Getting Started with NVIDIA Gelato

October 31, 2006

Introduction

NVIDIA® Gelato® is a film-quality, final-frame renderer designed to leverage the power of an NVIDIA Quadro® FX Graphics Processing Unit (GPU). Gelato has been developed by adhering to a fundamental principle: never compromise on the quality of the final rendered image. Gelato delivers images that meet the most rigorous demands of professional high-quality rendering markets, such as film, television, industrial design, and architecture. It features smooth anti-aliasing, adaptive tessellation, beautiful motion blur, layered shaders, and the full range of geometric primitives (NURBS, subdivision surfaces, points, curves, and procedurals). It also provides for flexible shading and lighting, including global illumination with ray-traced reflections and shadows, indirect illumination, and ambient occlusion.

Gelato was designed for fast and seamless integration into a digital production pipeline. Gelato ships with a simple, but powerful C++ API that can be used to integrate Gelato into existing production pipelines and to create plug-ins for other pipeline products. The Gelato API is fully documented (no proprietary APIs) and royalty free.

Gelato comes in two versions, basic Gelato, which is free of charge, and Gelato Pro, which is available with payment of a license fee. Both versions contain all the features needed to create images of the highest quality. Gelato Pro adds features that are useful in a professional production environment and comes with a comprehensive support package from NVIDIA. The features that are unique to Gelato Pro are:

- Sorbetto™ interactive re-rendering
- Network-parallel rendering
- Multithreading
- Native 64-bit support
- DSO shadeops
- Advanced access to new features and beta tests

Both Gelato and Gelato Pro use the same binary distribution files. Access to Gelato Pro features is controlled via a license file. No license file is required for Gelato, only Gelato Pro. See Licensing below.

Both basic Gelato and Gelato Pro ship with Mango™ and Frantic Films' Amaretto™, plugin rendering modules for Autodesk® Maya® and Autodesk 3dsmax®, respectively. Mango and Amaretto embed Gelato as a new renderer and provide an interface for assigning and adjusting shader parameters using the Maya and Max interfaces. Please see the Mango User's Guide and Amaretto help files for more details.

Visit GelatoZone for complete information about NVIDIA Gelato. This *Getting Started* guide provides a quick overview of Gelato installation, configuration, and basic functionality. For a more complete reference, see the Gelato Technical Reference Manual.

System Requirements

Basic Gelato and Gelato Pro have similar requirements. Gelato Pro is supported only on NVIDIA Quadro graphics cards. Basic Gelato runs on NVIDIA GeForce or Quadro graphics cards.

Operating System	• RedHat Linux 7.2 (32- or 64-bit) • Linux SuSE 8.0 (32- or 64-bit) • Windows® XP Service Pack 2 • Windows® XP Professional x64 (32-bit mode only)
CPU	• Intel Pentium III or higher • Intel Xeon • AMD Athlon • AMD Opteron
GPU	• NVIDIA Quadro® FX (Basic or Pro) • NVIDIA GeForce® 5200 or higher (Basic Only)
Graphics Driver	• Linux 32-bit: x86-1.0-8774(*) or later • Linux 64-bit: x86_64-1.0-8774(*) or later • Windows® XP: 91.36 or later • Windows® XP Professional x64: 91.36 or later
DCC Applications	• Autodesk Maya 7 & 8 • Autodesk 3D Studio Max 6, 7, & 8

You can download the latest drivers from NVIDIA's website.

NOTE: Native 64-bit operation is a Gelato Pro feature. If you wish to run basic Gelato on a 64-bit system, you must run the 32-bit version of the software.

There are no minimum memory requirements as Gelato is designed to effectively use all available system and GPU memory. However, it should be clear that larger scenes will perform better with more CPU memory. We recommend at least 2 GB of main memory, though Gelato requires considerably less for smaller scenes.

Gelato Pro requires that the system contain a Quadro FX graphics card, but there is no requirement for a display. Gelato Pro can be run on headless server machines. Our image viewer is designed to work remotely and interactively display images on a remote animator machine as they are computed.

Linux Notes

Gelato is supported on the following versions of Linux:

- RedHat 7.2, 7.3:
  - Copy the files from `$GELATOHOME/lib.compat` to `$GELATOHOME/lib` and add `$GELATOHOME/lib` to `$LD_LIBRARY_PATH`.
- RedHat 9, RedHat Enterprise Linux 3
  - No additional instructions.
- Suse 9.x, RedHat Enterprise Linux 4, Fedora Core, etc.: Gelato requires Python 2. which is different from what these distributions provide. You can check your distribution provides with:

  ```
  rpm -q python
  ```

  To install Python 2.2 (newer builds use Python 2.4):
  - Download Python from: python.org
  - mv /usr/bin/python /usr/bin/python.backup
  - mv /usr/share/man/man1/python.1.gz /usr/share/man/man1/python.1.gz.backup
  - cd /usr/tmp
  - tar zxvf Python-2.2.3.tgz
  - cd Python-2.2.3
  - ./configure --prefix=/usr
  - make install
  - mv /usr/bin/python.backup /usr/bin/python
  - mv /usr/share/man/man1/python.1.gz.backup /usr/share/man/man1/python.1.gz
  - ln -s ../lib/python2.2 /usr/lib64/python2.2 *[64-bit systems only]*

Installation

Simply run the appropriate executable for your operating system (*.exe for Windows; *.run for Linux). Gelato should install as any other application and simply follow the prompts. See below for OS-specific installation details.

Linux	• Install using: "`sh` *installername*`.run`" • By default the gelato installer will place the distribution into `/opt/nvidia/gelato`, though you may override this when installing. • Make sure you are running the installer with permissions to create and/or write to this directory (or whichever alternate installation directory location you choose). • You will need to set the environment variable `GELATOHOME` to point to the directory where Gelato was installed.
Windows XP	• Install (using the installer .exe file) • The installer will set GELATOHOME (but you'll need to reopen any command shell windows) • The installer will add %GELATOHOME%/bin to your path

You will be given the option to install three components:

- Gelato & Mango plug-in for Autodesk Maya
- Python (Note that the newest versions of Gelato use Python 2.5. If you were using an older version of Python included with previous versions of Gelato, you may wish to reinstall this.)
- Amaretto plug-in for Autodesk 3ds Max (Note: you must have 3ds Max already installed for Amaretto to install properly.)

If you do not have Python installed, you will need it to read in Python-for-Gelato, or PYG files, which are used in the examples. Mango, the Gelato plugin for Alias' Maya, is included automatically. If you will be using Autodesk's 3DS Max, you should install Amaretto, the Gelato Plugin for 3DS Max.

When installing Gelato, it will ask what version of Autodesk Maya you use, 7 or 8. If you do not use Maya, just select one; it does not matter which. You can change this selection after installation if you change Maya versions later on. This setting is used to configure the environment variables so that Mango will properly load into Maya.

The Windows installer will also add `MANGOHOME` and the Maya-related environment variables mentioned in the User Configuration section. You will need to add these variables to your environment manually under Linux.

Licensing

Basic Gelato does not require a license key to run. If you are only using Basic Gelato, you can skip forward to the Check Gelato section below. Without a license key, Gelato Pro features will be disabled, with the exception of Sorbetto interactive relighting, which will run for a limited number of times. A valid license key is required to use the Gelato Pro-only features. If you would like to purchase Gelato Pro, you may do so at the NVIDIA Products Store or from an authorized reseller. A list of authorized resellers can be found at NVIDIA's Where To Buy Web Page. Or contact gelatosales@nvidia.com.

Gelato Pro uses Macrovision's FLEXlm package for licensing. Local, nodelocked installations contain license keys that are matched to the MAC address or FLEXid dongle of each machine. The license file shows the associated MAC/dongle address in the HOSTID string. Multiple node locked and floating licenses can exist in a single file. The license you receive for Gelato Pro should include all of the nodelocked licenses in a single file.

You will receive your license via email from gelatosupport@nvidia.com. It will be properly configured for your setup. Please install the license file in the following location:

Version	License File Location
Gelato	No license required
Gelato Pro - Windows	C:\flexlm
Gelato Pro - Linux	/var/flexlm

Once installed, please run Check Gelato to confirm that you are running a correctly licensed version of Gelato Pro. Check Gelato will report whether Gelato is running in Basic or Pro mode, and the locations searched and license file used for Gelato Pro. Gelato will search the directories above and attempt to use any file with the `.lic` extension. If you do not have permission to install license files in the above locations, you can optionally install the license file in `$GELATOHOME/etc`

You can use the `lmhostid` program, found in the etc subdirectory of the Gelato installation, to discover the FLEXlm host id for the local machine. Other FLEXlm utilities can be found in the same directory to aid with starting and debugging the license server process. The `lmtools` program provides a graphical user interface which can help with license file installation and license server setup.

Floating Licenses

Gelato Pro uses Macrovision's FLEXlm package for licensing. Gelato Pro floating licenses can be used cross-platform (so a single pool of licenses can be used by both Windows and Linux machines). The license file shows the MAC address of your license server in the HOSTID string.

Local machines that use a shared floating license server must have a license file on the local machine with the following syntax. Note that the license file you receive will include this so that the same file can be used on both the local and server machines.

```
SERVER [machine_name] [mac#] [port#]
USE_SERVER
```

Floating license files will already include this information, so you should not need to modify the license file you receive in any way. The mac# is usually set to zero, and the port# is not required for a standard installation. For alternate locations use the environment variable `LM_LICENSE_FILE` or `NVIDIAD_LICENSE_FILE`. For floating licenses you can use a license file that points to the server or set one of the environment variables (replace host-name with your license server):

```
export LM_LICENSE_FILE=@host-name
```

The license server software is located in the $GELATOHOME/etc directory. You can test the license server with this command:

```
$GELATOHOME/etc/lmgrd -z -c licensefile
```

That will show debugging information about when a license is checked out and in. For regular usage `lmgrd` will generally be run automatically at startup and without the -z flag.

Please consult the FLEXlm documentation or contact support at gelatosupport@nvidia.com for more information. Further general information on FLEXlm licensing, including an end-user's manual, is available from Macrovision customer support.

Check Gelato

After installation is complete, run the Check Gelato utility from the Start Menu. Or for Linux, execute the following command in a shell:

```
"$GELATOHOME/bin/gelato" -check -verbosity 2
```

The Check Gelato utility will give you a summary of:

- Gelato version
- Mode (basic or Pro)
- NVIDIA driver status
- Errors and warnings, if any.

If your NVIDIA display driver is an older version, you are likely to get error and warning messages. You can download the latest drivers from NVIDIA's website.

If the program fails to run and displays a licensing error, please configure the FLEXlm license. To purchase a Gelato license, please contact the Digital Film Group sales at gelatosales@nvidia.com.

If the program displays some other error, possibly describing problems with the system or hardware configuration, please check the Release Notes, or contact gelatosupport@nvidia.com for more information. Please visit the online public forum for gelato issues, general discussion, tips and related products.

Please make sure that your system is configured to use an NVIDIA Quadro FX GPU. Running gelato, as described above, will only work if your system hardware supports all of the OpenGL features required by gelato. The installation contains a NVIDIA OpenGL driver. Please make sure that you upgrade your gelato systems to use the latest drivers.

User Configuration

Under Windows, the Gelato installer will automatically update the system variables appropriately. It will add a `GELATOHOME` environment variable that points at the installation directory. The `MANGOHOME` environment variable will point at the version of Mango selected during the install. `MANGOHOME` is also added to the `MAYA_SCRIPT_PATH`, `MAYA_PLUGIN_PATH` and `XBMLANGPATH` so that Maya correctly loads and displays icons for Mango.

Under Linux, The `GELATOHOME` environment variable must be set to point at the top level of the distribution directory. You may also want to update the Maya environment variables `MAYA_SCRIPT_PATH`, `MAYA_PLUGIN_PATH` and `XBMLANGPATH` so that Mango is properly loaded into Maya. For example, in `bash` you may use:

```
export MAYAVER=8.0
export GELATOHOME=/opt/nvidia/gelato
export MANGOHOME="$GELATOHOME/mango/maya$MAYAVER"
export MAYA_SCRIPT_PATH="$MANGOHOME/scripts;$MAYA_SCRIPT_PATH"
export MAYA_PLUG_IN_PATH="$MANGOHOME/plug-ins;$MAYA_PLUG_IN_PATH"
export PYTHONPATH="$GELATOHOME/lib;$PYTHONPATH"
export XBMLANGPATH="$MANGOHOME/icons;$XBMLANGPATH"
```

Gelato requires an OpenGL driver to run. Under Linux, this means that you must have X running before Gelato can run -- even on a headless server machine. In addition, Gelato only runs on the local, hardware accelerated X server (however you can run it on a server and display the results remotely). This implies that you must set the following environment variable:

```
export DISPLAY=:0
```

For full documentation on how to run Gelato on a headless server and display the results on a remote machine, please see the iv section of the Technical Reference.

When performing -net rendering on machines with multiple GPUs, having your shell set to bash may be required...

You may also need to add `$GELATOHOME/lib` to both the `$LD_LIBRARY_PATH` and `$PYTHONPATH` environment variable to if you are using an older version of Red Hat Linux (prior to version 9), do any program development, or are using Mango.

SSH

Gelato uses ssh when the image viewer, `iv`, is run remotely on a server while displaying the results on a local client machine. Gelato also uses ssh in network rendering mode. This section describes how to setup ssh for Gelato.

Note that you do not have to be logged in to the client machine under Windows to use it as a render node. Under Linux, you must have X started, and the X server must allow the user to have access for the render node to function. This means you need to either be logged in as the same user, or you need to have the X authority security mode allow remote use of the display.

SSH Under Windows XP

We recommend using **OpenSSH** under Windows. The latest version of OpenSSH ships as part of **cygwin**. Cygwin is not required by Gelato, but remote iv and network rendering require some form of ssh to function.

When installing Cygwin you need to install for "All Users" - if you have already installed, you can re-run setup and change the setting.

You must allow sshd to start up interactive applications. This can be done by modifying the `ssh-host-config` installation script, or [after running ssh-host-config] through the System Services control panel under Administrative Tools -> Component Services -> Services (Local) -> CYGWIN sshd -> Log On tab -> Allow service to interact with desktop.

When setting up ssh under Windows, you must install the sshd daemon. The script that comes with the OpenSSH cygwin package is a good start (`/bin/ssh-host-config`). You should modify the script to allow the sshd service to run interactive programs. This is done as part of the `cygrunsrv -I` installation line in the script. On the **two** lines in the script containing the `cygrunsrv -I`, add --interactive.

Change:

```
if cygrunsrv -I sshd -d "CYGWIN sshd"
```

into

```
if cygrunsrv -I sshd --interactive -d "CYGWIN sshd"
```

Then run `ssh-host-config`. When prompted about privilege seperation, either option is fine.

Make sure that the `/etc/passwd` *and* `/etc/group` files are correctly set up with your proper Windows *user and group ids* and a valid *home directory*. Note that the home directory filename can have spaces, but should not prepend the spaces with backslashes. Under Windows, you'll need to have GELATOHOME set as a system-wide environment variable, since ssh does not run your shell startup files when run with a command. Also note that changes to system environment variables are *not used* by ssh until after a complete system reboot. If you get a `setgid` error message when logging in to a machine, there is a mismatch between the group id in the passwd file and the available groups in `/etc/groups`.

Without the proper `/etc/passwd` and user account settings, `iv` and networked rendering with Gelato **will not work**. After running the configuration script, the following command will fix up the password file:

```
mkpasswd -c -l > /etc/passwd
```

This command will fix up the group file:

```
mkgroup -c -l > /etc/group
```

SSH Without A Password

To setup SSH to work transparently among multiple machines (for remote -iv viewing):

On the first system:

1. `ssh-keygen -t dsa -f ~/.ssh/id_dsa -N ''`
2. `cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys2`

To test:
`ssh $HOSTNAME`

You will probably be prompted about the authenticity of the host (respond yes) and then give a shell prompt (potentially in a different directory) without prompting for a password. Type exit to logout.

On the second system:

(replace *first* with hostname of first system)

1. `ssh-keygen -t dsa -f ~/.ssh/id_dsa -N ''`
2. `scp first:~/.ssh/authorized_keys2 ~/.ssh`
3. `cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys2`
4. `scp ~/.ssh/authorized_keys2 first:~/.ssh`

With the last step, you will probably be prompted about the authenticity of the host (respond yes). Then the file should copy without prompting for a password.

To test (from the second system):

```
ssh first
ssh second
```

The first command should give you a shell prompt on the first system without prompting for a password. With the second command, you will probably be prompted about the authenticity of the host (respond yes) and then give a shell prompt on the second system. To logout of the second system, type exit. To logout of the first system, type exit.

On the nth system:

1. follow the above steps for the second system
2. scp ~/.ssh/authorized_keys2 *second*:~/ssh
3. scp ~/.ssh/authorized_keys2 *third*:~/ssh
4. ...

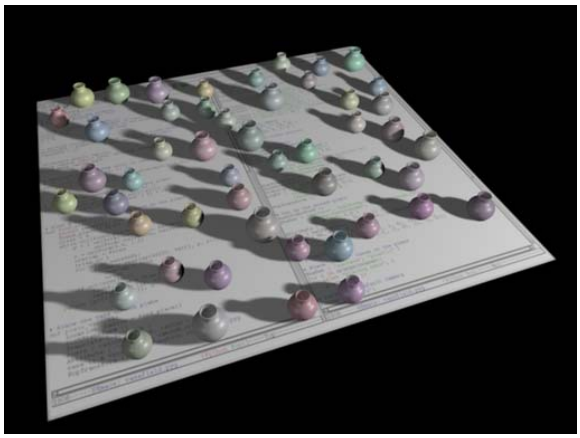
Setup Tips

- Make sure to check the /etc/passwd and /etc/group files if you have problems logging in.
- Make sure to check the permissions of the ~/.ssh directory and the files within it if you can log in, but can't get it to work without a password. The directory should have read, write and execute permission for the owner, but only read access for everyone else (755). The files within the directory should have read & write permission for the user, but only read permission for everyone else (644). The only exception is the id_dsa file which should have read & write permission for the user, and **no** permissions for anyone else.
- Edit /etc/motd to change the default login message from Fanfare!! You are successfully logged in to this server!!

Your First Image with Gelato

Once your system is correctly configured, you should be able to run the Gelato example programs. Start by testing the Pyg scene generator using the `vasefield` example:

```
cd ~
cp -
r $GELATOHOME/examples/vasef
ield .
cd vasefield
./run_me
```



We start by copying the example to your home directory to avoid modifying the original distribution directory. This shell script creates a texture map with the tool `maketx`, then runs the `gelato` renderer twice - once to render a shadowmap, and once to render the final image. As the final image is rendered, it displays the partially-completed image in our image-viewer tool, `iv`. The resulting image should look like this image.

You can compare your rendered image to the full-size reference image `vasefield_ref.tif`. You can do this by starting a new image viewer with both the newly computed and reference image:

```
$GELATOHOME/bin/iv vasefield.tif vasefield_ref.tif
```

Use the Page Up and Page Down keys to show each image. The image viewer contains many useful features which allow you to compare images. These images happen to be identical, so they aren't particularly useful. Please see the Technical Reference for full iv documentation.

We encourage you to look through the files of this example, to familiarize yourself with Gelato and the files it uses. The ".pyg" scene files we use here are in Gelato's Python-based "Pyg" format, described below. Try running another Pyg example which demonstrates *ambient occlusion*:

```
cd ~
cp -
r $GELATOHOME/examples/ambi
ent-occlusion .
cd ambient-occlusion
./run_me
```

If you run this test a second time, it will run faster because it is using the ambient occlusion *disk cache*, `occ.sdb`, created on the first run.

Other examples included in the distribution demonstrate delayed loaded generators, and C++ API interaction.

Configuration Files

The applications included in your distribution can read in user-specific startup files to load custom preferences. These configuration files can be used to set various studio- or job-wide paths or user interface preferences.

iv

The iv image viewer application reads the file `${HOME}/.ivrc` on startup. This file can be used to set user-specific settings for the viewing tool. Full documentation can be found in the technical reference. The most common use for this file is to set the gamma value associated with each input image file format.

Iv will gamma correct each image type according to the settings in the `~/.ivrc` configuration file. By default, no gamma correction is applied to the images displayed in iv (ie. gamma=1). If you prefer to have all TIFF files displayed with a gamma of 2.2, add the following line to your `~/.ivrc` file:

```
gamma.tif = 2.2
```

Similarly, if you would like JPG files to be displayed with a gamma of 1.8, use:

```
gamma.jpg = 1.8
```

If you would like to have all images displayed with a gamma of 2.2 without regard to their filename extension, use:

```
gamma = 2.2
```

Additional settings that control the sorting of multiple images, the filename which is used when saving images or regions from iv, the background color, and auto-zoom behavior can also be set in the configuration file. Please see the technical reference for more detail about this configuration file and iv use in general.

Full documentation of iv and the iv config file are available in the Technical Reference and in the help files.

gelato

Gelato automatically reads the *Pyg*-format file `~/.gelato.pyg` on startup. Pyg is Gelato's Python-based scene format, described later in this document. The `.gelato.pyg` file should be used to set user-specific attributes. Any valid Pyg statements may be included in this file, but, typically it is used to set up user paths and defaults. For example, you can add additional directories to the generator path that Gelato uses to find generator DSO plug-ins using the following statement:

```
Attribute ("string path:generator", "/home/guest/gelato/lib:&")
```

You may also override any of the default quality settings. However, please note that the `~/.gelato.pyg` file is the first file read by Gelato, so subsequent files will override these values. For example, you can double the default shading quality using:

```
Attribute ("float shadingquality", 2)
```

Please see the API specification in the Gelato Technical Reference Manual for a full list of attributes, and the Pyg documentation for further details of the syntax of Gelato's Python input files.

Files Included In Distribution

The gelato distribution includes the following files:

`bin/`	
`gelato`	*The renderer*
`gslc`	*The shader compiler*
`gsoinfo`	*Shader interrogator*
`guicat`	*Small GUI program for displaying error logs, used by Mango.*
`idiff`	*Image difference tool*
`interleave`	*Merges two field images into one frame*
`iv`	*Image viewer*
`maketx`	*Texture creator (from ordinary image files)*
`topyg`	*Converts scenes (eg. RIB) to Pyg*
`libgelato.dll`	*(Windows Only) Gelato shared library for application development*
`gslpp`	*(Windows Only) Preprocessor for shader compilation*
`*.dll`	*(Windows Only) Required shared libraries (pthread, python)*
`doc/`	
GettingStarted.html	*This document*
ReleaseNotes.html	*Current release notes for this version*
History.html	*Previous release notes*
techref.pdf	*Full technical reference (programmer's guide) for Gelato and its API's*
EULA.txt	*End user license agreement*
mango/index.html	*User's Guide for Mango, the Maya plugin*
iv/iv.html	*User documentation for `iv`, the image viewer utility*
shaders/	*User documentation for all the example shaders that come with Gelato.*

technotes/	Various technical notes and whitepapers.
`etc/`	
lmhostid, *et.al.*	*FLEXlm licensing tools*
`NVIDIA-Linux-*`	*NVIDIA OpenGL driver installer, in case you need it*
`lib/`	
`*.imageio.so`	*Image IO plugins for different file formats*
`*.generator.so`	*Scene generator plugins including Pyg, the Gelato Python scene file format*
`libgelato.so`	*The Gelato libraries for C++ program development*
`libgsoargs.a`	*The C++ shader interrogation library*
`libvecmat.a`	*A C++ vector and matrix library*
`examples/`	*Various example scenes, dso-shadeops, and C++ examples*
`include/`	*Public header files for Gelato APIs*
`inputs/`	*Shared input files for example scenes*
`shaders/`	*Surface, light, volume, and displacement shaders*
`textures/`	*Included texture files*
`mango/`	*Mango plugin for Maya (possibly for several different releases of Maya)*

Gelato Interfaces and Scene Files

Unlike many other renderers, Gelato does not have a preferred scene file format. Instead, a streamlined C++ API is provided, to access the renderer's core functionality. This API can be programmed to directly, or a flexible "generator" DSO mechanism may be used to write bindings to different scene file formats or scripting languages. No single scene file format or binding receives preferential treatment in the renderer - generator DSOs are transparently loaded, as appropriate, to process associated scene files specified on the Gelato command line.

Although the C++ scene API is very flexible, it requires considerable programming experience to use effectively. It is generally preferable to define scenes using a file-based scene-description format. A parsing program is then used to read the scene file, and make appropriate C++ API calls based on the file contents. The parser code which associates the scene file format with equivalent C++ API calls is called a "binding".

NVIDIA provides a Python API binding with the distribution, but does not mandate its use in any way. It is straightforward to create bindings to the Gelato API for different scene file formats or scripting languages. NVIDIA strongly encourages such customization, and supports it with the "generator" plug-in API, described below, which allows the "gelato" executable to seamlessly load and use custom API bindings.

A third-party has developed a RIB binding for Gelato which can be found along with other tools at the NVIDIA Film Group's download page. Also available is a converter which can translate shader source code into Gelato Shading Language (GSL).

Generator Plug-ins

Gelato features a simple API, called the "Generator" API, which allows users to easily define parser plug-ins to read custom scene formats. The "gelato" executable will load and run the appropriate generator plug-in, in order to parse one of these scene files. All scene file parsing and execution is carried out through the generator mechanism - the renderer itself has no in-built bindings to any scene file format.

The "generator" mechanism is activated via the Input() C++ API call, which reads and executes a given scene file. Typically, Input() uses the file-extension of the specified file to define a file-type string for the file - e.g. for a file "blah.foo", the file-type string would be "foo". If the renderer does not currently have a generator plug-in loaded which can parse that file-type, it searches for an appropriate plug-in in its "generator" search-path. On Linux, this plug-in would be called "foo.generator.so". The default location for these plug-ins is $GELATOHOME/lib, but the user may set the "generator" search-path to some other set of locations, if desired. This is done by setting the global attribute "path:generator".

For further details on the generator plug-in API, the user is referred to the Gelato Technical Reference Manual.

Pyg: Python for Gelato

The Gelato renderer ships with a fully-supported Python-language binding to the underlying C++ scene API. This binding, called "Pyg", for "Python for Gelato", enables scene descriptions which are highly flexible, since they can leverage the programmability and extension libraries of the Python language. The Pyg binding is implemented as the generator DSO "pyg.generator.so" - to be found in the "lib" directory of the Gelato distribution. NVIDIA hopes that the Gelato user community will find Pyg to be a format which is useful for flexible scene descriptions, and as a vehicle for file interchange within the community.

The Pyg API closely mirrors the C++ API in form and function. This should not be surprising, as Python is an object-oriented language with some syntactic similarity to C++. The user is referred to the Gelato Technical Reference Manual for detailed descriptions of the C++ and Pyg APIs. Here, we just aim to give an impression of Pyg, using a simple example.

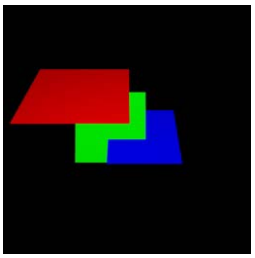
The following Pyg scene (or "program", or "script" - all of these notions are equivalent here) specifies three random quadrilaterals, lit by a point light:

```
# coloredquads.pyg
# Copyright 2004 NVIDIA Corp.
#
# Render three random quads.

# Import standard Python random-number package
import random

# Quadrilateral subroutine
def random_quad( color ):
    PushTransform()
    # Position the upcoming quadrilateral
    Translate ( random.uniform(-1,1),
                random.uniform(-1,1),
                random.uniform(1,5) )
    # Tilt it
    Rotate ( random.uniform(0,60), 1, 0, 0 )

    # Quadrilateral patch - point data first...
    Parameter ( "vertex point P", ((-1, 1, 0), (1, 1, 0),
                                   (-1, -1, 0), (1, -1, 0)) )
    # ...then color...
    Parameter ( "color C", color )
    # ...then geometry call.
    Patch ("linear", 2, 2)
    PopTransform()

# Specify output image file, and default camera "camera"
Output ( "coloredquads.tif", "tiff", "rgba", "camera" )

# Set global attributes
Attribute ( "string path:shader", ".:shaders:&" )
Attribute ( "int[2] resolution", (512, 512) )
Attribute ( "float shadingquality", 1 )
Attribute ( "string projection", "perspective" )
```

```
Attribute ( "float fov", 50 )

# Position the camera
Translate ( 0, 0, 3 )

# Start scene
World ()

PushTransform ()

# Position the light
Translate (0,1,-0.5)

# Set up light
Parameter ( "float intensity", 0.8 )
Light ( "thelight", "pointlight", "float falloff", 0 )

PopTransform ()

# Set surface shader
Shader ( "surface", "matte" )

# Python tuple of color triples
quad_colors = ( (1,0,0), (0,1,0), (0,0,1) )

# Seed the Python RNG
random.seed(1)

# Set up a quad for every color
for color in quad_colors:
    random_quad ( color )

# Render the scene
Render ()
```

This scene file is provided in the Gelato distribution as the file:

```
$GELATOHOME/examples/coloredquads/coloredquads.pyg
```



The scene may be rendered by simply typing `gelato coloredquads.pyg` in that directory. This causes the Pyg generator DSO to be loaded into gelato, which then parses and executes the Pyg code. The resulting image should look like this, only larger:

This example illustrates some salient features of Pyg. First, the scene is a true Python program. It is executed by the standard Python interpreter which is embedded in the Pyg DSO. Typical Python language features which can be seen in the example are the "import" statement at the head of the file, the "def" function definition, the "quad_colors" variable assignment, and the "for" loop. Interspersed among these standard Python features are calls to Gelato API routines such as Output, Light and Patch.

Ignoring the "random_quad()" function definition for the moment, we can see the following pattern of API calls:

- An Output call to set up the image output - in this case, to a TIFF file.
- Various Attribute calls to set global parameters, including default camera settings.
- Transformation calls before the World statement, to position the camera in space (Translate).
- A World call to signal the start of the "world" part of the scene. The "world" consists of geometry objects which the camera may look at (Patch), the appearance of these objects (Shader), and the lights by which to see them (Light). Interspersed among these calls are various positioning commands (Translate, Rotate), and operations on the current-transform stack (PushTransform, PopTransform).
- A final Render call to start rendering the scene.

All Gelato scenes consist of this general sequence of API calls. Users of other renderers may be accustomed to seeing similar scene setups embedded in flat, non-procedural scene files. One interesting aspect of Pyg is, of course, that it is executable Python code. This allows us to use variables, functions and control-flow in the above example. We use the "for" loop to set up three differently-colored and -positioned quadrilaterals, with a call to the function "random_quad()" to define each version. Note also that we make use of the standard Python "random" package, to generate some uniformly-random values. This is just one example of the use which can be made of the huge number of available Python packages.

The C++ API

Gelato features a C++-based scene API which can be used to set up and render scenes. The API is implemented as an opaque C++ class GelatoAPI. A new renderer instance is created using the CreateRenderer() function, and then member functions (instance methods) are used to set that renderer's state and render the final scene. The following example gives an impression of the overall look of the API:

```
#include "renderapi.h"

void main(int argc, char *argv[])
{
    GelatoAPI *r = CreateRenderer();

    r->Attribute( "string projection", "perspective" );
    float shutter[2] = { 0, 1 };
    r->Attribute( "float[2] shutter", &shutter );

    r->Output( "cplusplus.tif", "tiff", "rgba", "camera" );

    r->Translate( 0, 0, 3 );

    r->World();
```

```
    r->PushTransform();

    // Position the light
    r->Translate( 0, 1, -0.5 );

    r->Parameter( "intensity", 0.8 );
    r->Light( "thelight", "pointlight" );

    PopTransform();

    // Set up a red quadrilateral patch
    float P[] = { -1, 1, 0,  1, 1, 0,  -1, -1, 0,  1, -1, 0};
    r->Parameter( "vertex point P", P );
    float col[] = {1, 0, 0};
    r->Parameter( "color C", col );
    r->Patch( "linear", 2, 2 )

    r->Render();
}
```

For further details on the C++ API, the user is referred to the Gelato Technical Reference Manual, where the GelatoAPI class and its functions are fully documented.

Attributes and Parameters

Some discussion of Attribute and Parameter calls is in order. Both of these routines set state in the renderer, but they are different kinds of state, with different scope. Attribute sets a given value into the current attribute state record - a piece of renderer state which is carried along as API calls are made, much like the current variable values in a typical programming language. The attribute state serves as the environment in which various API setup calls are made (e.g., geometry definitions). These calls save the current attribute state which exists when they are defined, and reflect state when they are finally rendered, in their positions, color, or other characteristics. The current attribute state can be pushed and popped on a stack in various flexible ways(PushAttributes, PopAttributes). The current transformation matrix, as affected by Rotate, Translate, PushTransform, PopTransform, etc., is an attribute.

In comparison, Parameter calls do much less. Each call sets one characteristic of the next non-Parameter API call. Parameter has local action - it adds an entry to a "parameter list" which is consumed and cleared by the following API call. There is no stack for Parameters, and they do not retain state from API call to API call. Pyg offers a convenience feature whereby, instead of making explicit Parameter calls before an API call, the parameter name/value pairs may be listed in an inline "varargs" style, as part of the API call itself. For instance, in the example above, the light is set up with twoparameters - "intensity" and "falloff":

```
Parameter ( "float intensity", 0.8 )
Light ( "thelight", "pointlight", "float falloff", 0 )
```

Here, "intensity" is specified with an explicit Parameter call, whereas "falloff" is listed at the end of the Light call in the inline style. One could just as well have written the above example completely with Parameter calls:

```
Parameter ( "float intensity", 0.8 )
Parameter ( "float falloff", 0 )
Light ( "thelight", "pointlight" )
```

or fully in inline form:

```
Light ( "thelight", "pointlight", "float intensity", 0.8, "float
falloff", 0 )
```

The inline form may be more convenient for hand-edited Pyg code. Note, however, that the explicit Parameter form is a closer syntactic match to the equivalent C++ code:

```
GelatoAPI *r;
...
r->Parameter ( "float intensity", 0.8 );
r->Parameter ( "float falloff", 0 );
r->Light ( "thelight", "pointlight" );
```

Currently, all attribute and parameter names in the Pyg API require prefix type-strings - e.g. "float intensity", rather than just "intensity". This requirement does not apply in the C++ API, where such type-strings are typically optional. We plan to make type-strings also optional for Pyg in a future release.

Input API Call

Although not featured in the above example, Pyg implements a binding to the Gelato API's Input command. This call allows other scene files to be executed, using the "generator" DSO mechanism to load the appropriate parser DSO. This means that, for example, given a generator plug-in for a file-type "foo", you would be able to make calls like "Input ('teapot.foo')" from your Pyg file.

Conversion to Pyg: The `topyg` Program

The `topyg` program converts arbitrary scene files (for which there are generator DSOs) into Pyg files. It is typically run like this:

```
topyg -o blah.pyg blah.foo
```

Here, `blah.foo` is a scene file in some format "foo" which can be read by an associated Gelato generator DSO `foo.generator.so`, and `blah.pyg` is the output Pyg version of the scene. "topyg" operates by using a Pyg-emitting GelatoAPI subclass to handle Gelato scene API calls

made by generator DSOs. Essentially, this Pyg-text scene "renderer" is slotted in instead of Gelato's normal image renderer.

Use the `-r` option to make `topyg` expand Input() statements.

Binary Pyg

Gelato supports a binary version of the Pyg file format which can greately improve performance when reading large Pyg files. Binary Pyg is both faster to read and results in smaller files. You can convert existing scenes into binary Pyg using topyg, or by calling the C++ API directly. The format is enabled using the following attribute:

```
Attribute ("int format:binary", 1)
```

Binary Pyg files are restricted to contain only simple, flat Gelato calls, with no Python variables, expressions, or control structures. We call this subset of Pyg "Pyglet". (It is essentially the subset written by `topyg`.)

Pyglet is faster even for non-binary Pyg files, because the Pyg generator processes it immediately, bypassing the standard Python parser. This is particularly important for statements with long lists (such as point coordinates), as the standard Python parser can perform poorly on these. The user can tune the behavior of Pyglet and another Pyg efficiency technique called "chunking"; see the Technical Reference for details.

Efficiency vs. Flexibility

Pyg's proceduralism, flexibility, and access to Python's standard library are distinctive and attractive scene file features. Scene files that use them may take a little longer to run than an equivalent scene in a non-procedural file format. Flat file size is of the same order as non-procedural formats - perhaps 10% or so larger. However, some procedural Pyg scenes can of course be much smaller than non-procedural equivalents.

One approach is to use full procedural Pyg for "master" scene files, where the flexibility of Pyg can allow sophisticated control over larger subsidiary scene files (such as geometry) which may be stored in some more optimized form. Gelato's DSO-based Input call allows all these different file types to coexist transparently.

Copyrights and Trademarks

Gelato® is a trademark of NVIDIA Corporation. The Gelato software is:

Copyright © 2004-2006, NVIDIA Corporation. All rights reserved.

The TIFF library used by default in Gelato is:

Copyright © 1988-1997, by Sam Leffler
Copyright © 1991-1997, by Silicon Graphics Inc.

The OpenEXR library used by Gelato's OpenEXR imageio plugin is:

Copyright © 2006 Lucas Digital Ltd. LLC.
OpenEXR, Industrial Light & Magic and ILM are trademarks and service marks of Lucasfilm Ltd.; all associated intellectual property is protected by the laws of the United States and other countries. All rights reserved.
OpenEXR source code is available from http://www.openexr.com.

The DevIL library used by Gelato's DevIL imageio plugin is:

Copyright © 2001-2002 Denton Woods.

DevIL source code is available from http://openil.sourceforge.net.

Copyright © 2004-2006, by NVIDIA Corporation. All Rights Reserved.