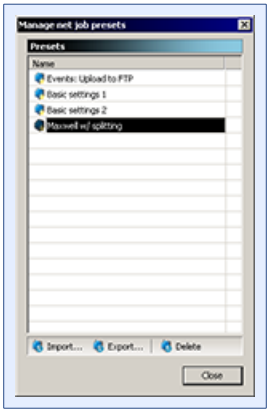



When creating a new preset, you can select which setting groups should be included:




This allows you to create presets that only contain certain settings, like pure ”event-only” presets, for example. When applying a preset, only the previously selected settings will be used, all other settings will be left untouched. Presets can also be imported, exported and deleted using the Manage presets dialog:




**Note:** All presets are saved in the **Presets/NetJobs** directory. This is important to know when you want to use presets in conjunction with the console Remote Controller (which will be described later), as preset files should be copied to the appropriate console Remote Controller directory first (either its Presets/NetJobs or root directory).

# 14. Renderer management

**Applies to:** Server, Remote Controller

The renderer management is used to create, edit and remove renderers in RenderPal V2. It can be both accessed from within the server or the remote controller (if the necessary user right is granted). All aspects can be edited using this powerful editor - you'll never need to edit any "source" files.

When changing the renderer pool in any way, all remote controllers will be automatically retrieve the changes; there's no need to update anything manually. The console remote controller will also always download the renderer pool when connected.

The entire renderer system, including its editor, has been explained in full detail in chapter 3.

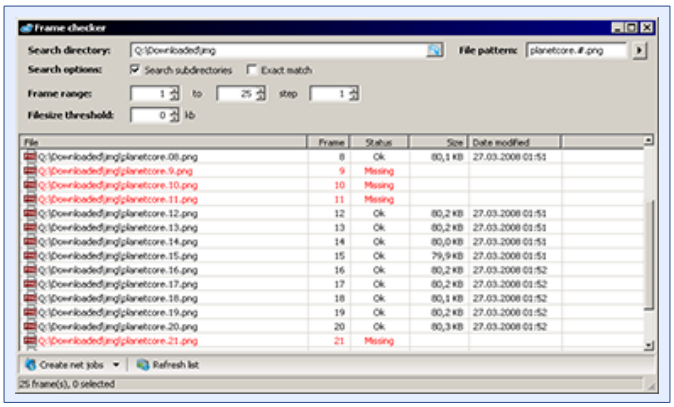
If you've used RenderPal V2 prior to version 2.4, you might miss the custom renderers. These have been superseded by the new renderer system, which allows you to create renderers in a far more flexible, unprecedented way. You might also miss the Additional commands parameter; this is now unnecessary, as you can add further parameters to a renderer whenever needed.

**See also:** The renderer system

# 15. Manual frame checking

**Applies to:** Server, Remote Controller

The manual frame checker can be used to manually perform frame checking on a net job, a single net job chunk or any arbitrary image sequence. The frame checker dialog is shown below:




When performing manual frame checking, which can be achieved by using the Perform frame checking command of the net job/net job chunk context menu, you can quickly create new net jobs for any missing frames. When new net jobs are created due to missing frames, the original net job will be set to be dependant on the newly created jobs. This way, the original job will only be finished when all missing frames have been rendered properly.

The manual frame checker works the same way as the automatic frame checking for net jobs. The search directory will be searched for images matching the file pattern; if enabled, subdirectories will be included as well.

When performing manual frame checking on a net job, RenderPal V2 will try to fill out all settings on its own, based on the render set settings.

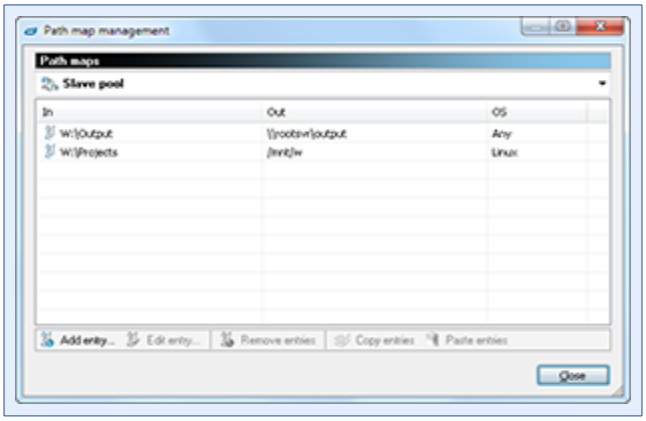
For more details about frame checking, refer to the automatic frame checking section in the net job editor chapter.

**See also:** Automatic frame checking

# 16. Path map management

**Applies to:** Server, Remote Controller

Path maps are used to map an incoming path or filename to something else. In simple terms, an original text is replaced by a new text (since you can actually not only map paths and filenames). The screenshot below shows the path map management:




Path maps can be specified per client, per pool (so that the map applies to all clients in that pool) or globally (applies to all clients); they are applied to all paths and filenames in RenderPal V2, as well as to net job event contents. The path map management dialog also allows you to quickly copy and paste existing entries.

**Important:** Paths and filenames inside scenes

Paths and filenames inside your scenes (like textures etc.) will **not** be translated.

Path maps can be used for a variety of things. One of their main uses is to map a path from one operating system to another. Other uses might include mapping a standard path to an UNC path (or vice versa) or to map net job event parameters. Entries can be configured to be applied to a specific operating system only, which further eases working with multiple operating systems.

Here is an example showing how path map entries work:

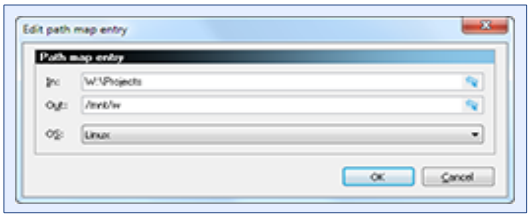
**Examples:** Path maps

```
This example maps a local (but shared) path to a mapped network drive:

Scene file:           F:\Shared\Scenes\MyScene.mb
In-path:              F:\Shared\Scenes
Out-path:             N:\Scenes
Resulting scene file: N:\Scenes\MyScene.mb
```

**See also:** Path maps

## 16.1 Path map entries

A path map can contain multiple entries; the order in which they are applied is arbitrary. Each entry consists of an in-path (In), as well as an out-path (Out). You can further select to which operating system (OS) the entry should be applied. The path map entry dialog is shown on the right.



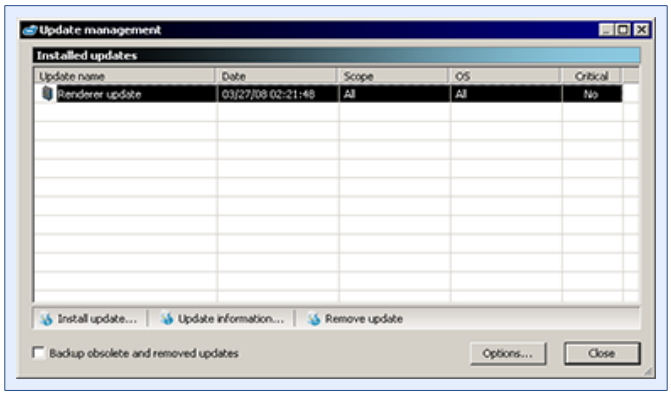

The operating system selection allows you to let this entry be applied only when the job is picked up by a client running under that operating system.

Entries must not necessarily be paths or filenames; you can enter any text you like. This means that you can also create simple variables with path maps.

# 17. Update management

**Applies to:** Server

In RenderPal V2, updates don't have to be installed manually anymore - the RenderPal V2 Server takes care of this for you. All you have to do is to add new updates via the update management, which can be seen in the following screen-shot:




Most of the times, the updates you will install are created by us. It is, however, also possible to create and deploy your own updates; this will be covered in a later chapter.

Updates always come as zip files. There is no need to unpack these files, all you have to do is to install it by either using the Install update button (and selecting the update file) or by simply dragging the file into the update management. The location of the update file doesn't matter, as RenderPal V2 will automatically copy it to the correct directory.

Once an update has been added in the server, RenderPal V2 will begin its deployment; there is nothing more you have to do. At first, the server will be updated. Clients and remote controllers will be automatically updated the next time they connect to the server. In the case of critical updates, the server will be restarted; this also causes any connected clients and remote controllers to be disconnected (this also ensures that they will be updated before they can be used again).

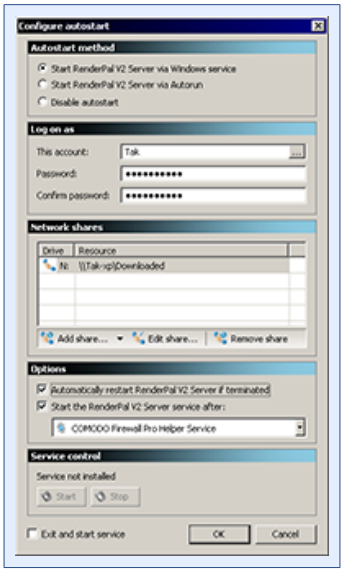
It is also possible to let RenderPal V2 create backups of removed and obsolete updates (an update might render a previous obsolete, thus removing it from the update list). These will be located in the **Updates\Backup** directory.

**Note:** You do not have to keep the original update file; it will be automatically copied into the correct server directory.

# 18. Autostart

**Applies to:** Server, Client

Both the RenderPal V2 Server and the RenderPal V2 Client can be configured to be automatically started on system startup. The following screenshot shows the autostart configuration dialog:

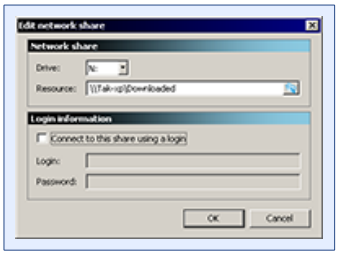



RenderPal V2 offers two autostart methods: via Windows service or Autorun. Using the service method has the benefit that no user needs to be logged in for the server/client to run; when Autorun is being used, the server/client will only be started when the (current) user logs in.

When using the service method, several options are available. You can (and should) specify an user account under which the service should run; if left empty, the **Local Service** account will be used. Under Windows XP and later, services do not have access to existing network shares; RenderPal V2 can, however, create its own mappings. This will be covered in the next section. The service can watch if the server/client has been terminated (due to a crash, for example) and restart it automatically; enable this option by checking Automatically restart RenderPal V2 Server/Client when terminated. It is also possible to let the service be started after another service (**service dependency**); this can be useful to avoid conflicts with personal firewalls, for example.

If the server/client is running as a service and you want to terminate it as well as stop the service at the same time, you can use the Shutdown server/client function found in the File menu; this will stop the service and exit the application (without using this function, the server/client would be restarted by the service if the corresponding option is enabled).

## 18.1 Network shares




Since Windows XP, mapped network drives are no longer available to services (even if they are running under an user account). Due to these changes, RenderPal V2 offers the ability to create its own mappings, which will then be available to RenderPal V2 and all renderers. This way, you can let RenderPal V2 simply recreate your network mappings.

The network share dialog is shown on the right. A share entry consists of the drive letter, as well as the network resource (usually an UNC path). You can also optionally specify a login to use for the share.

When selecting the service autostart method, RenderPal V2 will scan for your existing shares and recreate them automatically; this can also later be done manually.

**Note:** This feature can, of course, also be used to not just recreate existing shares, but to create new shares that should just be used for RenderPal V2 and the renderers.

# 19. Logging

**Applies to:** Global

In RenderPal V2, there are two types of logging: event logging and output logging. Event logging refers to the logging of the various events that occur in RenderPal V2, like network events, net job and net job chunk status changes, when an user modifies certain areas and so on. Output logging refers to the logging of any textual output produced by the renderers, net job events and so on.

There are many options related to logging. This includes the limitation of both the event and output log (to restrict memory usage), deletion of old logs and more. Log files are saved inside the **Logs** directory (if not overridden); you can always take a look at all log files using the **Logs** menu.

RenderPal V2 creates an event and (global) output log for each day; the client also creates a separate output log for every chunk it renders.

### Chunk history

Every net job chunk keeps a history of its events (like errors, status changes, messages from the renderer and so on). This history can be viewed using the server or remote controller. This is especially useful when searching for errors. The last history entry is also shown in the net job chunk list.

### Remotely view logs

RenderPal V2 also offers the ability to conveniently view event and output logs of all clients using the server or remote controller (server logs can also be viewed using the remote controller); it is even possible to view the output of specific net job chunks.

### Output buffers

For all textual output, RenderPal V2 uses so-called output buffers. For every job the client renders, a new output buffer will be created; an output buffer acts like a scope where all output will go. The maximum number of output buffers can be limited to restrict memory consumption.

**Usage Tip:** Locating renderer problems

Logging is not just an informative feature, but is also important when it comes to solving renderer related issues. It is most likely that a renderer will tell you what the exact cause of the problems (the most common problem being no rendered images) is, so taking a look at the output log of that particular client is always the first thing to do in such cases.

# 20. Options

**Applies to:** Global

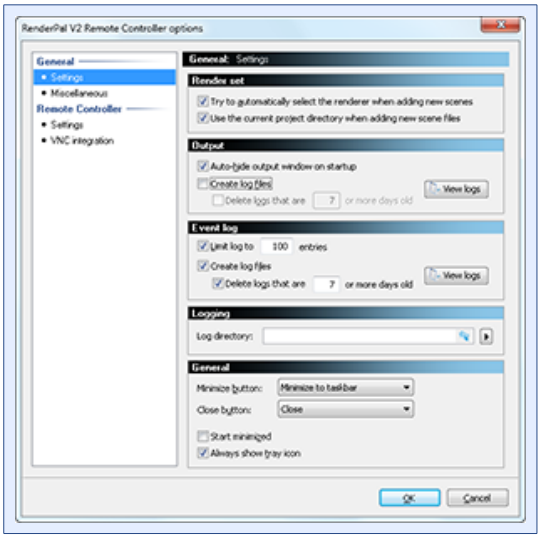
RenderPal V2 offers several options that allow you to tweak how the various aspects of the programs work. Even though all defaults were chosen carefully and should be sufficient in most cases, there still might be some settings you want to change. This chapter will explain the various options in detail. We will begin with general options which are common in all components of RenderPal V2; we will then continue with the component-specific options.

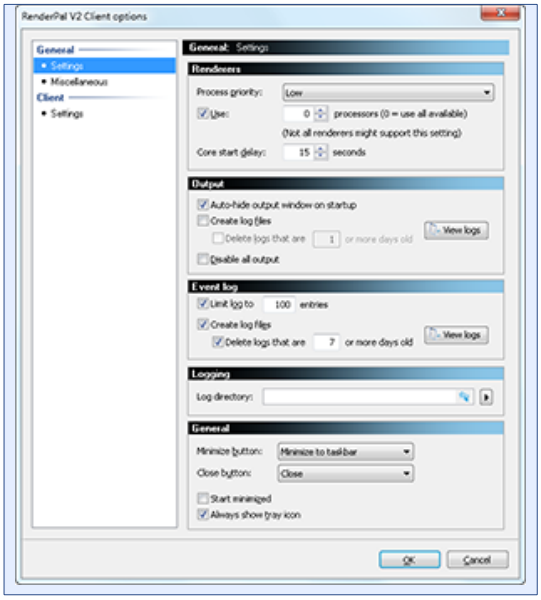
## 20.1 General options

The general options include the general and renderer settings.

### 20.1.1 General settings

The general settings are the same in both the server and remote controller; the client offers slightly different options. The screenshot on the left shows the server and remote controller options, while the right one shows the client options:




#### Server and Remote Controller

| Name | Description |
|---|---|
| **Render set** | |
| Try to automatically select the renderer when adding new scenes | If enabled, RenderPal V2 will try to select the correct renderer when adding new scenes to a render set (based on the file ending). |
| Use the current project directory when adding new scenes | If enabled, the ”Open files” dialog will automatically jump to the project directory currently set in the render set |
| **Output window** | |
| Auto-hide on startup | Turning this on will hide the output window when RenderPal V2 is started. |
| Create log files | If enabled, all output will be written to log files. |
| Delete logs that are X or more days old | If enabled, logs that are X or more days old will be automatically deleted (to save disk space). |
| **Event log** | |
| Limit log to X entries | This setting allows you to limit the maximum number of entries in the event log (to save memory). |
| Create log files | If enabled, all events will be written to log files. |

| Name | Description |
|---|---|
| Delete logs that are X or more days old | If enabled, logs that are X or more days old will be automatically deleted (to save disk space). |
| **Logging** | |
| Log directory | Overrides the default log directory with a custom one. |
| **General** | |
| Minimize button | Specifies the behaviour of the minimize button (minimize to the task bar or to the system tray). |
| Close button | Specifies the behaviour of the close button (close, minimize to the task bar or to the system tray). |
| Start minimized | Starts RenderPal V2 minimized |
| Always show tray icon | If enabled, the RenderPal V2 tray icon will always be shown |

### Client

| Name | Description |
|---|---|
| **Renderers** | |
| Process priority | This setting lets you select with which process priority the renderers should be started. **Below normal** or **Low** are recommended; this way, the system will still be responsive. |
| Use X processors | If enabled, the specified number of processors will be used for rendering. Note that this setting might not work with all renderers. |
| Core start delay | Specifies the delay between the start of parallel renderings. This is used to prevent excessive system and network traffic. |
| **Output window** | |
| Auto-hide on startup | Turning this on will hide the output window when RenderPal V2 is started. |
| Create log files | If enabled, all output will be written to log files. |
| Delete logs that are X or more days old | If enabled, logs that are X or more days old will be automatically deleted (to save disk space). |
| Disable all output | This will completely turn off all textual output; not recommended. |
| **Event log** | |
| Limit log to X entries | This setting allows you to limit the maximum number of entries in the event log (to save memory). |
| Create log files | If enabled, all events will be written to log files. |
| Delete logs that are X or more days old | If enabled, logs that are X or more days old will be automatically deleted (to save disk space). |
| **General** | |
| Minimize button | Specifies the behaviour of the minimize button (minimize to the task bar or to the system tray). |
| Close button | Specifies the behaviour of the close button (close, minimize to the task bar or to the system tray). |
| Start minimized | Starts RenderPal V2 minimized |
| Always show tray icon | If enabled, the RenderPal V2 tray icon will always be shown |

## 20.1.2 Miscellaneous

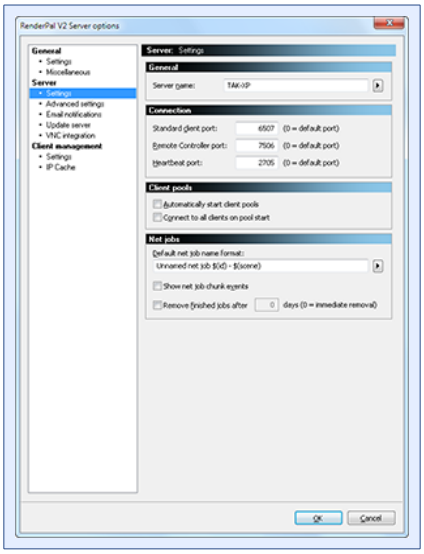
The Miscellaneous page allows you to reset all message boxes, so that previously hidden ones will be shown again. In RenderPal V2 Server, it is also possible to reset the highest net job ID back to 1 (only when the net job queue is empty) here. You can also reset all saved windows positions, which can be useful when using multiple monitors and some windows appear in unreachable areas.

## 20.2 Server options

The RenderPal V2 Server offers the most options. They include basic and advanced settings, e-mail notifications, update server configuration and client management related options.

### 20.2.1 Basic settings

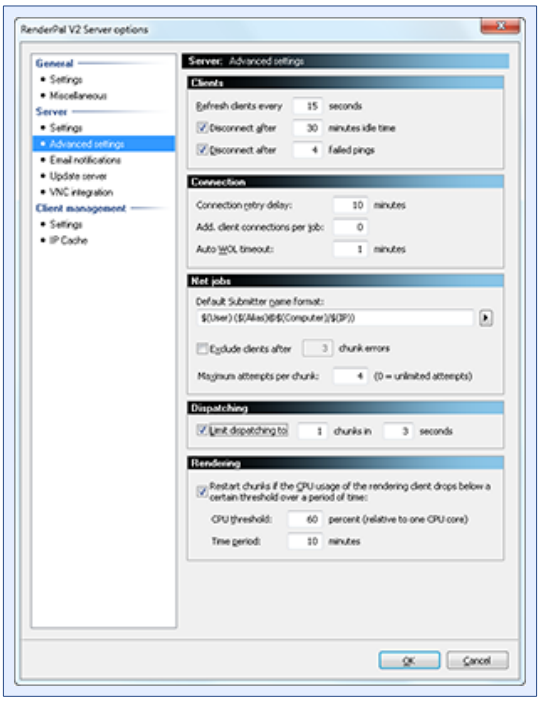
The basic server settings are shown in the screenshot below:




| Name | Description |
|---|---|
| **General** | |
| Server name | The name/alias of the server. Use `%COMPUTERNAME%` as a placeholder for the computer name. |
| **Connection** | |
| Standard client port | The standard port for all clients. This should only be changed if necessary. |
| Remote controller port | The port remote controllers will connect to. This should only be changed if necessary. |
| Heartbeat port | The listening port for heartbeats sent by the clients. This should only be changed if necessary. |
| **Client pools** | |
| Automatically start clients pools | If enabled, all client pools will be started on server startup; newly created pools will also automatically be started. |
| Connect to all clients on pool started | When enabled, the server will connect to all clients inside a pool when it is started. |
| **Net jobs** | |
| Default net job name format | This format will be used for all new net jobs. You can use various placeholders, which can be added using the drop-out button. |
| Show net job chunk events | If enabled, events about all net job chunks (chunk finished, failed etc.) will be logged. |
| Remove finished jobs after X days | Removes finished jobs after they are at least X days old; if X is 0, finished jobs will be removed immediately. |

## 20.2.2 Advanced settings

The advanced server settings are shown in the following screenshot:

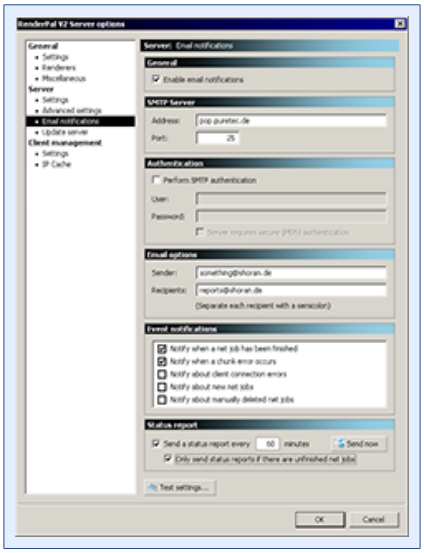



| Name | Description |
|---|---|
| **Clients** | |
| Refresh clients every X seconds | Sets the interval in which the server should send pings to the clients. This will also update the various client statistics, like CPU and memory usage. |
| Disconnect after X minutes idle time | If enabled, clients will be disconnected after the specified amount of idle time. |
| Disconnect after X failed pings | If enabled, clients will be disconnected if X pings failed (no answer received). |
| **Connection** | |
| Connection retry delay | Sets the delay between a new client connection retry after a failed one. |
| Add. client connections per job | By default, a job will only request the necessary amount of clients for rendering. This option allows you to let the server connect to a certain number of additional clients, which is helpful to circumvent waiting times during connection attempts. |
| Auto WOL timeout | When auto WOL is activated for a pool, this specifies the time after which a WOL is seen as failed. |
| **Net jobs** | |
| Default Submitter name format | This is the name format that will be used for the submitter name shown in the net job list. You can use various placeholders, which can be added using the drop-out button. |
| Exclude clients after X chunk errors | If enabled, clients will be excluded from a net job if it caused X or more chunk errors for that job. |
| Maximum attempts per chunk | The maximum number of attempts for a net job chunk; if this maximum is reached, the chunk will not be picked up anymore. |
| **Dispatching** | |
| Limit dispatching to X chunks in Y seconds | If enabled, the server will only send X chunks at maximum in Y seconds. This can be used to reduce network load. |

| Name | Description |
|---|---|
| **Rendering** | |
| Restart chunks if the CPU usage of the client drops below a certain threshold over a period of time | If a rendering crashes and hangs, it will never finish, wasting important resources, and the net job will never be done. Usually, such a hanging rendering will use almost no CPU. Using this option, it is possible to let RenderPal V2 monitor the CPU usage of its clients, and if the CPU usage drops below a certain threshold over a period of time, the chunk will be cancelled and it can be picked up again. |
| CPU threshold | The minimum CPU threshold in percent; the threshold is relative to a single CPU core. |
| Time period | The period (in minutes) over which the threshold has to be below the defined value in order to cancel the current chunk of a client. |

### 20.2.3 E-mail notifications

The RenderPal V2 Server can be configured to automatically send e-mails about various events, like a finished or erroneous net job, and to periodically send status reports. The e-mail notification options can be seen in the following screenshot:



| Name | Description |
|---|---|
| **General** | |
| Enable e-mail notifications | Enables or disables e-mail notifications |
| **SMTP Server** | |
| Address | This is the address of the SMTP server for sending e-mails. |
| Port | This is the SMTP port; the default port is 25. |
| **Authentication** | |
| Perform SMTP authentication | If your SMTP server requires authentication, enable this option. |
| User | The e-mail account user name. |
| Password | The e-mail account password. |
| Server requires secure (MD5) authentication | Enable this option if the SMTP server requires secure, MD5-based authentication. |
| **E-Mail options** | |
| Sender | The sender e-mail address which the notifications should be sent from. |
| Recipients | A list of global recipients that will receive notifications from all net jobs. This can be left empty if you only want to send notifications to individual users about their own net jobs. |

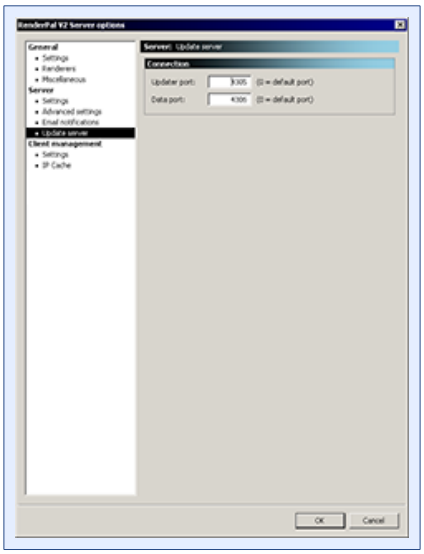
| Name | Description |
|---|---|
| **Event notifications** | |
| Notifications | This list lets you select which notifications should be sent. |
| **Status report** | |
| Send a status report every X minutes | If enabled, a status reports about all clients and net jobs will be send every X minutes. |
| Only send status reports if there are unfinished net jobs | By default, status reports will be sent even if the render farm is idle; turn this option on to send them only if there are unfinished net jobs. |

**Note:** User accounts also offer an e-mail field. If provided, notifications about their own net jobs will be e-mailed to them directly.

You should contact your e-mail provider about the various settings.

### 20.2.4 Update server settings

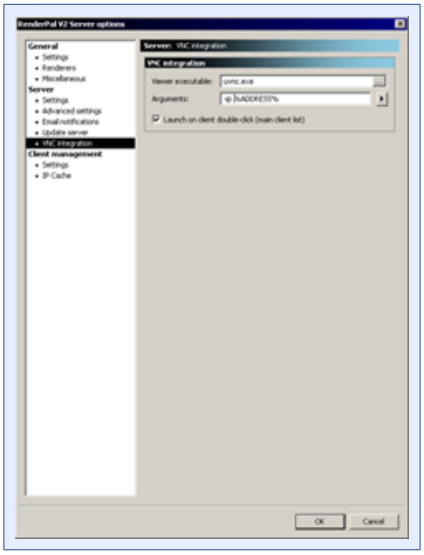
The update server in RenderPal V2 utilizes the FTP protocol. It uses two ports, one for incoming connections, one for the actual data transfer. The following screenshot shows the update server settings:



| Name | Description |
|---|---|
| **Connection** | |
| Updater port | This is the port the updater will listen on for incoming connections. |
| Data port | This is the port that will be used for the actual data transfer. |

### 20.2.5 VNC integration settings

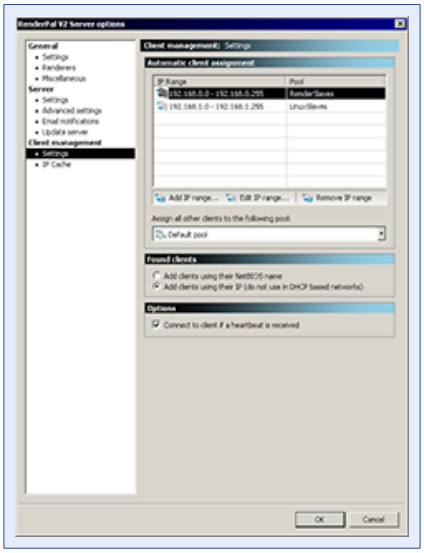
The VNC integration allows you to specify a VNC viewer application that can be launched for a given client. The following screenshot shows the VNC integration settings:



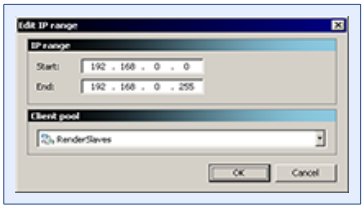
| Name | Description |
|---|---|
| **VNC integration** | |
| Viewer executable | The executable file of the VNC viewer to use |
| Arguments | The command-line arguments passed to the viewer; use `%ADDRESS%` to pass the client's address. |
| Launch on client double-click | When enabled, a double-click on a client in the main client list will launch the VNC viewer for that client (if disabled, the viewer has to be started using the client's context menu). |

### 20.2.6 Client management settings

The RenderPal V2 Server offers several settings related to the client heartbeat feature; these can all be found in the Client management settings, which are shown below:



When a new client is found by the server, it can be automatically assigned to a client pool according to its IP. It is possible to define multiple IP ranges that should be checked against when a new client is found. These ranges may not overlap, though a client pool can be the target for multiple IP ranges. The screenshot on the right shows such an IP range entry. Clients that do not fall within any specified IP range can be optionally assigned to a default pool.

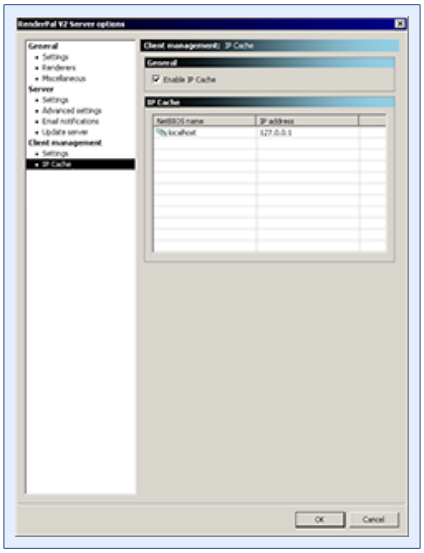



There are also some options dealing with automatically found clients:

| Name | Description |
|---|---|
| **Found clients** | |
| Add to known clients and assign to pool | If enabled, clients found via heartbeat will be automatically added and assigned to the selected pool. |
| Add clients using their NetBIOS name/ IP | Choose whether to add clients using their IP address (recommended for static networks) or NetBIOS name (recommended for dynamic networks). |
| **Options** | |
| Connect to client if a heartbeat is received | If enabled, the server will connect to the client if a heartbeat is received. |

### 20.2.7 IP cache

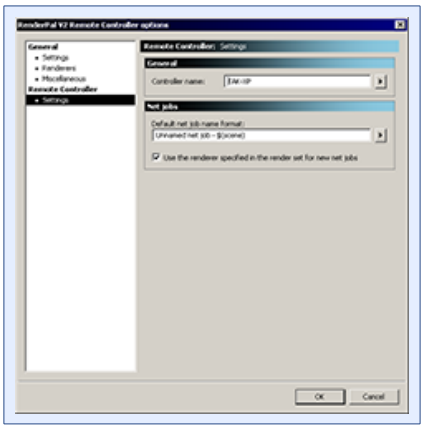
The IP cache can be used to speed up name lookups in RenderPal V2. When using a NetBIOS name, Windows has to look up the corresponding IP first; this can take quite a while. The RenderPal V2 Server offers its own IP cache, where it stores the NetBIOS name as long as its corresponding IP. If an entry for a NetBIOS name exists in the cache, the associated IP will be used directly. The IP cache dialog can be seen below:




The only option is to either enable or disable the IP cache. You can also delete specific entries from the cache.

## 20.3 Remote Controller options

The RenderPal V2 Remote Controller only offers a few settings which can be seen in the following screenshot:

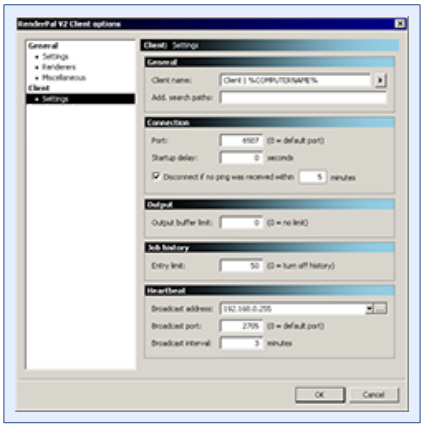


| Name | Description |
|---|---|
| **General** | |
| Controller name | The name/alias of the remote controller; this is used in the submitter column of the net job list, for example. Use `%COMPUTERNAME%` as a placeholder for the computer name. |
| **Net jobs** | |
| Default net job name format | This format will be used for all new net jobs. You can use various placeholders, which can be added using the drop-out button. |
| Use the renderer specified in the render set for new net jobs | If enabled, the renderer selected in the render set will be used for new net jobs; otherwise, the renderer set as the net jobs default will be used. |

**Note:** The Remote Controller also offers VNC integration; the settings are the same as in the Server.

## 20.4 Client options

The client offers several options, which can be seen in the following screenshot:

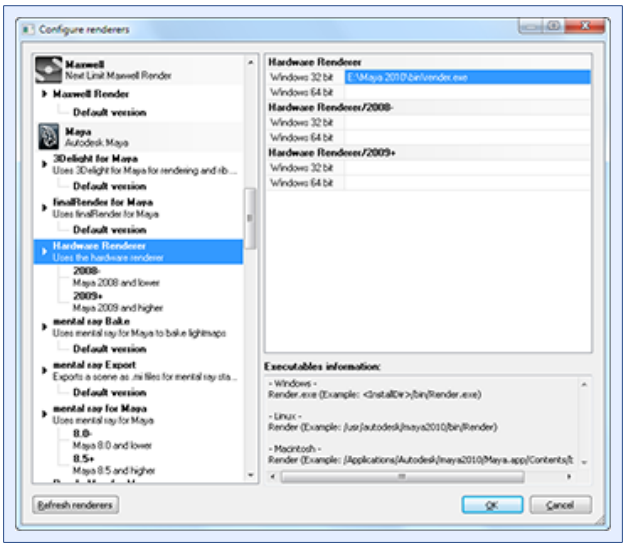


| Name | Description |
|---|---|
| **General** | |
| Client name | The name/alias of the client. Use `%COMPUTERNAME%` as a placeholder for the computer name. |
| Additional search paths | Enter any additional paths here that the renderers should use when searching files etc. |

| Name | Description |
|---|---|
| **Connection** | |
| Port | The client will listen on this port for incoming server connections. |
| Startup delay | If enabled, the client will wait for X seconds before accepting connections. This is useful when used in combination with autostart; this way, the client gets some time for the entire system to properly load before starting to render. |
| Disconnect if no ping was received within X minutes | If enabled, the connection will be terminated if no ping was received from the server within the last X minutes. |
| **Output** | |
| Output buffer limit | Sets the maximum number of concurrent output buffers. A limit should be set to restrict memory usage, especially for clients that are running day and night. |
| **Job history** | |
| Entry limit | Sets the maximum number of entries in the event log. This can also be useful to limit memory usage. |
| **Heartbeat** | |
| Broadcast address | The address to send heartbeats to. Preferably, this is set to the address of the RenderPal V2 Server. It can also be set to a broadcast IP (ending with a .255). Leave empty to disable heartbeat sending. |
| Broadcast port | The port used when sending heartbeats. |
| Broadcast interval | The interval in minutes in which to send heartbeats. |

### 20.4.1 Renderer configuration

Executables for renderers and versions can be configured in the RenderPal V2 Client as well, using a separate dialog, which is shown below:




For each renderer, you can select the renderer executables for both 32 and 64 bit. If the renderer/version provides information about which executable to choose, it will be shown below the executable fields, as seen in the screenshot. It is also possible to edit multiple entries at once: just select those you want to edit and hit F2 on your keyboard. There are also many useful commands that make configuring multiple executables very comfortable, which can all accessed via keyboard hotkeys and context menu entries (which will also show you which keyboard hotkeys to use):

| Name | Description |
|---|---|
| **Renderer list** | |
| Copy/Paste all executables | With this function, all executables of a renderer, as well as its versions, can be copied and pasted to a different renderer (version executables will be applied to versions with a matching name). |

| Executable list | |
|---|---|
| Renderer/version header | Double-clicking will select all executables of the renderer/version. |
| Copy/Paste | A single entry can be copied to the clipboard (if multiple entries are selected, the first filled out executable will be copied); pasting is possible for multiple entries at once. |
| Select all of same kind | When using this command, all executables of the same kind as the current selection (operating system and architecture) will be selected (multiple selections are supported). |

**Note:** In order to be able to configure renderers from within the client, the client has to be connected to the server when opening the renderer configuration for the first time after the client has been started, so that the renderers can be downloaded from the server. This will only be done once, but you can refresh the renderer list using Refresh renderers.

# 21. Remote client configuration

**Applies to:** Server, Remote Controller

In RenderPal V2, clients can be configured remotely using either the server or remote controller. The remote client configuration can be accessed from many places, including pool and client context menus, the main view and the client management. With the remote client configuration, you can conveniently configure multiple clients at once (batch configuration); you can also retrieve the configuration from a client (and apply it to some other clients, for example). Another useful feature is the ability to import and export configuration settings.

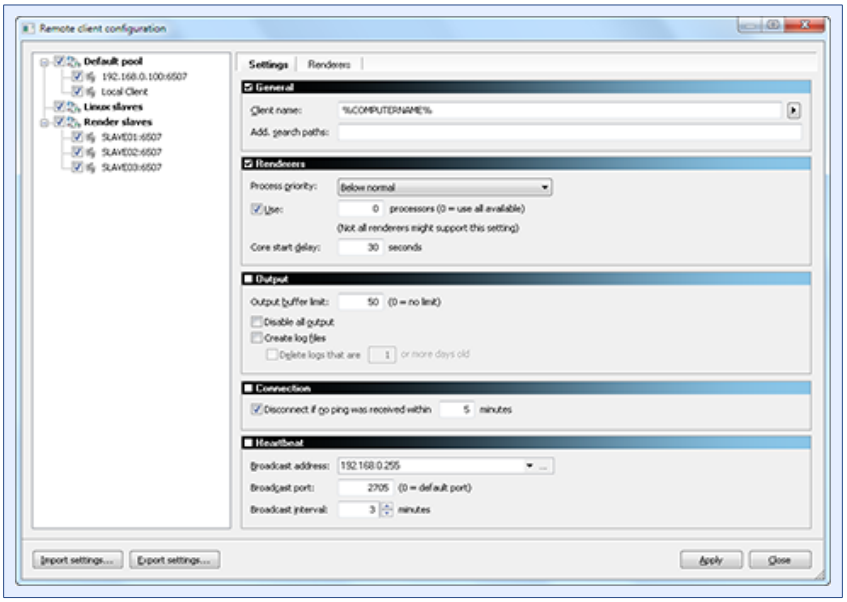
**Usage Tip:** Exporting settings

Exporting settings can be useful if you have several groups of clients that use the same configurations. This way, you can, for example, create configurations for different operating systems and import them again later.

The remote client configuration lets you configure several client settings, as well as renderer settings. They can be applied to all client versions. Even though client and renderer settings are on separate tabs, they will **both** be applied to the selected clients. A group of settings will only be applied if its corresponding check is set.

## 21.1 Client settings

The following screenshot shows the client settings:

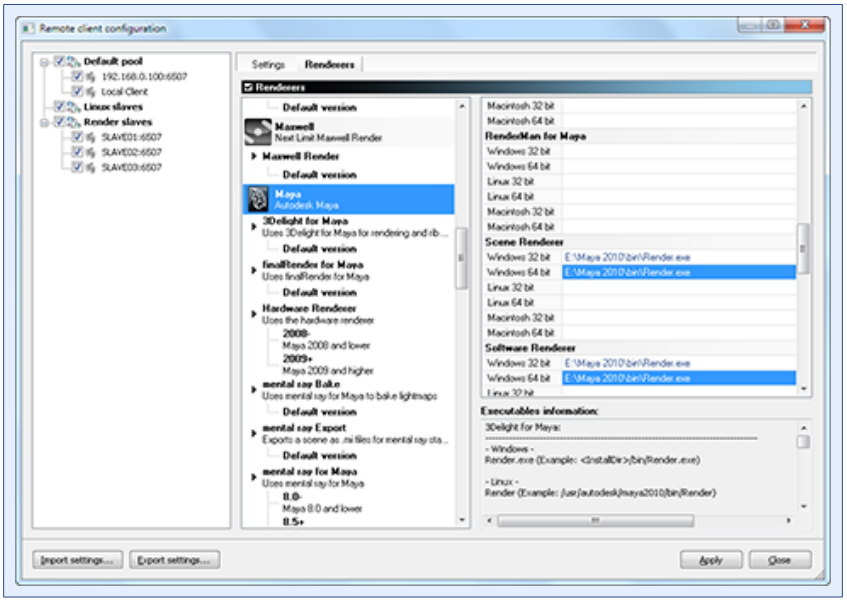



| Name | Description |
|---|---|
| **General** | |
| Client name | The name/alias of the client. The use of `%COMPUTERNAME%` is required if you configure more than one client at a time. |
| Additional search paths | Enter any additional paths here that the renderers should use when searching files etc. |
| **Renderers** | |
| Process priority | The process priority the renderers should be started with. It is recommended to use lower priorities, so that the whole system (and especially the RenderPal V2 Client) stays responsive. |
| Use X processors | If enabled, the specified number of processors will be used for rendering. Note that this setting might not work with all renderers. |

| Name | Description |
|---|---|
| Core start delay | Specifies the delay between the start of parallel renderings. This is used to prevent excessive system and network traffic. |
| **Output** | |
| Output buffer limit | Sets the maximum number of concurrent output buffers. A limit should be set to restrict memory usage, especially for clients that are running day and night. |
| Disable all output | This will completely turn off all textual output; not recommended. |
| Create log files | If enabled, all output will be written to log files. |
| Delete logs that are X or more days old | If enabled, logs that are X or more days old will be automatically deleted (to save disk space). |
| **Connection** | |
| Disconnect if no ping was received within X minutes | If enabled, the connection will be terminated if no ping was received from the server within the last X minutes. |
| **Heartbeat** | |
| Broadcast address | The address to send heartbeats to. Preferably, this is set to the address of the RenderPal V2 Server. It can also be set to a broadcast IP (ending with a .255). Leave empty to disable heartbeat sending. |
| Broadcast port | The port used when sending heartbeats. |
| Broadcast interval | The interval in minutes in which to send heartbeats. |

## 21.2 Renderer settings

The renderer settings allow you to configure the various standard renderers. The screenshot below shows the renderer settings:




For each renderer, you can select the renderer executables for all operating systems (Windows, Linux and Macintosh) and for both 32 and 64 bit. This allows you to configure clients of all operating systems in ”one go”. If the renderer/version provides information about which executable to choose, it will be shown below the executable fields, as seen in the screenshot. It is also possible to edit multiple entries at once: just select those you want to edit and hit F2 on your keyboard. There are also many useful commands that make configuring multiple executables very comfortable, which can all accessed via keyboard hotkeys and context menu entries (which will also show you which keyboard hotkeys to use):

| Name | Description |
| --- | --- |
| **Renderer list** | |
| Copy/Paste all executables | With this function, all executables of a renderer, as well as its versions, can be copied and pasted to a different renderer (version executables will be applied to versions with a matching name). |
| **Executable list** | |
| Renderer/version header | Double-clicking will select all executables of the renderer/version. |
| Copy/Paste | A single entry can be copied to the clipboard (if multiple entries are selected, the first filled out executable will be copied); pasting is possible for multiple entries at once. |
| Select all of same kind | When using this command, all executables of the same kind as the current selection (operating system and architecture) will be selected (multiple selections are supported). |

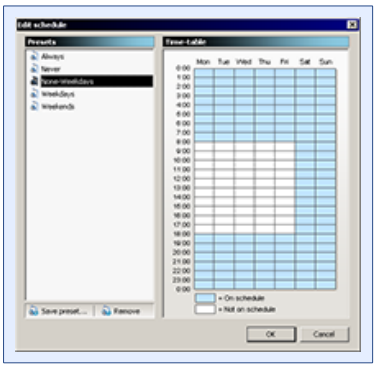
# 22. Miscellaneous

**Applies to:** Global

This chapter is dedicated to a handful features and functions that are too small to get their own chapter, but still deserve some attention.

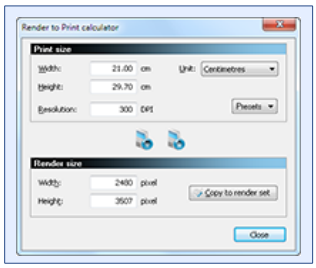
## 22.1 The scheduling dialog

The scheduling dialog always comes to use when editing schedules (used in pools, user accounts and so on). The screenshot on the right shows the scheduling dialog. The schedule shows a complete week, divided into days (columns) and hours (rows). To put a block of hours onto schedule, hold the left mouse button and drag; to remove a block from schedule, hold the right mouse button and drag. You can also save presets for later use; do load a preset, simply double-click it.



## 22.2 The render to print calculator

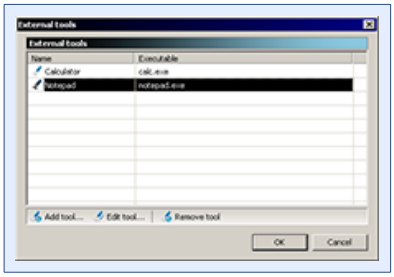
The render to print calculator, which is shown on the right, allows you to quickly convert render (pixel) sizes to print sizes and vice versa. A few common sizes are also provided as presets. The calculated render size can be copied to the current render set as well.

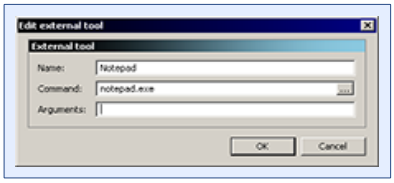


## 22.3 External tools

RenderPal V2 allows you to specify external tools you can quickly access from within the main GUI or by using the keyboard (Ctrl+1 - Ctrl+9). The screenshots on the right show the external tools list as well as the external tool dialog.

An external tool consists of a descriptive name, the executable (Command) and some optional command-line parameters (Arguments).





# 23. The console client

**Applies to:** Client (console)

The console-based client is available for Windows, Linux and Macintosh. As the name implies, it is a non-graphical, non-interactive client; it offers almost the same features as its graphical counterpart, with a few exceptions. The console client can be either configured by hand, editing its configuration file, or by using the remote client configuration feature of the RenderPal V2 Server and Remote Controller (which is recommended).

Using the console client is extremely easy: once configured (which will be covered in the next section), all you have to do is to start it using the ”rpclientcmd” file found in the console client directory. The client will then wait for commands from the RenderPal V2 Server.

**Note:** The console client supports output logging just like the graphical client; the output can also be remotely retrieved. Event logging is, however, not supported.

**See also:** Installation

## 23.1 Configuration

Manual configuring the console client involves editing its configuration file, **RpClientCmd.conf**. If you want to edit this file by hand, you should be familiar with the common configuration/ini file layout. In most cases, you will only want to modify the heartbeat options by hand. The following table lists all available settings:

| Name | Description |
|---|---|
| **Client.General** | |
| Alias | The name/alias of the client; this is used in the submitter column of the net job list, for example. Use `%COMPUTERNAME%` as a placeholder for the computer name. |
| SearchPath | Enter any additional paths here that the renderers should use when searching files etc. |
| **Client.Output** | |
| OutputBufferLimit | Sets the maximum number of concurrent output buffers. A limit should be set to restrict memory usage, especially for clients that are running day and night. |
| TurnOffOutput | This will completely turn off all textual output; not recommended. |
| **Output** | |
| Logging | If set to true, all output will be written to log files. |
| DeleteLogs | If set to true, old log files will be automatically deleted (to save disk space). |
| LogAgeDays | The maximum age (in days) for log files until they will be deleted. |
| **Client.Renderers** | |
| Processors | Specifies the number of processors that will be used for rendering. A setting of 0 usually means ”use all available”; setting this to -1 disables this feature. Note that this setting might not work with all renderers. |
| **Client.Connection** | |
| ListenPort | The client will listen on this port for incoming server connections. |
| KeepAlives<br>KeepAliveTime | If set to true, the connection will be terminated if no ping was received from the server within the last X minutes (defined in **KeepAliveTime**). |
| **Client.Heartbeat** | |
| BroadcastAddress | The address to send heartbeats to. Preferably, this is set to the address of the RenderPal V2 Server. It can also be set to a broadcast IP (ending with a .255). Leave empty to disable heartbeat sending. |
| BroadcastPort | The port used when sending heartbeats. |
| BroadcastInterval | The interval in minutes in which to send heartbeats. |

| Name | Description |
|---|---|
| **Renderers.<Name>** | |
| Windows_32bit<br>Windows_64bit<br>Linux_32bit<br>Linux_64bit<br>Macintosh_32bit<br>Macintosh_64bit | The executable file for the given renderer, operating system and architecture. |

**See also:** Configuration basics

## 23.2 Autostart

### Linux

To install the client as a daemon, you have to edit the file **daemon/rpclient** in the console client directory first. At the beginning of this file, you will find three variables: `RP_USER`, `RPCLIENT_DIR` and `DAEMON_ARGS`. Set the variable `RP_USER` to the user you want to run the client under; it is recommended to use **root** here (otherwise, system control commands will not work). Set `RPCLIENT_DIR` to the console client directory. You can also specify any additional arguments that should be passed to the daemon in `DAEMON_ARGS`. Once done, open a terminal, change to the console client directory and type **sudo ./install-daemon.sh install**. To remove the daemon, use **remove** as the argument. You can also **start** and **stop** the client daemon.

### Macintosh

To install the client as a daemon, open a terminal, change to the console client directory and type **sudo ./install-daemon.sh install**. To remove the daemon, use **remove** as the argument. You can also **start** and **stop** the client daemon.

Both the Linux and Macintosh console client offer a handy tray icon utility that can be used to control the client and its service. When installing the daemon (see above), you will be asked whether to install the tray icon. Please note that in order to use the tray icon you will need at least Java 1.6 installed.

## 23.3 Command-line options

The console client offers a few command-line options. The following table lists all available options:

| Name/Syntax | Description/Parameters |
|---|---|
| **userdir** | Overrides the user directory (the directory where all log files and temporary files will be located) |
| `-userdir:<userdir>` | `<userdir>:` Specifies the directory to use. |
| **verbose** | Sets the verbosity level |
| `-verbose:<level>` | `<level>:` Level from 1 to 4; the higher the level, the more information will be shown. |
| **quit** | Terminates a running client |
| `-quit` | No parameters. |
| **heartbeat** | The address to send heartbeats to. Preferably, this is set to the address of the RenderPal V2 Server. It can also be set to a broadcast IP (ending with a .255). |
| `-heartbeat:<address>` | `<address>:` The server address to use. |
| **help** | Displays command-line options |
| `-help` | No parameters. |

Switches and their parameters can either be delimited by a space ( ), a colon (`:`) or an equal sign (`=`); if a parameter includes spaces, it has to be enclosed in quotes.

# 24. The console remote controller

**Applies to:** Remote Controller (console)

The console-based remote controller is available for Windows, Linux and Macintosh. The console remote controller can be used to submit new net jobs to the RenderPal V2 Server. This makes it the perfect tool for rendering pipeline integration. To use the console remote controller, its configuration file has to be edited first.

Once the console remote controller has been configured, it can be used by calling the ”rprccmd” file and passing the various command-line switches. The command-line allows you to specify the various render set settings, as well as numerous net job related settings. Default settings for all command-line switches can also be specified in a special configuration file; this way, you don’t need to specify the same switches all the time.

**Note:** The console remote controller will always download the entire renderer pool when logging in, so there is no need to copy any files manually.

**See also:** Installation

## 24.1 Configuration

To configure the console remote controller, you have to edit the **RpRcCmd.conf**. If you want to edit this file, you should be familiar with the common configuration/ini file layout. In order to be able to use the console remote controller, it is necessary to edit its configuration first - but do not worry, there is not much to configure. The following table lists all available settings:

| Name | Description |
|---|---|
| **RemoteController.General** | |
| Alias | The name/alias of the remote controller; this is used in the submitter column of the net job list, for example. Use `%COMPUTERNAME%` as a placeholder for the computer name. |
| **RemoteController.Server** | |
| ServerAddress<br>ServerPort | The address of the RenderPal V2 Server, as long as its port, to use. |
| UserName | The login name to use. |
| UserPwd | The login password to use. |
| **RemoteController.NetJobs** | |
| NetJobNameFormat | Sets the default net job name if non is specified; use the `$(scene)` placeholder for the scene filename |

**See also:** Configuration basics

## 24.2 Command-line options

The console remote controller offers numerous command-line options. Most of them are optional, though some have to be specified (like the renderer to use for a net job). All render settings can be set via command-line as well. To do so, simply pass the corresponding parameter ID as a switch, as long as the desired value (like `-camera ”cam”`). These switches won’t be listed here, but you can always use `-listparams` to get a complete list. The following tables list all available options:

### General switches

| Name/Syntax | Description/Parameters |
|---|---|
| **login** | Specifies the login to use, overriding the login information given in the configuration file |

| Name/Syntax | Description/Parameters |
|---|---|
| -login "<user> [:<pass>]" | <user>: Specifies the user name<br><pass>: Specifies the password for the account |
| **server** | Specifies the server to use, overriding the server information given in the configuration file |
| -server "<address> [:<port>]" | <address>: Specifies the server address<br><pass>: Specifies the server port |
| **userdir** | Overrides the user directory (the directory where all log files and temporary files will be located) |
| -userdir <userdir> | <userdir>: Specifies the directory to use. |
| **defsection** | Overrides the name of the section to use from the RpRcDefaults.conf file |
| -defsection <section> | <section>: The name of the section to use |
| **listrenderers** | Shows a list of all available renderers |
| -listrenderers [<group>] | <group>: An optional group name; only renderers belonging to that group will be listed |
| **listpools** | Shows a list of all available client pools. |
| -listpools | No parameters. |
| **listparams** | Shows a list of settings that can be supplied via command-line for the specified renderer. |
| -listparams <renderer> | <renderer>: The renderer from which the parameters should be shown. |
| **query** | Queries data from the server. |
| -query <querystring> | <querystring>: The query string that tells the server which data you are interested in (see below for details about querying the server). |
| **control_nj** | Controls one or more net jobs. |
| -control_nj <commandstring> | <commandstring>: The command string that tells the server which net jobs to control in what way (see below for details about controlling net jobs). |
| **control_pool** | Controls one or more client pools. |
| -control_pool <commandstring> | <commandstring>: The command string that tells the server which pools to control in what way (see below for details about controlling pools). |
| **execute, execute_idleonly** | Executes a command on clients. |
| -execute <clients><br>-execute_idleonly | <clients>: A list of target clients, separated by semicolons; to target an entire pool, prefix the pool name with a pool: (e.g. "pool:My pool"). Immediately after this switch the actual command-line to execute follows (the first token being the executable, e.g. -execute "Client1;Client2" myprog.exe -param1 value1 -param2 "value number 2").<br>If -execute_idleonly is specified, the -execute command will only be executed on idle clients. |
| **compact** | When using this switch in conjunction with -listpools or -listrenderers, only the "bare" list without any extra information/decoration will be shown. |
| -compact | No parameters. |
| **retnjid** | When specified, the ID of the newly created net job will be returned by the executable. |
| -retnj | No parameters. |
| **log** | Enables logging of all output to a log file (RpRcCmd.log) |
| -log | No parameters. |
| **help** | Displays command-line options |
| -help | No parameters. |

### Render set switches (excluding render settings)

| Name/Syntax | Description/Parameters |
|---|---|
| **importset** | Imports the specified render set |
| `-importset <set>` | `<set>:` The render set file to import. If no qualified path is specified, the set will be searched in the Presets/NetJobs and root directory. The file ending can also be left out. |

### Net Job switches

| Name/Syntax | Description/Parameters |
|---|---|
| **nj_preset** | Imports the specified net job preset; can be used multiple times. |
| `-nj_preset <preset>` | `<preset>:` The preset to import. If no qualified path is specified, the preset will be searched in the Presets/NetJobs and root directory. The file ending can also be left out. |
| **nj_name** | Sets the net job name. |
| `-nj_name <name>` | `<name>:` The net job name to use. |
| **nj_priority** | Sets the net job priority. |
| `-nj_priority <priority>` | `<priority>:` The net job priority to use (1-10). |
| **nj_urgent** | If specified, the net job will be marked as urgent. |
| `-nj_urgent` | No parameters. |
| **nj_renderer** | Sets the renderer to use. |
| `-nj_renderer <renderer>` | `<renderer>:` The renderer for the net job. For a complete list of available renderers, use `-listrenderers`. The syntax for the renderer name is ”Renderer/Version”; if no version is supplied, the default version will be used. |
| **-nj_render32bit**<br>**-nj_render64bit** | Enables rendering using 32/64 bit renderers. If none of these is provided, both architectures will be used. |
| `-nj_render32bit`<br>`-nj_render64bit` | No parameters. |
| **nj_rendercores** | Specifies the number of parallel renderings (on a client). |
| `-nj_rendercores <count>` | `<count>:` The number of render cores (0 = one core per processor) |
| **nj_noemails** | If specified, no email notifications will be sent about this net job. |
| `-nj_noemails` | No parameters. |
| **nj_emailusers** | Adds additional users who should also receive email notifications about this net job |
| `-nj_emailusers <users>` | `<users>:` List (separated by semicolons) of additional user names |
| **nj_emailrecpt** | Sets additional email recipients for this net job. |
| `-nj_emailrecpt <addresses>` | `<addresses>:` A list of additional email addresses, separated by semicolons. |
| **nj_splitmode** | Sets the frame splitting mode to use. |
| `-nj_splitmode ”<mode>, <count>”` | `<mode>:` The splitting mode (1 = total pieces, 2 = frames per job).<br>`<count>:` The number of total pieces/frames per job. |
| **nj_slicemode** | Sets the image slicing mode to use. |
| `-nj_slicemode ”<mode>, <x>,<y>,[overlap], [format]”` | `<mode>:` The slicing mode (1 = rows/columns, 2 = piece sizes)<br>`<x>:` The number of rows/piece width<br>`<y>:` The number of columns/piece height<br>`<overlap>:` Overlap value in pixels.<br>`<format>:` The image name format. |
| **nj_extsplitting** | Specifies parameters used for additional splitting |
| `-nj_splitting <params>` | `<params>:` The parameters to use for splitting, separated by semicolons. |
| **nj_notes** | Sets the notes for this net jobs. |
| `-nj_notes <notes>` | `<notes>:` The textual notes. |
| **nj_tags** | Specifies tags for the net job. |

| Name/Syntax | Description/Parameters |
|---|---|
| -nj_tags <tags> | <tags>: A list of tags, separated by colons or semicolons. |
| **nj_pools** | Assigns the specified client pools to the net job; if no pools are specified, all available will be used. |
| -nj_pools <poollist> | <poollist>: Names of the pools to use. Separated by semicolons or commas. |
| **nj_paused** | If specified, the net job will be started paused. |
| -nj_paused | No parameters. |
| **nj_clientlimit** | Sets the maximum number of clients to use. |
| -nj_clientlimit <limit> | <limit>: The client limit. |
| **nj_minclientpriority** | Sets the minimum client priority to use. |
| -nj_minclientpriority <minpriority> | <minpriority>: The minimum client priority (1-10). |
| **nj_mindispatchdelay** | Sets the minimum delay between chunk dispatches. |
| -nj_mindispatchdelay <mindelay> | <mindelay>: The minimum dispatch delay (in seconds). |
| **nj_lowerth, nj_lowerth_sec, nj_lowerth_action** | Sets the lower rendering time threshold and action for render time monitoring. |
| -nj_lowerth <th><br>-nj_lowerth_sec <ths><br>-nj_lowerth_action <a> | <th>: The lower threshold in minutes.<br><ths>: The lower threshold in seconds<br><a>: The action to take (0 = Restart chunk, 1 = Cancel) |
| **nj_upperth, nj_upperth_sec, nj_upperth_action** | Sets the upper rendering time threshold and action for render time monitoring. |
| -nj_upperth <th><br>-nj_upperth_sec <ths><br>-nj_upperth_action <a> | <th>: The upper threshold in minutes.<br><ths>: The upper threshold in seconds.<br><a>: The action to take (0 = Restart chunk, 1 = Cancel, 2 = Skip) |
| **nj_dependency** | Sets the ID of the net job the new net job should depend on; can be used multiple times. |
| -nj_dependency <id> | <id>: ID of the net job the new net job should depend on. |
| **nj_deptype** | Sets the dependency type |
| -nj_deptype <type> | <type>: The dependency type (0 = Job, 1 = Chunk IDs, 2 = Frames, 3 = Image slices) |
| **nj_depunfinishedasdone** | If specified, unfinished net jobs will be treated as done when checking for net job dependencies. |
| -nj_depunfinishedasdone | No parameters. |
| **nj_framechecking** | Sets the automatic frame checking parameters. |
| -nj_framechecking <mode>,[imgname], [imgdir],[threshold], [subdirs],[exact], [padding],[disable subsequent checking] | <mode>: The frame checking mode (1 = Job, 2 = Chunk)<br><imgname/dir>: The image name/directory<br><threshold>: The filesize threshold (in kb)<br><subdirs>: Search inside of subdirectories (yes/no)<br><exact>: Perform exact matches (yes/no)<br><padding>: Frame number padding<br><disable...>: Pass yes to disable frame checking for subsequent jobs |
| **nj_dispatchorder** | Sets the chunk dispatch order for the net job. |
| -nj_dispatchorder <order> | <order>: The dispatch order (0 = Forward, 1 = Backward, 2 = Evenly distributed, 3 = Random) |
| **nj_firstlastfirst** | If specified, renders the first and last chunk first. |
| -nj_firstlastfirst | No parameters. |
| **nj_clients** | If specified, only the given clients will be used for rendering. |
| -nj_clients <clients> | <clients>: A semicolon-separated list of clients; either use the GUID of the client (useful when using in combination with data queries) or the client name. |
| **nj_blockedclients** | The specified clients will be blocked from this net job. |

| Name/Syntax | Description/Parameters |
|---|---|
| `-nj_blockedclients <clients>` | `<clients>`: A semicolon-separated list of clients; either use the GUID of the client (useful when using in combination with data queries) or the client name. |
| **nj_color** | Sets the net job color. |
| `-nj_color "<r>,<g>,<b>"` | `<r>`: The red component (0-255)<br>`<g>`: The green component (0-255)<br>`<b>`: The blue component (0-255) |

Switches and their parameters can either be delimited by a space ( ) or an equal sign (=); if a parameter includes any spaces, it has to be enclosed in quotes.

**Usage tip:** Parsing the available renderers and pools

When integrating RenderPal V2 into a rendering pipeline, it is often useful to retrieve the list of available renderers and client pools for further processing. To do this, use the command-line remote controller with the -listrenderers/-listpools switch in conjunction with the -compact switch. The resulting output can then be easily parsed.

**See also:** Renderers

### 24.2.1 Advanced scene filename syntax

If a scene filename is preceded with an @ (e.g. `@C:\Images\*.jpg`), the console remote controller will parse the specified filename for special tokens. If you do not precede the filename with an @, it will not be parsed and submitted as-is. Two types are supported: **wildcards** and **auto frame-numbering**.

#### Wildcards

Any wildcards (* and ?) will be parsed, and all files matching the specified pattern will be added to the net job. Note that you have to specify a full path in this case, and that the path has to be accessible by the console RC.

#### Auto frame-numbering

The console remote controller can easily add scene files that belong to a sequence (e.g. File001.ifd, File002.ifd...) without the need to specify each file. The syntax to achieve this is: `#[frame-list]`. The frame-list is a list of frame numbers that shall be added; this can be a single frame or a range, and multiple frames can be separated by a comma: `#[1,10-50,100-110]`. To set the frame number padding, simply specify multiple #: `###[...]` will result in a padding of 3; an optional frame step can be specified after the last frame range, separated by a semicolon: `#[1-10;2]` will add the frames 1, 3, 5 and so on.

**Example:** Auto frame-numbering

```
@C:\Scenes\MyScene.###[1,6-10;2].scn

This will result in:
C:\Scenes\MyScene.001.scn
C:\Scenes\MyScene.006.scn
C:\Scenes\MyScene.008.scn
C:\Scenes\MyScene.010.scn
```

### 24.2.2 Default values

As mentioned earlier, you can specify default values for all available command-line switches (including render settings) using a special configuration file (**RpRcDefaults.conf**). This file can contain multiple sections with different default values. By default, the default section **[Defaults]** will be used; to use a different section for default values, use the `-def-section` command-line parameter. To set a default value, simply add an entry for the switch (without the preceding minus) and set its value. Here is an example configuration file:

**Example:** RpRcDefaults.conf

```
[Defaults]
nj_renderer = maya_sw
s = 1
e = 100
rp_cmd = -tempdir C:\Temp

; To use this section, pass ”-defsection MyOtherDefaults” to the console RC
[MyOtherDefaults]
nj_renderer = maya_rman
s = 0
e = 10
```

### 24.2.3 Net job presets

Net job presets are an easy way to use net job features which are not directly supported via command-line, like net job events. Simply create the desired net job preset using RenderPal V2 (or create it with your own tool or script) and import it via `-nj_preset`.

You can import multiple presets by using multiple `-nj_preset` switches; the presets will be imported in the order in which they were passed to the command-line. This is useful to import various net job events, for example. Since for every preset you can define which settings of a net job should be applied, combining multiple presets can save you a great amount of time and typing.

**Usage tip:** Copying preset files

If you want to be able to use the -nj_preset switch without the need to specify fully qualified paths, you need to copy the corresponding preset files to the Presets/NetJobs or root directory of the console Remote Controller. This way, you only need to specify the name of the preset to import. Example:

```
-nj_preset ”mypreset”
```

This will search for a file called ”mypreset.rnjprs” in the Presets/NetJobs directory first, then in the root directory.

### 24.2.4 Controlling net jobs, net job chunks and client pools

RenderPal V2 allows you to easily control one or more net jobs, net job chunks and client pools via command-line using the `-control_nj, -control_chunk` and `-control_pool` switches. Both switches use the same command string format:

#### Command string syntax

A command string consists of one or more command tokens, which tell RenderPal V2 what to do. The general syntax of such a token is: `control-verb:identifier`. The control-verb can be one of the following:

| Net Job control verbs | Description |
|---|---|
| `pause` | Pauses the specified net job. |
| `unpause` | Unpauses the specified net job. |
| `remove` | Removes (deletes) the specified net job. |
| `cancel` | Cancels the specified net job. |
| `cancelpending` | Cancels all pending chunks of the specified net job. |
| `restart` | Restarts the specified net job. |
| `restartcancelled` | Restarts all cancelled chunks of the specified net job. |
| `restarterroneous` | Restarts all erroneous chunks of the specified net job. |
| **Net Job Chunk control verbs** | **Description** |
| `skip` | Skips the specified chunk |

| Net Job control verbs | Description |
|---|---|
| restart | Restarts the specified chunk. |
| cancel | Cancels the specified chunk. |
| **Pool control verbs** | **Description** |
| start | Starts the specified pool. |
| stop | Stops the specified pool. |
| removefromall | Removes the pool from all net jobs. |
| assigntoall | Assigns the pool to all net jobs. |
| connectall | Connects to all clients of the specified pool. |
| disconnectall | Disconnects from all clients of the specified pool. |
| shutdown | Shuts all clients of the specified pool down. |
| reboot | Reboots all clients of the specified pool. |

The identifier is the identifier for the object to control. For a net job this is the net job number; for pools this is their name. For net job chunks, the identifier syntax is `chunk-id1,chunk-id2,...@netjob-id` (e.g. `-control_chunk "skip:1,2,3@4711"`); you can also specify a range of chunks: `-control_chunk "cancel:1-10@4711"`.

To control more than one object at a time, you can either use multiple `-control_*` switches, or you can combine tokens and identifiers. To send the same control verb to two token, simply combine them using a comma: `cancel:100,101,200`. To send different control verbs, combine them using semicolons: `pause:100;cancel:100`.

### 24.2.5 Data queries

Using the switch `-query`, it is possible to retrieve live data from the RenderPal V2 Server. Using a simple query string, you can fetch any data from the server, including net jobs, chunks, pools and clients. All data is returned in a simple XML format:

**Example:** Data query

```
<?xml version="1.0" encoding="UTF-8"?>
<ServerData>
  <NetJobChunk id="1881">
    <Frames>1.000 - 10.000</Frames>
    <EndTime>0</EndTime>
    <ChunkID>1</ChunkID>
    <StartDateTime>40547.905428</StartDateTime>
    <Slice>Entire image</Slice>
    <Attempts>3</Attempts>
    <StartDateText></StartDateText>
    <RenderingClient></RenderingClient>
    <Priority>0</Priority>
    <StartTime>0</StartTime>
    <NetJobID>1881</NetJobID>
    <ExtSplitting>None</ExtSplitting>
    <RenderingClientUuid>00000000-0000-0000-0000-000000000000</RenderingClientUuid>
    <History>
      <Value>1|1294173827|Rendering started [Local Client]|</Value>
    </History>
    <RenderingTime>0</RenderingTime>
    <Scene>blahahahhaa</Scene>
    <StatusText>Erroneous</StatusText>
    <Status>6</Status>
  </NetJobChunk>
</ServerData>
```

The returned data is contained within a `<ServerData>` element. Most values should be self-explaining; in some cases,

you will find two different fields for the same value (like the `<Status>` and `<StatusText>` values): These usually represent the data as it is used by RenderPal, as well as a more readable version.

The server is queried using the `-query` switch; this switch only has one parameter, its query string (see below). You can also pass an additional file name at the end of the command-line (like `-query "..." X:\MyQuery.xml`); all data will then be written to that file (if no file is specified, the data will simply be printed to the console). `-query` can be used in conjunction with `-compact`, of course.

### Query string syntax

A query string consists of one or more query tokens, which tell RenderPal V2 which data it should return. The general syntax of such a token is: `token-type:identifier`. The token-type can be one of the following:

| Token type | Description |
|---|---|
| `netjob` | Fetches the specified net job (does not include its chunks). |
| `chunks` | Fetches the chunks of the specified net job. |
| `pool` | Fetches the specified pool (does not include details about its clients). |
| `pool-clients` | Fetches the specified and includes all its assigned clients. |
| `client` | Fetches the specified client. |
| `statistics` | Fetches some general server information (like total number of net jobs, number of finished net jobs and so on). |

The identifier is the identifier for the requested object. For a net job or its chunks, this is the net job number; for pools and clients, this is their name; to query all clients of a specific pool, you can use the following syntax: `client:(poolname)`. It is also possible to query all objects of the same type by using an asterisk (*): `netjob:*` will return all net jobs. The asterisk should be used with care, as it can result in large amounts of data. The `statistics` token does not need any identifier.

To query more than one object at a time, you can either use multiple `-query` switches, or you can combine tokens and identifiers. To query two tokens of the same type, simply combine them using a comma: `netjob:100,101,200`. To query tokens of different types, combine them using semicolons: `netjob:100;chunks:100`. It is always advisable to query all the data you need in one go (instead of launching the console remote controller multiple times), as this reduces the network traffic.

**Note:** When querying the server, only the data the user account has access to will be returned (in other words, data querying also respects user account/group settings).

# Part III - Advanced Topics

The final part of this manual will deal with more advanced topics, including dynamic scene and output names, user-created updates and script-based renderers. The topics discussed here will usually not be necessary for the common daily RenderPal V2 tasks and act more as a reference for advanced users.

# 25. Dynamic scene and output file names

The scene files as well as the output file and directory in a render set can contain so-called dynamic variables; these are parameters from the selected renderer that will then be replaced with the corresponding value set in the render set (or an automatically calculated one) for each chunk of the resulting net job.

The general syntax for variables is: `$(<parameter>[:format])`, where `<parameter>` is the ID of a renderer parameter (like `sf` for the start frame) and `format` is an optional format specifier.

A simple variable could look like this: `$(sf)` or `$(camera)`; these would then be replaced with the value taken from the render set. To get more control about how the value is formatted, you can also specify a format specifier, which follows more or less the common printf-syntax: `%[padding]<type>`, where `padding` is an optional padding (e.g. `%04` will pad with up to four zeros; note the leading zero) and the type can be either `i` or `d` for integers or `f` or `e` for floating-point numbers. If you do not specify a format, the value will be formatted ”as-is”. Here are a few examples:

**Examples:** Dynamic scene and output file name variables

```
The given values are:
sf = 1; camera = SphereCam

W:\Scenes\Results\MyImg.$(sf:%04d).jpg =>
W:\Scenes\Results\MyImg.0001.jpg

W:\Output\MyCameras\$(camera)\ =>
W:\Output\MyCameras\SphereCam\
```

If you use a dynamic variable, you should always ensure that there is an actual value in the render set for it (or an automatically calculated one, like `sf` and `ef`); otherwise, things will not work. It is also possible to use more than one variable (e.g. `W:\Resources\$(camera)\MyRes.$(sf:%03d).jpg`).

Usually, you will not use simple ”static” variables that will be the same for every chunk. Instead, you will use variables that are different for each chunk, usually due to the applied splitting (like frame splitting, camera splitting or even image slicing); such variables are the start- and end-frame, slice dimensions or other split values (cameras, layers and that alike). If you use one of these variables, the chunks will have different scene/output file names, depending on the values of the chunk. It is also important to understand that using dynamic variables will never result in more chunks, but they can result in multiple scenes in one chunk, even if there is only one in your render set - this is where the true power of this feature comes in!

## 25.1 List parameters

If you use a simple, non-list parameter (like image width and height), the dynamic variables will be replaced with the current value (of the corresponding net job chunk) and that’s about it. However, if you use a list parameter (like the frame list, a camera or layer list) and the corresponding net job chunk has more than one value in its list (due to the applied splitting - or the lack of splitting for that value), some magic will happen. While the net job will not have more chunks than expected, the chunks will automagically have more than one scene - even if there’s only one in your render set.

Each net job chunk has its own portion of values of a list parameter you’ve filled in the render set; depending on the applied splitting, this can be a single value, some values or the entire list. For each list value, a new scene file entry will be created automatically. Here is a quick example:

**Example:** List parameter

```
The given values are:
scene  = W:\MyScenes\Scene_$(camera).scn
camera = SphereCam, ConeCam, BoxCam

No splitting is applied to ”camera”, so the resulting chunks will have all three values
in their list.
```

```
Resulting scene files in each chunk:
W:\MyScenes\Scene_SphereCam.scn
W:\MyScenes\Scene_ConeCam.scn
W:\MyScenes\Scene_BoxCam.scn
```

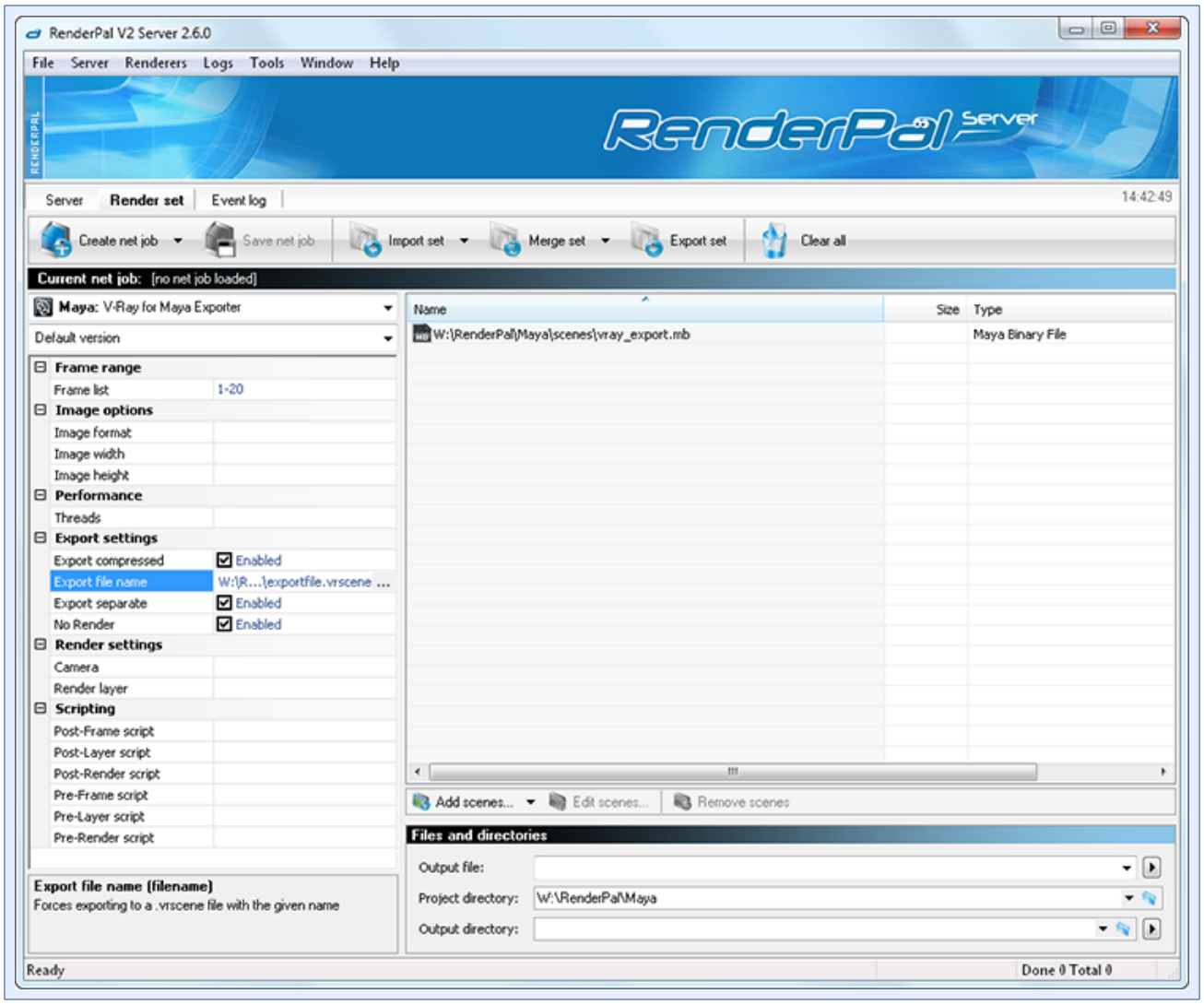
If the selected renderer does not use a combined scene list, the client will render each scene individually, so one chunk can actually result in more than one rendering (in the above example, it would render three times, each time with a different scene).

Another example is to use the frame list parameter (`$(frames)`); this is especially useful for sequences. Instead of adding all files by hand to the render set (in some cases, the scenes might not even exist yet - for example, if another job has to create them first), you'll only add one scene file, like `W:\Scenes\MySequence.$(frames:%04d).png` and fill out the frame range of the sequence in the render set (and probably the frame step etc. as well). Now you just create the job, apply a frame splitting as you seem fit, and RenderPal V2 will automagically generate all scene file names based on the frame range of each chunk (so, depending on your splitting, each chunk might have more than one scene). If you'd use the start frame parameter (`$(sf)`) instead, the chunk would only have one scene carrying the start frame value of the chunk (which is just one value, not a list).

**Note:** List parameters usually only make sense for scene files; they can be used in the output file name/directory, but only the first value in the list will be used.

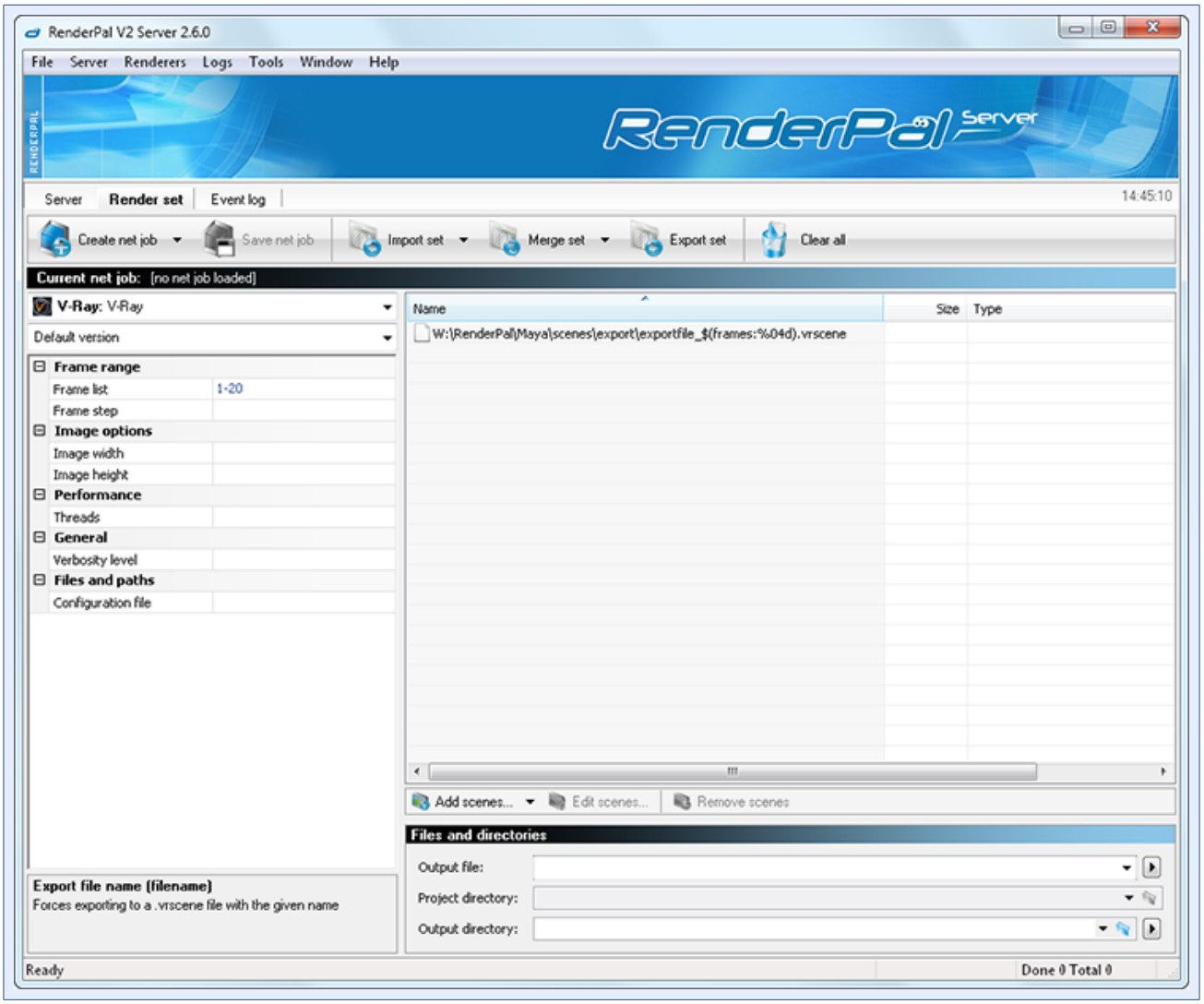
## 25.2 A common example

This example will show you the render sets of two net jobs, where the first one will export a sequence of V-Ray scenes from a Maya scene, while the second one will render those exported .vrscene files with the V-Ray Standalone renderer.

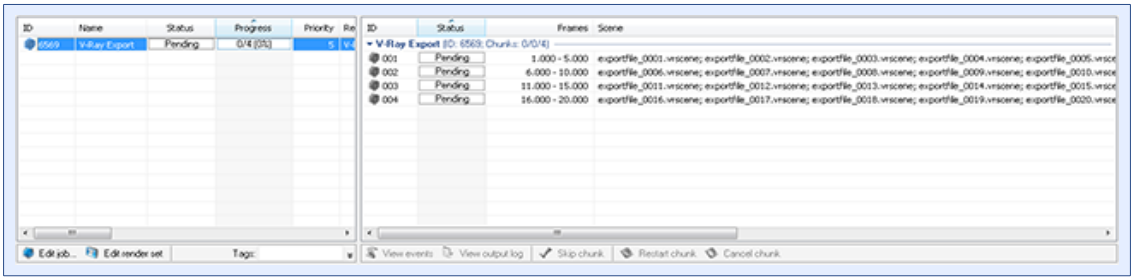


This render set holds nothing special. The only thing to take a closer look at is the ”Export file name” entry (note that this field is specific to this particular renderer): `W:\RenderPal\Maya\scenes\export\exportfile.vrscene`. This is the path and filename for all exported V-Ray scenes; the exported scenes will look like `W:\RenderPal\Maya\scenes\export\exportfile_0001.vrscene, exportfile_0002.vrscene` and so on (from frames 1-20).

Our goal is now to render these files with another net job using the V-Ray Standalone renderer. We could wait for the job to finish first, so that all exported scenes are available and we can manually add all of them to a new job - but that would waste quite a lot of time! Instead, we will create a new job right away using a dynamic variable for the scene name. Take a look at the following render set:

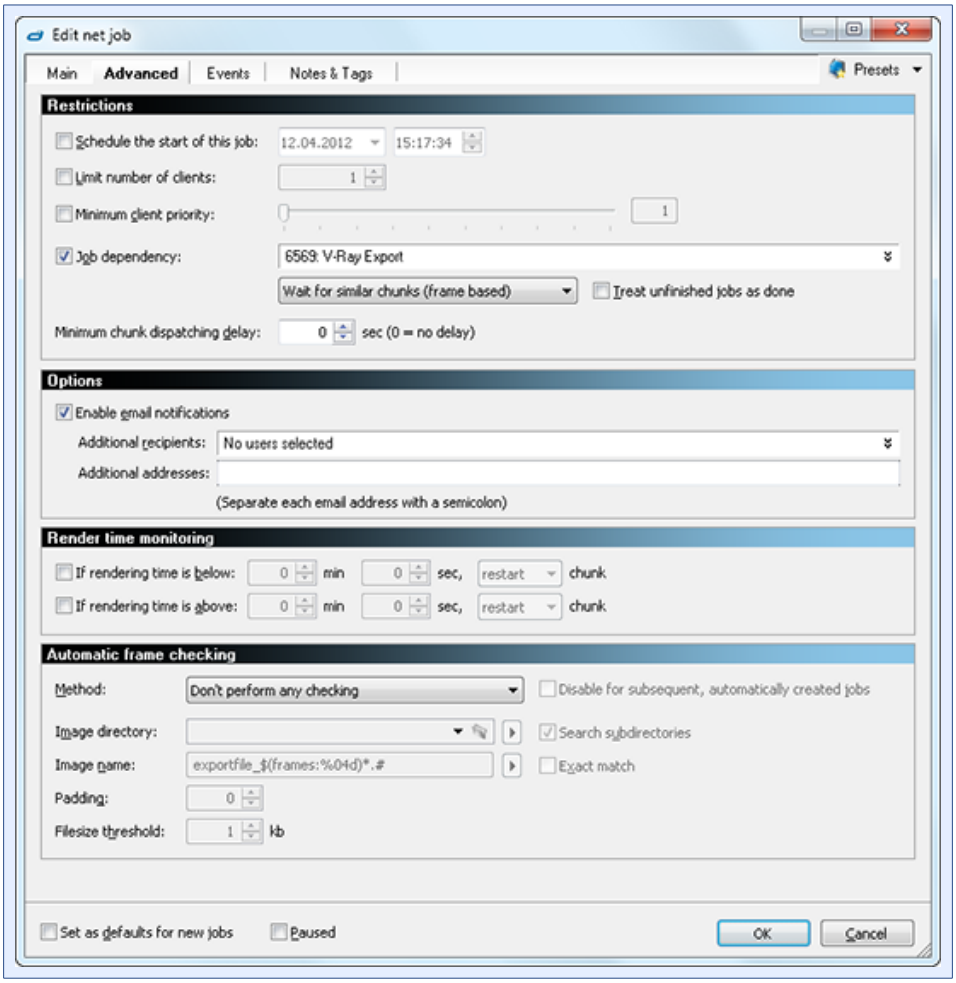



The frame list contains, just like our previous render set, the frames 1-20; note the scene file entry: `W:\RenderPal\Maya\scenes\export\exportfile_$(frames:%04d).vrscene`. In place of the frame number (like `0001` and so on), we have used a dynamic variable for the frame list (and applied a padding of 4 zeros: `%04d`). If we now create a net job based on this set, the `$(frames)` variable will be parsed, and for each frame there is (1-20 in this case), a new scene will be automatically added to the job (depending on the applied frame splitting, the resulting 20 scenes will be distributed across various chunks). To make things a bit clearer, take a look at this screenshot:



We applied a frame splitting of 4 chunks in total to this job; as you can see, each chunk now ”carries” 5 frames, and for each frame, a new scene has been automatically added - just as we wanted! Each chunk will then result in 5 renderings, every time rendering a different scene.

For such scenarios, it is necessary that the second net job will wait on the first one (otherwise, the scenes wouldn’t have been exported and thus wouldn’t even exist yet). Since it might be a waste of time to wait for the entire job, it is advisable to use the frame-based chunk dependency instead, as shown in the following screenshot:




This way, whenever a few scenes have been exported, they can be rendered right away without waiting for the entire job to be done.

This is just one example where dynamic scene and output file names can be useful; this advanced feature opens many new ways of creating complex rendering pipelines. Another useful application for this feature are converters.

# 26. User-created updates

While most updates will directly come from us, it is also possible to create your own updates. An update is a zip file containing at least two files: the update description (**updDesc.ini**), as well as the actual update script (**updScript.py**). To write your own updates, you should have a basic knowledge of Python (or at least programming in general). Besides these two files, the zip can contain anything you want to deploy. Let it be some Python scripts for net job events or your favorite music - it's all up to you.

Before we will discuss the update description file and the updater Python API in detail, we will show you a simple example first, which will just copy a text file to every client.

**Update example:** Zip contents

```
ExampleUpdate.zip
 |-- updDesc.ini
 |-- updScript.py
 |-- MyFile.txt
```

The task of this update will now be to copy **MyFile.txt** to all clients. But first, we will need a description of this update, which will provide RenderPal V2 with the necessary details about the update. A full reference will follow later.

**Update example:** updDesc.ini

```
[Update]
APIVersion = 1.0

[Update.Description]
Name = Update example
Description = A simple update example
MOTD =

[Update.Settings]
Scope = All
OS = All
Critical = No
TerminateApp = No

[Update.ObsoletePackages]
```

The above description will tell RenderPal V2 the name and description of the update, as well as to which components it should be applied (`Scope` and `OS`). We could have set the scope to only clients, but we want to control the file copying solely with the updater script.

**Update example:** updScript.py

```
import RpUpd

if RpUpd.IsType("Client"):
      RpUpd.CopyFiles("$UpdDir/MyFile.txt", "$InstDir", True)
```

That's it! This small script file first checks if the update is applied to a client, and then copies the text file into the client directory. As you can see, creating your own updates is a no-brainer. An empty update (**UpdateBase.zip**) that can be used as the base for your own update can be found in the RenderPal V2 Server directory, located in the **Updates** subdirectory.

**See also:** Update management

## 26.1 Update description file reference

The update description file, **updDesc.ini**, will tell RenderPal V2 various details about the update, including its name, to which components it should be applied, whether it is a critical update and if the update renders any other updates obsolete (so that they will be removed automatically).

The layout of a description file is as follows:

**Layout:** updDesc.ini

```
[Update]
APIVersion = 1.0

[Update.Description]
Name =
Description =
MOTD =

[Update.Settings]
Scope = All
OS = All
Critical = No
TerminateApp = No

[Update.ObsoletePackages]
Package1 =
```

The following table shows all settings available in an update description:

| Value | Description |
|---|---|
| **Update** | |
| APIVersion | The version number of the updater API. Has to be set to **1.0**. |
| **Update.Description** | |
| Name | The name of the update. |
| Description | A brief description of the update. |
| MOTD | The MOTD (message of the day) will pop up when the update is added to the update list in the update management. |
| **Update.Settings** | |
| Scope | Determines which components of RenderPal V2 will receive this update. Possible values are: **All**, **Server**, **Client** and **RemoteController**. Separate multiple components with a comma. |
| OS | Determines to which operating systems this update should be applied. Possible values are: **All**, **Windows**, **Linux** and **Mac**. Separate multiple operating systems with a comma. |
| Critical | Determines whether this is a critical update or not. Set to **Yes** or **No**. If set to yes, the application will automatically be terminated (the TerminateApp flag will be ignored). |
| TerminateApp | Determines whether the application needs to be terminated before this update can be applied (if the executable has to replaced, for example). Set to **Yes** or **No**. |
| **Update.ObsoletePackages** | |
| Package1<br>Package2<br>...<br>PackageN | A list of updates that should be declared as obsolete (superseded by this update). Set the value to the file name of the update (including extension, like `SomeUpdate.zip`). |

## 26.2 Update script API

The updater utilizes small Python scripts that perform the various operations. While these scripts are usually very simple, a certain knowledge of Python is still required to write them. The update script API offers a handful functions that can be used for file operations and to control and restrict the updating process.

**Important:** Python file functions

While Python offers many built-in file functions, it is not recommended to use them instead of the update script functions unless necessary. Only the update script functions will handle directory placeholders and errors correctly.

### Directory placeholders

The update script API uses two directory placeholders: `$UpdDir` and `$InstDir`. These are used in the various file and folder operations. `$UpdDir` points to the directory where the update has been extracted to (source), while `$InstDir` points to the RenderPal V2 directory (destination). Also, you should always use forward slashes in any path and file name. Here is an example showing how to copy a file from the update file to a subdirectory of RenderPal V2:

```
RpUpd.CopyFiles("$UpdDir/MyFile.txt", "$InstDir/SomeSubDir", True)
```

This example will copy the file **MyFile.txt** from the update into the subdirectory **SomeSubDir** of the RenderPal V2 directory (which will be automatically created if it doesn't exist).

**Usage Tip:** Updates going global

Updates are not just restricted to updating files inside of the RenderPal V2 directory. In fact, you can do pretty much everything that can be done with a Python script. While this has a lot of potential, it can also be dangerous, so be careful when creating updates that perform actions on a "global" scope.

All update script functions are contained in a module called `RpUpd`, so an update script has to import this module at the beginning (`import RpUpd`). The following table lists all available update script functions:

| Function/Arguments | Description |
|---|---|
| **GetVersion()** | Returns the current RenderPal V2 version (build number) as an integer. |
| No arguments | |
| **GetDestDir()** | Returns the destination directory as a string. |
| No arguments | |
| **GetSourceDir()** | Returns the source directory as a string. |
| No arguments | |
| **IsType(*type*)** | Returns `True` value if the environment matches the specified type. |
| `type:` string | The type to check for; can be **Server**, **Client** or **RemoteController**. |
| **IsOS(*os*)** | Returns `True` if the environment matches the specified operating system. |
| `os:` string | The OS to check for; can be **Windows**, **Linux** or **Mac**. |
| **IsConsole()** | Returns `True` if the update is applied to a console based component. |
| No arguments | |
| **IgnoreErrors(*ignore*)** | Set whether to ignore any occurring errors (in the update script functions). |
| `ignore:` boolean | Determines whether to ignore errors or not. |
| **CopyFiles(*src, dest, overwrite*)** | Copies the specified files to the destination directory, optionally overwriting any existing files. |
| `src:` string | The source files to copy; wildcards (`*`, `?`) are allowed. |
| `dest:` string | The destination directory. |
| `overwrite:` boolean | If set to `True`, existing files will be overwritten. |
| **MoveFiles(*src, dest, overwrite*)** | Moves the specified files to the destination directory, optionally overwriting any existing files. |
| `src:` string | The source files to move; wildcards (`*`, `?`) are allowed. |

| Function/Arguments | Description |
|---|---|
| `dest:` string | The destination directory. |
| `overwrite:` boolean | If set to `True`, existing files will be overwritten. |
| **RenameFile(*src, dest*)** | Renames the specified file. |
| `src:` string | The source file to rename (including path). |
| `dest:` string | The new file name (without path). |
| **DeleteFiles(*files*)** | Deletes the specified files. |
| `files:` string | The files to delete; wildcards (`*`, `?`) are allowed. |
| **Execute(*command, wait*)** | Executes the specified command; usually used to launch external programs. |
| `command:` string | The command to execute. |
| `wait:` boolean | If set to `True`, the script will wait until the command has been executed. |
| **CopyDir(*src, dest, overwrite*)** | Copies the specified directory and all of its contents to the destination, optionally overwriting any existing files. |
| `src:` string | The source directory to copy. |
| `dest:` string | The destination directory. |
| `overwrite:` boolean | If set to `True`, existing files will be overwritten. |
| **CreateDir(*dir*)** | Creates the specified directory. |
| `dir:` string | The directory to create. |
| **DeleteDir(*dir*)** | Deletes the specified directory and all of its contents. |
| `dir:` string | The directory to delete. |

As an example, we will now show you a script that we internally use to replace all executable files of RenderPal V2:

**Example:** Replace executable files

```
import RpUpd

if RpUpd.IsType("Server"):
      RpUpd.CopyFiles("$UpdDir/RenderPalSvr.exe", "$InstDir", True)
elif RpUpd.IsType("Client"):
      RpUpd.CopyFiles("$UpdDir/RenderPalCl.exe", "$InstDir", True)
elif RpUpd.IsType("RemoteController"):
      RpUpd.CopyFiles("$UpdDir/RenderPalRc.exe", "$InstDir", True)
```

This script checks for the RenderPal V2 type and copies the corresponding executable, overwriting the existing one. The **TerminateApp** flag in the update description is set to **Yes**, so that RenderPal V2 will be terminated before applying the update.

# 27. Net job event examples

The possible uses of net job events are limitless, so we will just show you a small number of examples to give you some inspiration.

**See also:** Net job events

## 27.1 Program example

This example will create a zip file of all rendered images using WinRar (any other archiver does the job just as well). Since we want to zip **all** rendered images, we will implement it as a server-side post-execution net job event.

**Example:** Archive all rendered images

```
Executable: WinRar.exe
Arguments:  A -afzip "F:\Archives\$(NetJob.Name).zip" "$(RenderSet.OutputDir)\*.jpg"
```

The interesting arguments here are the archive filename and the files to add. The archive that will be created will use the net job's name (plus the .zip extension); all .jpg images from the output directory specified in the render set will be added to this archive. If our net job is called "MyNetJob", and the output directory is "M:\QuarksAndStuff", the resulting command-line will be:

```
WinRar.exe A -afzip "F:\Archives\MyNetJob.zip" "M:\QuarksAndStuff\*.jpg"
```

We could now also use the resulting archive in further net job events (maybe copy it to multiple locations, or upload it to an FTP server with a Python script event).

## 27.2 System command example

This example will clear the temporary files folder before rendering a job. This will be a client-side pre-execution net job event.

**Example:** Clear temporary files

```
Command: del /Q /S %TEMP%
```

We do not use any variables in this event; this is just to show that it is not always necessary to use them.

## 27.3 Python script example

We will now take that idea from the program example and upload the created archive to an FTP server. This event has also to be implemented as a server-side post-execution net job event and should be added right after the program event. Note that this is a simplified example; things like error-checking were left out for clarity.

**Example:** Upload archive to FTP server

```
import ftplib
import os

# Create the FTP object, connect to our host, login and set the working directory
ftpObj = ftplib.FTP('ftp.myhost.com')
ftpObj.login('tak', 'mysecret')
ftpObj.cwd('imgarc/tak/')

# Create a new directory, using the net job ID as its name
# Don't forget the quotes, since this method wants a string
ftpObj.mkd('$(NetJob.NetJobID)')
```

**Example:** Upload archive to FTP server

```
ftpObj.cwd('$(NetJob.NetJobID)')

# Store our archive file name
arcFile = 'F:\\Archives\\$(NetJob.Name).zip'

# Upload the file and disconnect
ftpObj.storbinary('STOR %s' % arcFile, open(arcFile, 'rb'))
ftpObj.quit()
```

Easy, isn't it? With two small events, you have just zipped all images and uploaded them to an FTP server.

## 27.4 Further ideas

There are many more things you can do with net job events, and the above examples barely show their true power. You could, for example, create web statistics of your render farm, copy textures around before a job starts, update some kind of database or reboot a client once it has rendered a job.

# 28. The RenderPal V2 database

RenderPal V2 stores most of its data in a database. While this database should never be modified directly, it can be used to query various information about the different objects (like clients and net jobs) of your render farm. This is an ideal way to create reports or integrate statistics of your farm into a web page.

RenderPal V2 currently uses SQLite 3; to get more information about this database (including links to viewers), visit http://www.sqlite.org.

If you have any questions regarding the database layout, feel free to contact us.

# 29. Command-line switches

This chapter will show you all available command-line switches for the graphical components of RenderPal V2. Most of these switches are especially useful during installation and setup. Note that these switches do not apply to the console versions of RenderPal V2; these come with their own switches (which were already described).

## Common switches

The following switches are available in all components of RenderPal V2:

| Name/Syntax | Description/Parameters |
|---|---|
| **quit** | RenderPal V2 will quit after processing the command-line; only useful in conjunction with other command-line switches. |
| `-quit` | No parameters. |
| **silent** | If specified, no messages will be shown for other command-line operations ; only useful in conjunction with other command-line switches. |
| `-silent` | No parameters. |
| **minimize** | Minimizes RenderPal V2 to the system tray on start. |
| `-minimize` | No parameters. |
| **env** | Either registers or removes the RenderPal V2 path environment variable for the specific component. |
| `-env:register` | Registers the path environment variable. |
| `-env:unregister` | Removes the path environment variable. |
| **alias** | Sets the alias/name of the RenderPal V2 component. |
| `-alias:<name>` | `<name>:` The new name. |

## Service switches

The RenderPal V2 services, which is available in the server and the client, can also be controlled using the command-line:

| Name/Syntax | Description/Parameters |
|---|---|
| **service** | Used to control the RenderPal V2 service. |
| `-service:start` | Starts the RenderPal V2 service (if it has been installed). |
| `-service:stop` | Stops the RenderPal V2 service (if it has been installed). |
| `-service:install(user, pwd,dep,restart, scan)` | Installs the RenderPal V2 service. This switch can take various parameters:<br><br>`user:` The user name under which the service shall run.<br>`pwd:` The password for the specified user.<br>`dep:` Name of a service for which the RenderPal V2 service should wait.<br>`restart:` If set to yes, the service will automatically restart RenderPal V2 if terminated.<br>`scan:` If set to yes, RenderPal V2 will automatically scan for any existing shares.<br><br>All parameters are optional and can be left empty. If the parameter list contains spaces **anywhere**, enclose the entire `install(...)` string with quotes. |
| `-service:uninstall` | Uninstalls the RenderPal V2 service if installed. |

**See also:** Autostart

### Client switches

The following switches are client specific:

| Name/Syntax | Description/Parameters |
|---|---|
| **port** | Overrides the client listening port. |
| `-port:<port>` | `<port>:` The new port number. |
| **heartbeat** | Sets the server address for heartbeat sending. |
| `-heartbeat:<address>` | `<address>:` The address of the server to send heartbeats to. If no port is specified, the default one will be used. |

# 30. Renderer instructions

While most renderers can be used without any special instructions, there are still some that require further explanation. These will be given in this chapter.

## 30.1 After Effects

Adding After Effects jobs to your render farm is a pretty simple and straightforward task. The easiest way is to fill the render queue in your After Effects project and add the project file to the render set in RenderPal V2. Create a new net job, and all jobs in the queue will be rendered by one (and only one) machine on the network.

Dispatching a job to more than one computer requires a bit more work, though.

In After Effects, you will have to specify to render an image sequence in the output module settings; rendering movie files will not work. You will also have to check the ”Use Comp Frame Number” option. In the render set, you have to set the name of the composition you want to render, as well as the frame range (start and end frame). Create a new net job and set the frame splitting accordingly.

# 31. Using RenderPal V2 with Virtualization and Wine

## Virtualization

RenderPal V2 can be used under a ”virtual” operating system without any problems or extra precautions. This allows you, for example, to run the RenderPal V2 Server (which requires Windows) under a Macintosh - all you have to do is to use a virtualization software, let Windows run in a virtual machine and install RenderPal V2.

A good and free virtualization software is VirtualBox (http://www.virtualbox.org).

## Wine

Wine (http://www.winehq.org) is an easy to use and well-functioning Windows emulator for Linux and Macintosh. Since version 2.0.1, RenderPal V2 is compatible with Wine, so you can use all GUI components (server, client and remote controller) under Linux and Macintosh.

There are no special requirements for using the Windows versions of RenderPal V2 with Wine; see the Wine website for installation and usage details. RenderPal V2 can then be used just like under a true Windows system (this also includes using the RenderPal V2 setup).

Our user forum also contains a section dedicated to using RenderPal V2 with Wine.

**Important:** Wine support

Wine support is still in an experimental stage, so not everything might work as expected when using RenderPal V2 with Wine. If you encounter any problems, please report them in our user forum.

# 32. Tips & Tricks

This chapter will give you a few tips & tricks about using RenderPal V2 that might not come to ones mind immediately. We will update this section frequently, so you should check back whenever there is a new RenderPal V2 release.

## 32.1 General

### RenderPal V2 path environment variables

RenderPal V2 will register a global environment variable containing its path for each component: `RP_SERVER_DIR` for the server, `RP_CLIENT_DIR` for the client and `RP_REMOTECONTROLLER_DIR` for the remote controller. These can then be used in batch files, for example.

### Environment variables

A features that is often forgotten are environment variables. Every operating system supports them, and they are especially handy for paths. In RenderPal V2, every path or file can contain environment variables (since these will be handled by the system). For example, you could create an environment variable for the path of a renderer on every system and set the executable of that renderer accordingly (like `%MYRENDERER%\renderer.exe`).

### ”Event-only” net jobs

Net job events are a powerful feature, and they can be used for many things that are not always directly related to a ”real” net job. Creating such event-only net jobs (in other words, net jobs that don’t render anything, but will execute some events) is easy. Create a renderer that pretty much does nothing and set up the net job accordingly. A good renderer candidate (for Windows) is **cmd.exe** (the Windows command prompt). If used in conjunction with the **/C** switch, it will just pop up and exit immediately. Add some events to the net job, set all others settings to your likings and you have an event-only net job.

## 32.2 RenderPal environment variables file

When started, RenderPal V2 will import a file called **RenderPal.env** containing a list of environment variables. These variables will then be ”seen” by all child processes launched through RenderPal V2 (mainly the renderers), so this file is an easy way to specify environment variables that should be used by the renderers. Each line contains a single variable assignment in the form of `variable=value`.

**Example:** RenderPal.env

```
MI_CUSTOM_SHADER_PATH=W:\_MentalRayShader\lib;W:\_MentalRayShader\include
MI_MAYA_SOCKETS=1
```

# Z. Appendix

Finis operis

**Shoran Software**
Heidekamp 1
48268 Greven
Germany

**Phone:**
+49 (0)2571 - 953 353

**Fax:**
+49 (0)321 - 213 188 78

**E-Mail:**
contact@shoran.de

http://www.renderpal.com
http://www.shoran.de