OpenGL SDK Guide

February 2008

Document Change History

Version	Date	Responsible	Reason for Change
0.9	2/21/2007	Ehart	Beta release
1.1	1/31/2008	Ehart	Feb 2008 release

Overview

The samples in the OpenGL SDK serve two purposes. First, they demonstrate how to use the OpenGL API and its extensions to access the newest hardware features. Secondly, the SDK samples strive to show interesting and unique techniques that can be integrated into an application. This overview guide focuses primarily on the former issue, describing the samples intended to teach API usage. Additionally, it catalogues the API features used by more complicated effects, and it points to any additional documentation explaining those rendering effects.

Updated versions of the SDK will be available on NVIDIA's developer website at:

http://developer.nvidia.com

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

SDK Structure

The SDK is broken into four sub-components represented in the directory hierarchy of the SDK. The first component is the common code directory. This directory contains components reused throughout the SDK. These include model loading, image loading, math, and user interaction code. The next component in the SDK is the media directory. The media directory contains shared data files, like models and textures. The next component is the external code directory. This directory contains widely available libraries which the SDK uses to support common tasks like window management and file access. The final component in the SDK is the sample code. All samples can be built from a single solution file, release.sln.

Common Code

The common code for the SDK is intended to encompass operations that most SDK samples require. The intent is to provide a common set of tools so that anyone familiar with a single sample will be able to easily understand a different sample. The majority of the common code been placed in the "nv" namespace to prevent collisions and make reuse simpler, but some minor pieces from older SDKs have been brought forward and are not contained in the standard namespace.

nvImage

The nvImage library is a tool for loading images for textures. It supports png, dds, and hdr image formats. This provides it with a variety of capabilities, including compressed textures, cubemaps, pre-generated mipmaps, volume textures, and a wide array of bit depths. The interface returns pointers to data formatted for direct use by glTexImage calls. nvImage is built as a shared library, so a single instance can be shared by all samples.

nvModel

The nvModel library is a tool for loading meshes. It presently only support polygonal objects in the obj format. The library supports reading normals and texture coordinates from the file, but it also supports the automatic generation of normals and tangent space vectors. The library will compile a mesh into a format acceptable for use with vertex arrays. Additionally, nvModel will produce index lists supporting point, edge, triangle, and triangle adjacency versions of the model. As with nvImage, nvModel is built as a shared library.

nvMath

nvMath is a collection of header files designed to support the common math needs of 3D applications. It is broken into nvMath.h, nvVector.h (tuple support), nvMatrix.h (matrix support, and nvQuaternion.h (quaternion support). These components provide templated vector, matrix, and quaternion math classes and the

utility functions to operate on them. The classes and functions are all designed to match the naming and behavior of GLSL and Cg as closely as possible to make the sample code and shader code behave consistently.

nvWidgets

nvWidgets is an immediate mode user interface library. The immediate mode style of the library reduces the setup and interaction code within the sample, making it easier to follow the key details in the sample. nvWidgets is not intended to be a complex GUI toolkit, instead it is intended to only provide enough support to make samples more easily comprehensible.

GLEW

GLEW is the OpenGL Extensions Wrangler. This code it not created by NVIDIA, but the version shipped with the SDK occasionally has modifications to support the latest NVIDIA extensions. The license and copyright for GLEW are included in the header files for the library. The version shipped with this SDK is 1.5, plus a minor fix related to the glProgramVertexLimitNV entrypoint..

GLEW provides a mechanism for querying and initializing OpenGL extensions, and core versions newer than are supported in the platform's ABI definition. GLEW parses extension specifications to produce the source for the extension library. The SDK only contains the generated source, and not the generation scripts. To obtain the generation scripts, or for more information about GLEW, please visit the homepage at: http://glew.sourceforge.net/.

External Libraries

The external libraries contained in the external directory are not developed by NVIDIA. The copyright and license information is contained in the headers for all external libraries. They are common libraries used by many other projects, and they may be safely replaced with alternate versions of the ones provided with the SDK. The headers, libs, and dlls are the versions the SDK has been tested against, and they are provided for convenience. The external libraries provided with the SDK are:

- Libpng – PNG loading library
  - Libpng home: http://www.libpng.org/pub/png/libpng.html
- Zlib – compression library used by libpng
  - Zlib home: http://www.zlib.net/
- GLUT – OpenGL Utility Toolkit for managing OpenGL windows

For zlib and libpng, the compiled binaries provided are from the GnuWin32 project at http://gnuwin32.sourceforge.net/.

Media Files

Some of the texture files for the SDK originate from external sources. The following list identifies them and provides links to the original source.

- Paul Debevec's Light probe images - http://www.debevec.org/Probes/
  - rnl_cross.dds
  - grace_new_cross.hdr
- SpeedTree - http://www.speedtree.com/
  - FraserFirNeedles_MD_1.dds
  - FraserFirNeedles_MD_1_Normal.dds

Cg 2.0

The full SDK comes with an installer for Cg 2.0. This version supports the new features of the GeForce 8000 series GPUs. This version has been qualified to work with the samples in this SDK.

Running the Samples

Most samples in the SDK require GeForce 8000 series or later hardware. In many cases they can be run in software emulation on older hardware if necessary. This support can be enabled by the NVemulate tool located at:

http://developer.nvidia.com/object/nvemulate.html

Samples

Cascaded Shadow Maps

This sample demonstrates the use of cascading shadow maps to more gain a better distribution of shadow texels through the scene. The included whitepaper provides an overview of the algorithm for computing the shadow splits and utilizing them in a shader to render the scene. In addition to the cascading shadow maps, this sample demonstrates a few different filtering methods to show how they interact with different shadow map configurations.

Features used

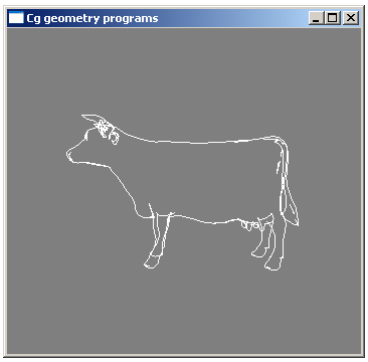
- GL_EXT_gpu_shader4
- GL_EXT_texture_array
- GL_EXT_framebuffer_object
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Move viewer forward
A	Move viewer left
S	Move viewer right
D	Move viewer back
Space	Move the viewer up
1	Use only one shadow buffer
2	Use two shadow buffers

3	Use three shadow buffers
4	Use four shadow buffers
+	Increase the split weight
-	Decrease the split weight

Cg Geometry Program



This sample demonstrates how to use geometry programs with Cg. Specifically, it demonstrates compiling the programs to the GL_NV_gpu_program4 profile. Included in the shader programs with this sample are: Bezier curve tessellation, b-spline curve tessellation, silhouette determination, and fin generation.

Features used

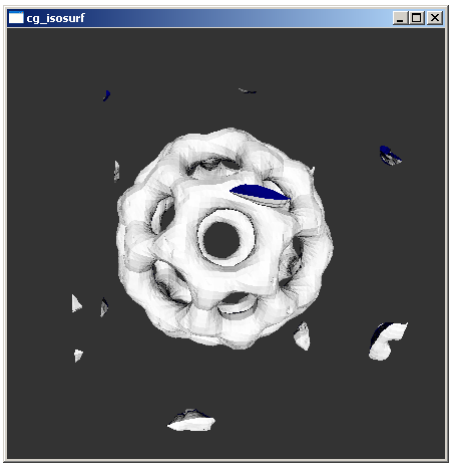
- GL_NV_gpu_program4
  - GL_NV_geometry_program4 (part of GL_NV_gpu_program4)
- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_NV_texture_rectangle
- GL_ARB_texture_float
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe
M	Toggle between rendering the model and rendering simple

	geometry
P	Toggle drawing with a geometry program
O	Toggle drawing the original geometry
Space	Toggle continuous animation of the object
+	Increase the number of subdivisions used for curve tessellation
-	Decrease the number of subdivisions used for curve tessellation

Cg Isosurf



This sample demonstrates how to perform iso-surface extraction from three different field sources. The iso-surface is produced through an algorithm called marching tetrahedra, using the geometry shader. The geometry is recreated dynamically each frame.

Marching Tetrahedra Algorithm

Background

Iso-surface extraction is a way to create a polygonal representation of a continuous field, such as those produced by mathematical description such as implicit surfaces or direct measurement such as medical scan. An iso-surface attempts to display the surface in the volume for which all points are at a single value, the iso-value. Changing the iso-value will produce different surfaces

Marching tetrahedra produces an iso-surface by filling the space of the field with tightly packed tetrahedral. At each vertex of the tetrahedra the value of the field at that point is compared to the iso-value to produce a Boolean value. If the Boolean is different for the vertices of a single tetrahedron, then the iso-surface must have passed through the area represented by that tetrahedron. Assuming that the iso-surface is planar within the tetrahedron, the intersection will form either a triangle or a quadrilateral.

GPU Implementaion

Iso-surface extraction is implemented on the GPU using a combination of the vertex shader and the geometry shader. The base geometry is a fine tetrahedralization of the volume, where the four vertices of the each tetrahedron are submitted as a line with adjacency. The vertex shader samples the field, compares it to the iso-value, and transforms the vertex. The geometry shader performs a logical or of the results from the iso-value compares. This four bit integer determines the geometry produced by the geometry shader. If all the bits are set, a value of fifteen, or all the bits are not set, a value of zero, then the tetrahedron does not intersect the iso-surface, and no geometry is generated. Otherwise, the 4-bit value indexes into a texture that contains a table of values describing the geometry produced. These eight values pulled from the table describe the three or four edges which intersect the iso-value. The values correspond to the indices of the vertices that bound the edge. The fourth edge is special, in that it is encoded such that its first value, the seventh overall, will always be non-zero if the intersection is a quadrilateral. This allows it to be used as a condition for whether to output a fourth vertex. To produce the output primitive, the shader solves a linear equation on the three or four edges to interpolate the vertex position to the point where the field would intersect the edge. These values are output as the vertices for the generated primitives

Swizzling

In addition to implementing the straight iso-surface extraction, this sample also demonstrates a technique for optimizing volume traversals. Instead of rendering a set of tetrahedral that march through the grid in row order, it reorders them to improve locality of reference. This works by a bit swizzling technique. In this sample, the volume is traversed as a set of 8 miniature volumes (2 x 2 x 2 configuration) each containing six tetrahedral. These blocks are then stepped in a regular row-ordered traversal. The improvement in cache behavior results in up to a thirty percent increase in performance. Other traversal orders are possible, but the one used in this sample provides a good compromise between complexity and performance, as it is within five percent of the best measured performance.

Features used

- GL_NV_gpu_program4
  - GL_NV_geometry_program4 (part of GL_NV_gpu_program4)
- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_ARB_texture_rectangle
- GL_ARB_texture_float
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu

Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe
V	Toggle using a vertex program to render the geometry
F	Toggle using a fragment program to render the geometry
G	Toggle using a geometry program to render the geometry
P	Toggle drawing geometry to just drawing points
L	Toggle drawing the geometry for tetrahedrons to layers
Space	Toggle continuous animation of the object
+	Increase the isovalue
-	Decrease the isovalue
]	Increase the number of primitives
[	Decrease the number of primitives
1	Switch to the metaballs program
2	Switch to the volume texture program
3	Switch to the procedural program

Compress Normal DXT

Compress NormalDXT demonstrates the online compression of normal maps into DXT format textures with a shader on the GPU. Importantly, it uses a separate alpha channel or a two-component compressed format to improve the quality of the normal map over the standard RGB compression scheme. The sample also demonstrates the use of integer operations within the pixel shader. Once the compression has been done in the pixel shader, the compressed block is written out to an RGBA unsigned 16-bit integer texture. A pixel buffer object is then used to transfer the image data to the compressed texture object, avoiding a host readback while converting between the different surface formats.

GPU Texture Compression Algorithm

The texture compression algorithms used in this sample are detailed in a whitepaper published at: http://developer.nvidia.com/object/real-time-normalmap-dxtcompression.html

Features used

- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_EXT_framebuffer_object
- GL_NV_gpu_program4
- GL_ARB_pixel_buffer_object
- GL_ARB_texture_compression
- GL_EXT_texture_compression_s3tc
- GL_EXT_texture_integer
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
+	Zoom into the texture
-	Zoom out from the texture
B	Benchmark the compress operation
R	Reset the texture position
Left Arrow	Scroll the texture left
Right Arrow	Scroll the texture right
Up Arrow	Scroll the texture up
Down Arrow	Scroll the texture down

Compress YCoCg DXT

Compress YCoCgDXT demonstrates the online compression of DXT format textures with a shader on the GPU. It provides a mode that uses an alternate color space from the standard RGB one to provide better compression results. The sample also demonstrates the use of integer operations within the pixel shader. Once the compression has been done in the pixel shader, the compressed block is written out to an RGBA unsigned 16-bit integer texture. A pixel buffer object is then used to transfer the image data to the compressed texture object, avoiding a host readback while converting between the different surface formats.

GPU Texture Compression Algorithm

The texture compression algorithms used in this sample are detailed in a whitepaper published at: http://developer.nvidia.com/object/real-time-ycocg-dxt-compression.html.

Features used

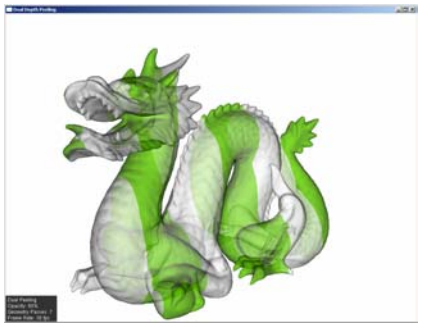
- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_EXT_framebuffer_object
- GL_NV_gpu_program4
- GL_ARB_pixel_buffer_object
- GL_ARB_texture_compression
- GL_EXT_texture_compression_s3tc
- GL_EXT_texture_integer
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera

Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
+	Zoom into the texture
-	Zoom out from the texture
B	Benchmark the compress operation
R	Reset the texture position
Left Arrow	Scroll the texture left
Right Arrow	Scroll the texture right
Up Arrow	Scroll the texture up
Down Arrow	Scroll the texture down

Dual Depth Peeling



Dual Depth Peeling demonstrates how the new capabilities in the GeForce 8000series can be utilized to cut the number of depth peeling asses in half when attempting to render high quality transparency. The sample also provides an implementation of regular depth peeling and some alternate approximations for transparency for the purposes of comparison. This sample includes a whitepaper fully detailing the algorithms used.

Features used

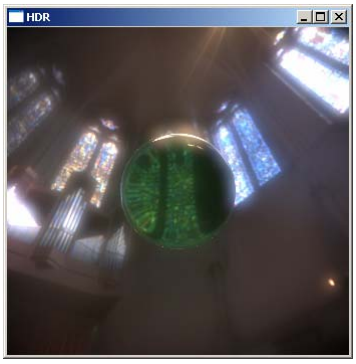
- GL_ARB_texture_rectangle
- GL_ARB_texture_float
- GL_NV_float_buffer
- GL_NV_depth_buffer_float
- OpenGL version 2.0

Controls

Control	**Action**
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera

Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
A	Decrease the model opacity
D	Increase the model opacity
1	Dual depth peeling mode
2	Regular depth peeling mode
3	Weighted average mode
4	Weighted sum mode
R	Reload shaders
B	Change background color

HDR



The HDR sample demonstrates the use new GeForce 8000 series features in the rendering and display of a high-dynamic range scene. The sample uses the GL_EXT_packed_float and GL_EXT_texture_shared_exponent extensions to optimize the storage of floating point framebuffers and textures. The sample also demonstrates the use of GL_EXT_framebuffer_multisample with these and other high dynamic range formats. Finally, in addition to the bloom and exposure operations frequently used in HDR rendering, this sample implements an effect that approximates streaming rays one would often see emanating from bright objects.

Features used

- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_EXT_framebuffer_object
- GL_EXT_framebuffer_multisample
- GL_EXT_framebuffer_blit
- GL_NV_framebuffer_multisample_coverage
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe
A	Toggle MSAA 4x <-> CSAA or supersampling
F	Toggle using a fragment program
C	CSAA 16x only (FBO size factor = 1 and no kernel filter used)
S	CSAA 16x with supersampling (FBO size factor = 2x + kernel filter)
-	Cycle backward through the HW MSAA/CSAA modes for supersampled FBO
+	Cycle forward through the HW MSAA/CSAA modes for supersampled FBO
9	Decrease the supersample size factor
0	Increase the supersample size factor
1	Select single bilinear downsample technique
2	Select five bilinear tap downasmple technique
3	Select advanced downsample technique
Space	Toggle continuous animation of the object

Multisample Coverage

Multisample_coverage demonstrates the use of the WGL_NV_multisample_coverage extension. This extension allows an application to request Coverage Sample Anti-Aliasing (CSAA) for the pixel format in a window. CSAA allows a higher number of coverage samples to be combined with color and depth samples to generate a higher quality anti-aliased image with less cost.

Features used

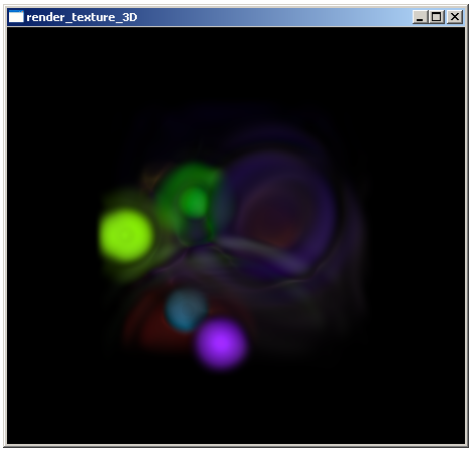
- WGL_NV_multisample_coverage

- OpenGL version 2.0

Controls

Control	Action
Escape	Quit the sample

Render to 3D Texture



Render to 3D Texture demonstrates the use of GL_EXT_framebuffer_object to render directly into slices of a three dimensional texture. In this case, a simple wave simulation is rendered into the texture. The texture is visualized by a shader marching rays through the texture.

Features used

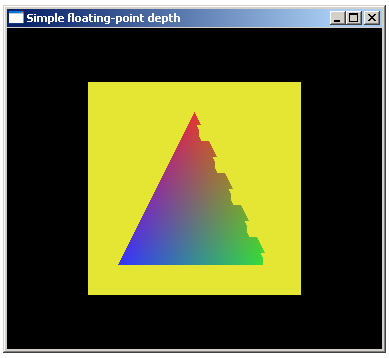
- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_EXT_framebuffer_object
- GL_NV_gpu_program4
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera

Escape	Quit the sample
A	Toggle automatically seeding the volume with splats
R	Reset the simulation
C	Draw the bounding cube in wireframe
X	Add a splat to the volume
Enter	Advance the simulation one step
Space	Toggle continuous update of the simulation

Simple Depth Float



The simple_depth_float sample demonstrates how best to utilize the precision of a floating point depth buffer. The sample renders the test scene to an FBO with one of three depth buffer configurations (16 bit integer, 24 bit integer, and 32 bit floating point). The sample also demonstrates the differences between using the typical mapping of depth values into the depth buffer (near plan to 0 and far plane to 1) and the an inverse mapping (near plane to 1 and far plane to 0). Due to the concentration of precision close to zero in a floating point frame buffer, the inverse mapping provides significantly better results when attempting to eliminate depth fighting typically seen with distant objects.

Features used

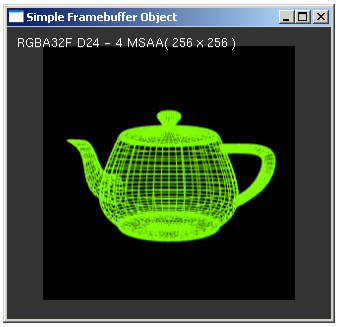
- GL_EXT_framebuffer_object
- GL_NV_depth_float
- OpenGL version 2.0

Controls

Control	**Action**
Right Mouse Button	Application menu
Escape	Quit the sample
W	Toggle rendering in wireframe

D	Toggle displaying the depth buffer
I	Toggle using an inverted depth
M	Toggle using the projection matrix to invert the depth
Space	Toggle continuous animation of the object

Simple Framebuffer Object



The simple_framebuffer_object sample demonstrates how to discover and use formats for framebuffer objects. It covers the creation and use of multisample framebuffer objects, framebuffer objects with floating point formats, and framebuffer objects with sRGB formats.

Features used

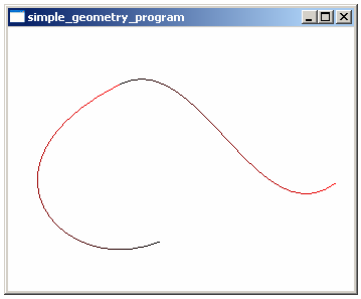
- GL_EXT_framebuffer_object
- GL_NV_depth_float
- GL_EXT_framebuffer_multisample
- GL_EXT_framebuffer_blit
- GL_NV_framebuffer_multisample_ceverage
- GL_ARB_fragment_program
- GL_ARB_texture_float
- GL_EXT_packed_float
- OpenGL version 2.0

Controls

Control	**Action**
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe

D	Toggle displaying the depth buffer
O	Toggle overlaying the rendered texture
+	Increase the size of the FBO
-	Decrease the size of the FBO
Space	Toggle continuous animation of the object

Simple Geometry Program



Simple_geometry_program demonstrates the use of the GL_NV_gpu_program4 extension to control the geometry shader.

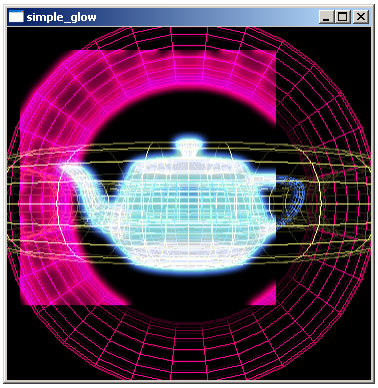
Features used

- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_NV_gpu_program4
  - GL_NV_geometry_program4
- OpenGL version 2.0

Controls

Control	**Action**
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe
P	Toggle the geometry and vertex programs on/off
+	Increase the number of segments for curve tessellation
-	Decrease the number of segments for curve tessellation
Space	Toggle continuous animation of the object

Simple Glow



Simple_glow shows a highlight shader using framebuffer object and Cg. The whitepaper in the sample's doc directory explains the technique in detail.

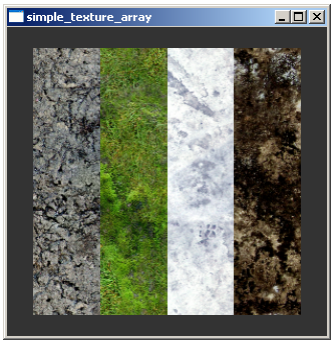
Features used

- GL_EXT_framebuffer_object
- GL_ARB_vertex_program
- GL_ARB_fragment_program
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle wireframe
1	Decrease the filter width
2	Increase the filter width
3	Decrease the size of the FBO
4	Increase the size of the FBO
-	Decrease the blend factor for the glow
+	Increase the blend factor for the glow
Space	Toggle continuous animation

Simple Texture Array



The simple_texture_array sample demonstrates the use of GL_EXT_texture_array to create multilayered array textures. It demonstrates the creation and loading of the texture and the use within a shader.

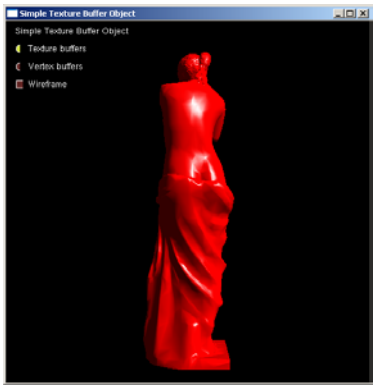
Features used

- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_NV_gpu_program4
  - GL_NV_geometry_program4
- GL_EXT_texture_array
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe
F	Toggle the fragment program on/off
L	Toggle between the lerping and non-lerping fragment programs
Space	Toggle continuous animation of the object

Simple Texture Buffer Object



The simple_texture_buffer_object sample demonstrates the use of GL_EXT_texture_buffer_object to access large blocks of data from within the vertex shader. In this sample, the texture buffer is used to contain irregularly indexed vertex data, with the indexing all handled from within the shader. Note that the performance when fetching from the texture buffer object is significantly less than the performance when using vertex buffer objects to supply the geometry. This is because the model and shader in this sample are not bound by the vertex attribute fetch stage of the pipeline. Instead, they are bound by the shader, so the additional work of fetching the vertices reduces the performance. Under certain scenarios, where the hardware is bottlenecked on vertex attribute fetching, using a texture buffer object to hold some of the attributes may provide a performance advantage.

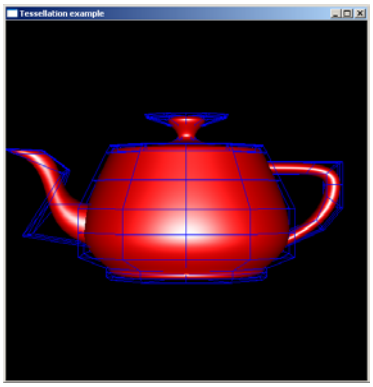
Features used

- GL_EXT_gpu_shader4
- GL_EXT_texture_buffer_object
- OpenGL version 2.0

Controls

Control	Action
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
W	Toggle rendering in wireframe
B	Toggle between rendering with texture buffers and vertex buffers
Space	Toggle continuous animation of the object

Tessellation



This sample demonstrates the tessellation of Bezier patches with a vertex shader. The patch data is stored using GL_EXT_bindable_uniform, and the rendering of the patches utilizes GL_EXT_draw_instanced to render the multiple patches in a single draw call.

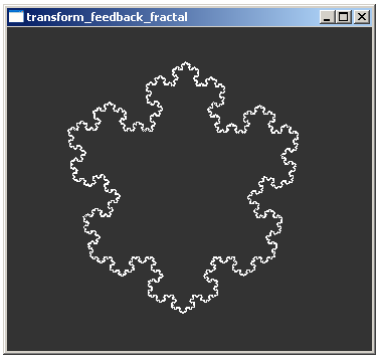
Features used

- GL_EXT_gpu_shader4
- GL_EXT_bindable_uniform
- GL_EXT_draw_instanced
- OpenGL version 2.0

Controls

Control	**Action**
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
w	Toggle displaying wireframe
1	Toggle drawing the control mesh
2	Toggle drawing the GLUT reference teapot

Transform Feedback Fractal



This sample demonstrates how an application can use a feedback loop with the GL_NV_transform_feedback extension to generate large amounts of geometry that cannot be generated by a single pass through the geometry shader.

Features used

- GL_ARB_vertex_program
- GL_ARB_fragment_program
- GL_NV_gpu_program4
  - GL_NV_geometry_program4
- GL_NV_transform_feedback
- GL_ARB_texture_float
- OpenGL version 2.0

Controls

Control	Action
Right Mouse Button	Application menu
Left Mouse Button	Rotate object
Shift + Left Mouse Button	Pan object in front of the camera
Control + Left Mouse Button	Dolly the object toward/away from the camera
Escape	Quit the sample
w	Toggle displaying wireframe
l	Toggle lighting the mesh
c	Toggle continuous tessellation
+	Increase tessellation level
-	Decrease tessellation level
]	Increase random transform scale
[	Decrease random transform scale
r	Reset values
s	Step forward through the subdivision steps (only applicable when continuous subdivision is not enabled)

Space	Toggle continuous animation of the object

Christmas Tree Renderer

This sample demonstrates the use of deferred shading with framebuffer objects to render a Christmas tree. The sample has its own whitepaper describing the techniques and steps in detail.

Features used

- GL_EXT_gpu_shader4
- GL_EXT_geometry_shader4
- GL_EXT_packed_float
- GL_EXT_framebuffer_object
- GL_ARB_texture_float
- OpenGL version 2.1

Controls

Control	Action
Right Mouse Button	Application menu
Escape	Quit the sample
w	Move viewer forward
a	Move viewer left
s	Move viewer back
d	Move viewer right
+	Increase exposure

-	Decrease exposure
[	Decrease sigma
]	Increase sigma
<	Decrease blur amount
>	Increase blur amount
;	Decrease blur buffer size
‘	Increase blur buffer size
t	Toggle the display of the timer graph
v	Toggle display of current HDR parameters
i	Toggle display of intermediate results
b	Toggle drawing the tree branches
f	Toggles drawing the needles
o	Toggle drawing the ornaments
r	Toggle drawing the reflections
l	Toggle drawing the lights
x	Change the current intermediate displayed
Space	Toggle continuous animation of the object