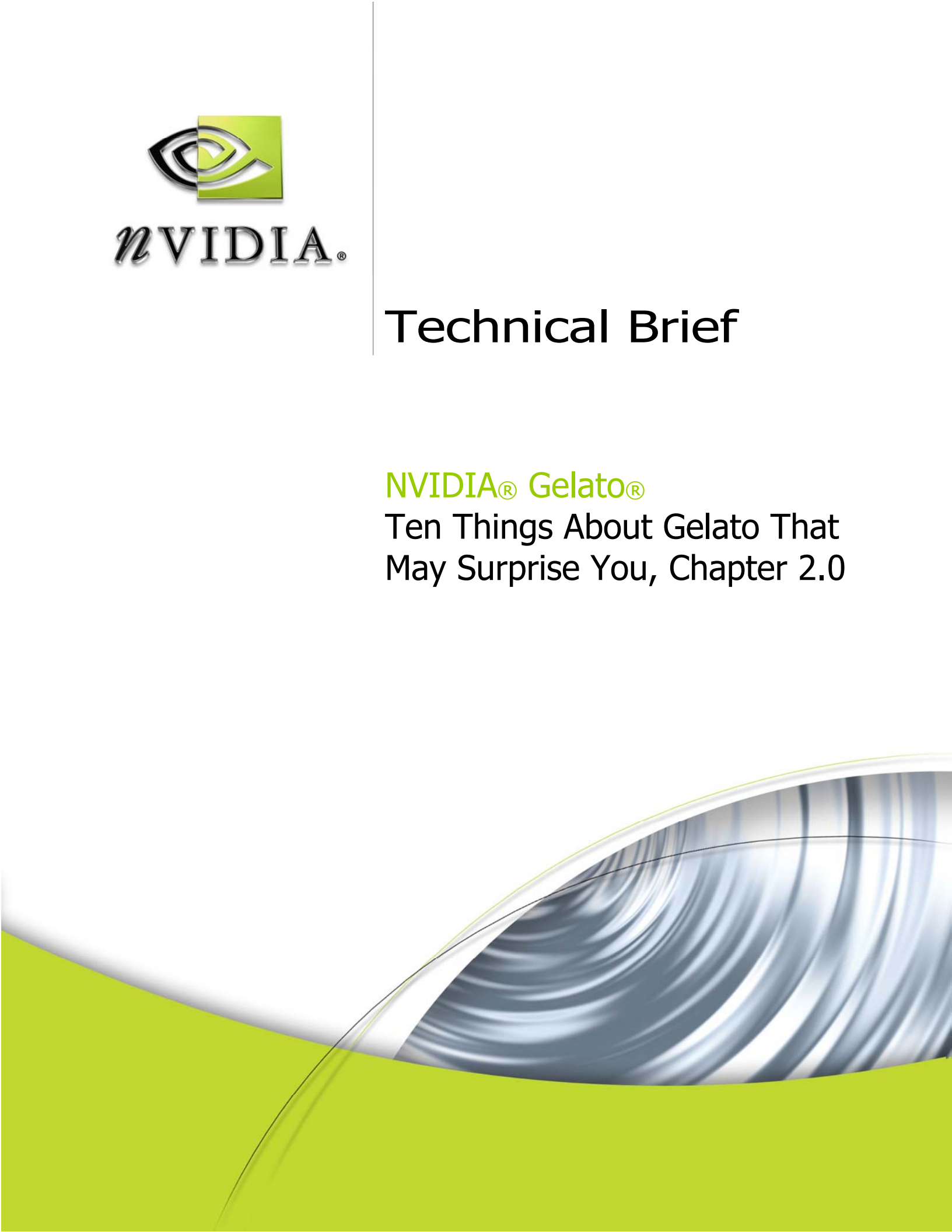

Technical Brief

NVIDIA® Gelato®

Ten Things About Gelato That May Surprise You, Chapter 2.0

Table of Contents

Overview

The NVIDIA® Gelato® rendering software is a complex product that offers exciting features that are buried deep. Although Gelato may look similar to other renderers, it has several functions that are very different from what you are used to, and are far superior.

A previous document, *Ten Things About Gelato That May Surprise You*, listed several great things about release 1.x of the Gelato rendering software that may not be apparent from a cursory glance at the marketing materials or even the documentation. This white paper documents another ten things new to Gelato 2.x that deserve notice.

Two Flavors: Basic Gelato and Gelato Pro

Beginning in May, 2006, NVIDIA is pleased to offer Gelato in two flavors: basic Gelato, and Gelato Pro.

Basic Gelato is *free of charge* but nonetheless is a fully-functional film quality renderer—its quality, speed, features, and performance are not crippled, and we think it's capable of creating production imagery rivaling or exceeding that of other top-of-the-line commercial renderers. Basic Gelato comes with either the Mango plugin for Autodesk Maya, or the Amaretto plugin for Autodesk 3dsMax. You may run basic Gelato using either GeForce or Quadro cards. Support from NVIDIA for basic Gelato is limited, and it lacks certain features that are related to "scalability" to studio environments, but you are nonetheless free to use it for any purpose, including commercial work. Basic Gelato is ideal for students, teachers, small startup studios and emerging markets. Our goal is to make high-quality rendering available to everyone. It's also possible (with separate permission from NVIDIA) to embed basic Gelato in your 3rd party application, to let your customers have access to a great high-end renderer.

Gelato Pro is a commercial package available with a license fee (US$1500 suggested retail price, at the time of this writing). Gelato Pro contains all the features of basic Gelato, is *fully supported* by NVIDIA (though we can only certify it on NVIDIA Quadro cards), and has several additional features that are helpful in a professional production environment, including:

- Sorbetto™ interactive re-rendering
- Network-parallel rendering
- Multithreading
- Native 64-bit support

- DSO shadeops
- Advanced access to new features and beta tests

Over time, it is likely that certain additional scalability features may be added to Gelato Pro only, and also that some features that are currently Pro-only will eventually become part of basic Gelato in future releases. But we will never take an existing basic Gelato feature and turn it into a Pro-only feature.

Sorbetto™ Rerendering

Gelato Pro 2.0 comes with *Sorbetto* rerendering technology, a mode in which the scene geometry for a frame is retained, and the frame can be rerendered multiple times with different attributes (in particular, different lighting values). Basic Gelato also lets you try Sorbetto, although only seven times without having to start over—just enough to see what it does.

This makes it possible to create lighting tools—you can add lights, delete lights, change light linking (which lights shine on which objects), move lights (including automatic recomputation of shadows, whether they are maps or ray-traced), and change any lighting parameter. With only changes to lights, the image can be re-rendered in a fraction of the time it would take for a full render (typically 5-10x faster), which can radically improve a lighting artist's workflow.

Sorbetto can rerender any scene that Gelato can render, without any limitations on scene content, nor any requirements for special preparation of the scene or shaders. And a Sorbetto rendering is not a "preview" -- it is pixel-for-pixel identical to an offline Gelato render of the frame, including antialiasing, motion blur, transparency, etc.

Sorbetto functionality is available through the usual Gelato C++ or Python API's, so you can write your own lighting tool if you want (look for details in the *Gelato Technical Reference*). Of course, our Mango plug-in for Autodesk Maya already contains plenty of Sorbetto controls so you can do your relighting from within Maya (see the *Mango User Manual*). Sorbetto support for Amaretto is planned for a future release.

Volume Shadow Maps

Gelato 2.0 adds a new shadowing technique that allows high-quality shadowing of hair, thin geometry, partially-transparent objects, and volumetric effects. Volume shadow maps take longer to generate than ordinary or Woo shadow maps of the same resolution and use much more disk storage for the maps themselves. But they can capture shadows of transparent as well as opaque objects, including many layers of transparency, and are ideal for shadowing hair and fur.

Volume shadow maps address the shortcomings of ordinary shadow maps by

storing not a single depth value per pixel, but rather a series of depth values representing the depths at which the accumulated opacity crosses various thresholds. You can choose how many of these values to store at the time that you create the map.

API geek details: volume z files are generated when the `Output` call uses `"volz"` as the data parameter and an optional parameter `"int zchannels"` giving the number of z values to store per pixel:

```
Output ("shadow.vz", "tiff", "volz", "camera", "int zchannels", 3)
```

Multiple samples per pixel (i.e. `"spatialquality"`) may be used, and are encouraged for hair or other fine geometry (in contrast to ordinary z files, which can store only one z value per pixel).

A "flat" volume z file can be converted to a MIP-map volume z shadow using the `maketx` utility:

```
maketx -volshad shadow.vz -o shadow.vsm
```

Then, volume z shadows are looked up using the ordinary `shadow()` calls inside shaders. In fact, a shader does not need to know whether it's being handed a volume shadow or an ordinary shadow map, and therefore, volume shadow maps may be used with your existing shaders without modification.

Stereo and Multi-camera Rendering

Gelato can render stereo views—left and right eyes—in a single pass, in only slightly more time (and using almost no additional memory) than a single frame. In our tests, it takes about 20-30% longer to render both left and right eyes than to render a single frame. Two simple stereo modes are particularly simple: either a "toe-in" setup with convergence of the camera views at a set distance, or else "parallel" cameras (as is typically done for stereo IMAX).

API geek details: here's a Pyg example of a camera that will render a stereo pair, automatically producing two output images (one for each eye), using just a single additional Parameter to turn it into a stereo camera:

```
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Parameter ("float stereo:separation", 2.5)
Camera ("main")
```

The `iv` image viewer utility has a special mode for viewing left and right eyes as anaglyph images (viewable with red/cyan glasses) for quick spot-checking of your stereo images, and you can even view the stereo images as they are in the process of rendering.

Actually, stereo rendering is just a special case of Gelato 2.0's general ability to render multiple cameras simultaneously, as long as a single camera (the first one declared) controls all the shading and tessellation. Another important use of multiple cameras is to generate several shadow maps simultaneously, in one pass (rather than a separate pass for each shadow map). Please see the *Technical Reference* for more details.

Dynamic Shadow Maps

Dynamic shadow maps are generated on demand and in memory, and only create those parts of the shadow image that are needed. This can greatly reduce the "time to first pixel," eliminate the need for a completely separate shadow pass before beauty pass rendering can begin, and eliminate the need to actually write out a disk file for the shadow map. In other words, separate shadow maps, and separate render passes to generate those maps, are no longer required in order to use shadow mapping.

To use dynamic shadows, you just make sure that the beauty pass input has the `Camera` and `Output` statements that would have been in the shadow pass file. That is, (1) the beauty pass input, in addition to the main beauty camera, has a `Camera` corresponding to the shadow camera; (2) the beauty pass has an `Output` statement corresponding to the shadow map (and referencing the shadow camera) that declares itself as dynamic using `Parameter ("int dynamic", 1)`; (3) objects that are to appear in the shadow map are put in the geometry set corresponding to the shadow camera (i.e., the geom set has the same name as the camera); (4) light shaders reference the shadow map by its file name, just as they would for a disk shadow map (even though the file will not actually be written to disk).

Image Viewer

Gelato comes with an image viewer, iv, that is packed with interesting features. Here are just a few things about it that you may find better than other image tools:

- While Gelato is rendering to an iv window, you can sweep out a rectangle to tell Gelato which area of the image you want to render next (hold down Shift and the Left mouse button and drag out a rectangle). No more waiting for the render to get to the part of the image you care about! You can even do this several times within a lengthy render, to keep re-focusing attention on areas you care about.
- Wipe between the current image and the last one viewed by holding down Ctrl and the Left mouse button, and drag the dividing line between the images.
- View any image format for which you have an imageio plug-in. Gelato ships with plug-ins that read TIFF, JPEG, OpenEXR, HDR, IFF, PPM, DDS, BMP, PNG, TGA, and SGI rgb formats, as well as all the formats Gelato uses for

textures (shadow maps, volume shadows, MIPmaps, environment cube maps, etc.). It's easy to write your own imageio plug-ins for additional formats.

- View 8-bit, 16-bit integer, 16- or 32-bit float images, scanline or tiled.
- Zoom in and out, pan across huge images, view multiple images (and multiple MIPmap levels within an image), view individual channels (R, G, B, A),
- Apply RGB lookup tables or gamma correction to calibrate your display. Separate default gamma values may be set for each image format type (for example, you may default to no correction for JPG files, but correct TIFF files to a gamma of 2.2).
- Interactively adjust the intensity scaling of images by f-stop, allowing easy viewing of HDR images.
- View stereo pairs as red/blue anaglyphs.
- Load a foreground and/or background image to be composited with other images that you view.
- View close-up pixel values, false color, difference two images, and toggle or wipe between two images.
- Cycle through images (animate) forward or backward, at a given frame rate (or as close as possible to it).

Shader Metadata

It is now possible to embed annotations in your shaders that describe the shader and its parameters. The metadata are embedded in the .gso file and may be read by `libgsoargs` and `gsoinfo`. This is intended primarily to allow shaders to give UI hints to applications that allow users to adjust shader parameters.

Metadata may be attached to the shader as a whole, or to individual parameters of the shader. The metadata is completely up to you—arbitrary names, types, and values may be used, though we propose a few conventions that you may wish to follow, as detailed in the *Technical Reference*, Chapter 8.

For illustration, below is a simple shader with metadata. As you can see, each parameter is given a longer description, and in some cases a variety of hints about what kind of UI widget should be used to control the parameter in a modeling application. (Hint: metadata are described inside the double brackets `[[...]]`)

```
Displacement
lumpy
  [[ string description = "Lumpy displacement, procedural" ]]
(
  float Km = 1
      [[  string description = "Amplitude" ]],
  float sharp = 1
      [[  string description = "Sharpness",
          float UImin = 0 ]],
  float octaves = 3
      [[  string description = "Octaves of noise",
          string UItype = "int", float UImin = 1, float UIsoftmax = 8 ]],
  float freq = 1
      [[  string description = "Frequency" ]],
```

```
  point Pref = P
      [[  string description = "Reference pose (defaults to P)" ]],
  string shadingspace = "shader"
      [[  string description = "Shading space",
          string UItype = "coordsys" ]],
  float shadingfreq = 1
      [[  string description = "Shading frequency" ]]
)
{
  point Pshad = freq * shadingfreq * transform (shadingspace, Pref);
  float dPshad = filterwidth(Pshad);
  float amp = fBm (Pshad, dPshad, octaves, 2, 0.5);
  amp = pow (amp, sharp);
  displace (shadingspace, -Km*amp);
}
```

Have you ever been rendering a big crowd scene in 2.35:1 aspect ratio, only to find that your renderer comes to a screeching halt when it gets to the scanlines that intersect all 10,000 of your characters?

Perhaps you have tricks where you turn the image on its side by modifying the input files to alter the camera matrix, render, and then rotate the image when finished?

That's unnecessary in Gelato. Just put the following in your file:

```
Attribute ("string bucketorder", "vertical")
```

Units

It's easy to write shaders that have functionality or feature sizes that are tied to real-world units, and that automatically convert scene units to real-world units. This allows you to easily write physically correct shaders, or shaders that behave consistently regardless of the units you use to model your scene.

Scene units may be set using two new scene-wide attributes:

```
Attribute ("string units:length", "cm")
Attribute ("float units:lengthscale", 0.5)
```

In this case, it is declared that one unit of "common" space (scene modeling units) corresponds to a real-world distance of 0.5 cm. Gelato understands many units of length, including mm, cm, m, km, in (inches), ft (feet), mi (miles). You can also declare units of time, which is handy for converting between frame numbers and seconds, for example.

A new function, `transformu()`, has been added to the Gelato Shading Language, that converts units. A few examples are below:

```
transformu ("common", "m", 1.0)  // return the number of meters in one
  scene unit
transformu ("cm", "common", amp) // convert amp from cm to scene units
transformu ("in", "cm", 5)       // return the cm equivalent of 5 inches
transformu ("mm", "shader", d)  // return the "shader" space equivalent of
  d mm
transformu ("s", time)          // return the value of the 'time'
  variable, in seconds
```

Below is an example shader that desaturates color based on distance from the camera, and allows the user to specify in which units it operates (also note the clever use of shader metadata):

```
Volume
desatdist
    [[ string description = "Desaturate the appearance of distant objects"
  ]]
(
    float extinctiondist = 100
        [[ string description = "Distance at which saturation decreases by
  1/e",
           float UImin = 0 ]],
    float mindist = 0
        [[ string description = "Minimum distance at which desaturation
  starts",
           float UImin = 0 ]],
    float maxdist = 1e6
        [[ string description = "Maximum distance at which desaturation
  ends",
           float UImin = 0 ]],
      string units = "common"
        [[ string description = "Units used for distances",
           string UItype = "enum:common,km,m,cm,mm,mi,ft,in"
        ]]
    )
{
    float Ilen = transformu (units, length(I));
    Ilen = clamp (Ilen, mindist, maxdist);
    float d = (Ilen - mindist) / extinctiondist;
    C = mix (color(luminance(C)), C, exp(-d));
}
```

Trace Loops

A new syntactic structure has been added to GSL: a `trace` loop, which is rather analogous to `lights` loops, except that `trace` spawns a number of rays and then loops over them, allowing user code to be executed for every ray. Within the trace loop, a variety of messages may be retrieved from the rays, such as the ray's color, opacity, distance to hit, and information about the closest object hit, such as its shaded or unshaded color, normal, uv coordinates, or any named parameter to the shaders on the hit object.

Here is an example:

```
float hits = 0;
  normal Nhit = 0;
  trace ("reflection", P, R, angle, samples) {
      float hitdist;
      getmessage ("trace", "hitdist", hitdist);
      if (hitdist < 1e38) {  // actually hit something
          normal n;
          getmessage ("trace", "N", n);
          Nhit += n;
          hits++;
      }
  }
  if (hits)
      Nhit /= hits;
```

The example above fires `samples` rays from `P` in the direction of `R`, with a spread `angle`. For each of the rays that hit anything, it retrieves the normal of the hit object at the intersection point, and eventually sets `Nhit` to the average normal of the things hit.

Detailed Shader Library Documentation

Extensive shader library documentation now comes with Gelato. Each of the example shaders that ships with Gelato is fully documented, with detailed descriptions of every parameter and images showing how the resulting images change as each parameter takes on different values. Take a look, you'll like it.

Conclusion

Gelato continues to improve substantially with every new release. Sometimes it's hard to explain just how many new and improved features have been packed into the software. While this whitepaper cannot hope to fully document all the new features in Gelato 2.0, we hope that this list will at least inspire you to explore some of these features and to look in more detail at the full documentation for the ones that interest you.

Enjoy, and stay tuned for the next release.

Notice

Trademarks

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com