Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

CUDA, Supercomputing for the Masses: Part 18

Using Vertex Buffer Objects with CUDA and OpenGL

May 27, 2010
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/225200412

In CUDA, Supercomputing for the Masses: Part 16 and Part 17 of this article series, I discussed new features in the CUDA Toolkit 3.0 release that can make day-to-day development tasks easier, less error prone, and more consistent. Essentially expanded, consistent coverage appears to have been the thinking behind the release that includes memory checking, runtime and driver interoperability, C++ class and template inheritance as well as Fermi and OpenCL enhancements.

This installment returns to the topic of mixing OpenGL and CUDA C within the same application first introduced in Part 15 of this series. Part 15 demonstrated how to create 2D images with CUDA C on a pixel-by-pixel basis and display them with OpenGL through the use of PBOs (Pixel Buffer Objects). This article will complete that discussion by demonstrating how to use VBO (Vertex Buffer Objects) to create 3D images with CUDA C and render them using OpenGL as 3D collections of points, wire frame images, and surfaces.

The provided examples will demonstrate how to achieve very high rendering and compute performance through the use of primitive restart, an OpenGL extension CUDA programmers can exploit to by-pass PCIe bottlenecks. On a GTX 285, primitive restart can be used to render at 60-90 frames per second faster than other optimized OpenGL routines such as multiDraw. Even highly-experienced OpenGL programmers should find this article and working programming examples to be both new and informative as the OpenGL standards compliant primitive restart capability can deliver high-performance high-quality graphics even when the images require irregular meshes.

Readers should note that care was taken in the design of the software framework to make it as easy as possible to adapt to new applications. With only minor refactoring, the same framework used for the PBO examples in the Part 15 article has been extended to create animated 3D point, wireframe and surface images. Different CUDA kernels can then be used to create different visualization applications ranging from simple to complex. (Similarly it was converted in Part 17 to a C++ class framework that can use difference CUDA kernels and define new textures via inheritance.) Also, the examples in this article (and Part 17) support both the CUDA 3.0 and deprecated pre-3.0 graphics interoperability APIs.

This article first creates an animated sinusoidal surface based on the NVIDIA simplGL.cu CUDA kernel. A single image from this example is shown in Figure 1. Then the Perlin noise generator from the simple PBO article will be slightly modified to create a virtual terrain model that the user can fly around in as well as dynamically alter with keyboard commands. Figure 2 shows a sample surface, Figure 3 a pilot's eye view, Figure 4 a wireframe version and Figure 5 shows an artificial terrain rendered with points.

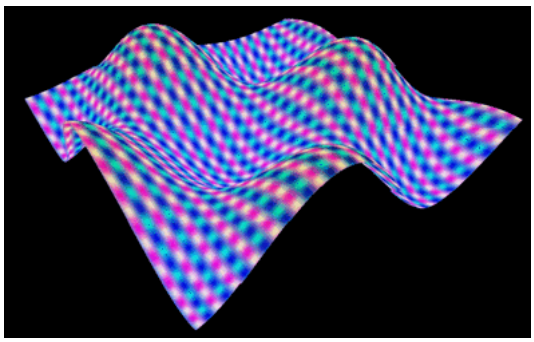

Figure 1: Example of a surface created with the sinusoidal surface VBO kernel.

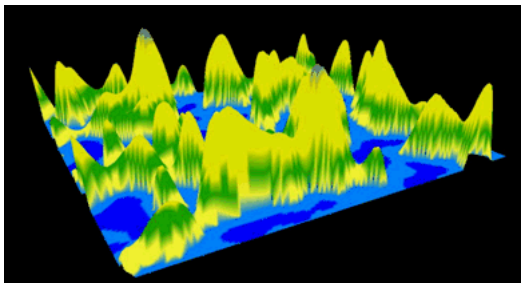

Figure 2: Example of a 3D surface created with the Perlin noise kernel.

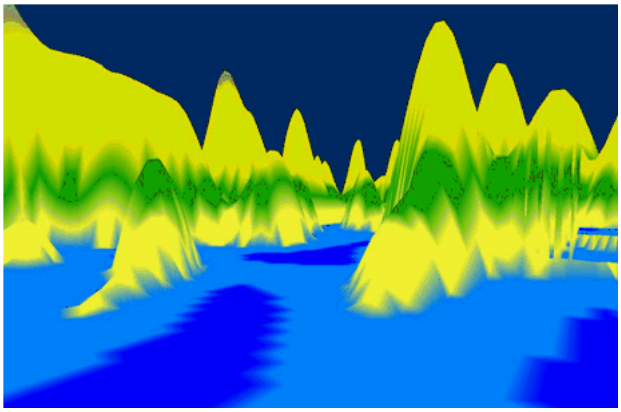

Figure 3: A pilot's eye view.

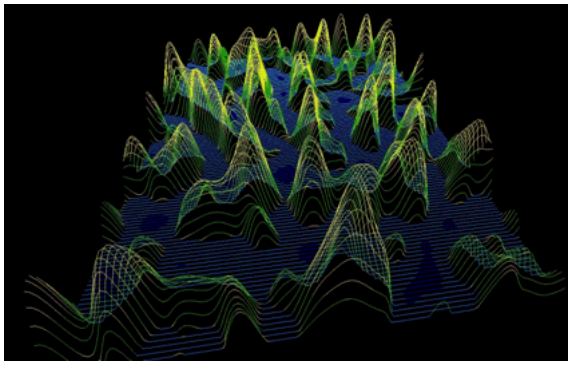

Figure 4: A terrain wireframe.

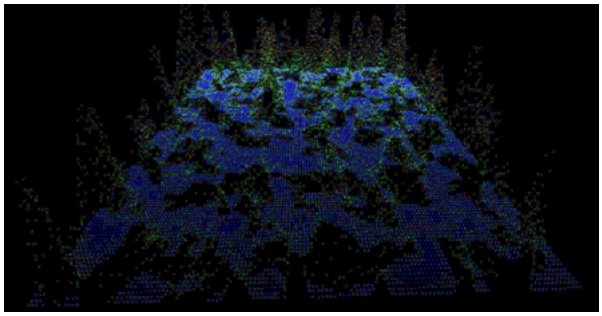

Figure 5: Rendering an artificial terrain with points.

Essentially, creating new, complicated applications can be as simple as compiling with a different CUDA kernel. For clarity, separate 3D vertex and color arrays are used within the source code for both data creation and display. This should help speed understanding and make data visualization as easy as writing a new kernel or loading data from disk to alter the 3D vertex array, color array, or both. Those readers who choose to create their own CUDA kernels should gain a strong practical sense of how easy and flexible visualization can be with a combined CUDA/OpenGL approach.

The examples are known to compile and run on Linux and Windows although this article discusses how to build the codes under Linux. Go to the GPUcomputing.net website for information about building the examples with Microsoft Visual Studio.

Using Primitive Restart for 3D Performance

As mentioned in the introduction, the examples in this article utilize an OpenGL extension called "primitive restart" to minimize communications across the PCIe bus and speed rendering. Simply, primitive restart allows the programmer to specify a data value that is interpreted by the OpenGL state machine as token indicating the current graphics primitive has completed. The next data item is assumed to be at the start of the next graphics primitive. Valid graphics primitives include TRIANGLE_STRIP, TRIANGLE_FAN, and others.

The following illustrates this process. Assume the variable `qIndices` contains the indexes of data points that are to be used in drawing a triangle strip:

```
unsigned int qIndicies[] = { 0, 1, 2, 3, 65535, 2, 3, 4, 5};
```

The call to `glDrawElements` shown below will draw seven triangles. Note: `size` is the number elements in `qIndicies`.

```
    glDrawElements(GL_TRIANGLE_STRIP, size, GL_UNSIGNED_INT, qIndices);
```

The following code snippet calls `glPrimitiveRestartIndexNV` to specify that the value 65535 (passed via `RestartIndex`) is the primitive restart token. The routine

`glEnableClientState` is then called to tell the OpenGL state machine to start using primitive restart:

```
glPrimitiveRestartIndexNV(RestartIndex);
glEnableClientState(GL_PRIMITIVE_RESTART_NV);
```

Now a single call to `glDrawElements` using `qIndicies` will draw four triangles because the value 65535 tells OpenGL to act as if two separate `glDrawElements` calls were made.

```
glDrawElements(GL_TRIANGLE_STRIP, size, GL_UNSIGNED_INT, qIndices);
```

The advantages of the primitive restart approach are many-fold as:

- All control tokens and data for viewing can be generated and kept on the GPU.
- Variable numbers of items can be specified between the primitive restart tokens. This allows irregular grids and surfaces to drawn as arbitrary numbers of line segments, triangle strips, triangle fans, etc, can be specified depending on the drawing mode passed to `glDrawElements`.
- Rendering performance can be optimized by arranging the indices to achieve the highest reuse of the data cache in the texture units.
- Higher quality images can be created by alternating the direction of tessellation as noted in the primitive restart specification and illustrated in Figures 6 and 7.

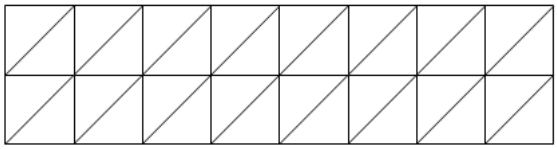

Figure 6: Two strips.

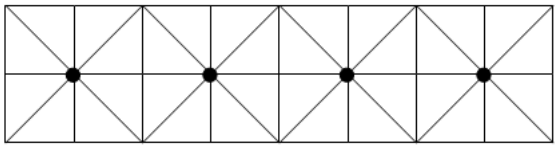

Figure 7: Four fans (center marked with filled circle).

More information on various OpenGL optimizations including multiDraw (an alternative OpenGL method to draw multiple items with one call) can be found here on the OpenGl website. In particular, the primitive restart specification notes that multiDraw "still remain[s] more expensive than one would like".

A rough performance comparison using example code from this article on Linux demonstrates the speed of primitive restart compared to other techniques. Source code for the examples in this article that can utilize different OpenGL rendering techniques (selectable using preprocessor `#define` statements) can be found here on the GPUcomputing.net website. For clarity, the `#ifdef` preprocessor statements were not included in the source code provided in this example. Of course, performance results can vary depending on the machine and GPU combination as well as driver version and settings. In addition, no attempt was made to fully optimize any of the drawing methods; see Table 1.

Method	Observed FPS	Rough Average (FPS)
Simple one by one	470-500	500
MultiDraw	490-510	508
Primitive Restart	550-590	560

Table 1: Approximate performance number on GTX285.

It is important to stress that these frame rates include the time required to re-compute the 3D position and color for every vertex and color in the image. This represents a worst case frame rate scenario that demonstrates the power and speed possible with hybrid CUDA/OpenGL applications. Real applications will undoubtedly deliver much higher performance by recalculating all the data only when necessary.

Vertex Buffer Objects

Most of the PBO software framework described in Part 15 will be reused to demonstrate drawing with VBOs in this article. As with pixel buffer objects, high-speed interoperability is achieved by mapping OpenGL buffers into the CUDA memory space without requiring a memory copy. Detailed information about the API and OpenGL calls used when mixing CUDA with OpenGL can be found in the PBO article as well as Joe Stam's 2009 NVIDIA GTC conference presentation What Every CUDA Programmer Should Know About OpenGL (also available in video here).

Just as in the NVIDIA SDK samples, GLUT (a window system independent OpenGL Toolkit) was utilized for Windows and Linux compatibility. This article focuses on building and running the examples under Linux. Go here for more information about building these examples with Microsoft Visual Studio.

Figure 8 summarizes how the VBO example code interacts with GLUT. The boxes highlighted in green indicate those areas that required refactoring the original PBO code so it can work with VBOs.

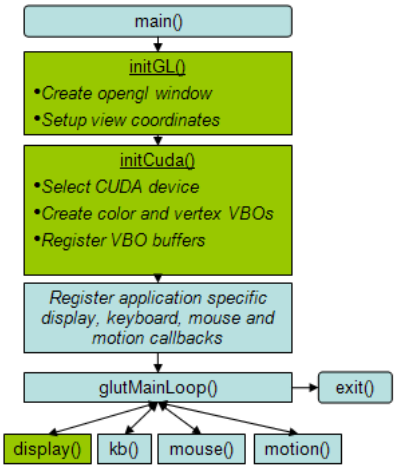

Figure 8: Schematic of GLUT VBO code interactions.

The relationship between the four files used in the procedural framework discussed in this article is illustrated in Figure 9.

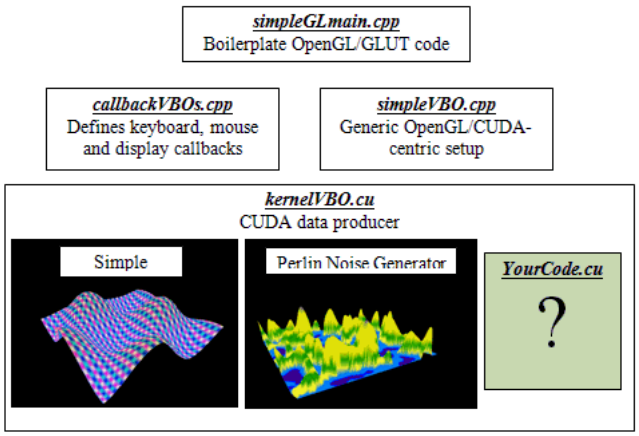

Figure 9: Organization of files and activities.

The simpleGLmain.cpp File

The file simplGLmain.cpp from Part 15 already provides all the boilerplate needed to open a window on the screen and set some basic viewing transforms sufficient to display the two-dimensional pictures created in our PBO examples.

From a coding point of view, adapting the 2D PBO version of simpleGLmain.cpp to display 3D VBO objects merely requires specifying a three-dimensional perspective in `initGL` with the call:

```
gluPerspective(60.0, (GLfloat)window_width/(GLfloat)window_height, 0.1, 1, 0.);
```

From an OpenGL point of view, adding a perspective with `gluPerspective` effectively places a camera in a three-dimensional location from which to view the data generated by our CUDA kernel(s). Three-dimensional rendering occurs on the GPU when the programmer:

- Specifies objects in 3D space using simple triangles, vertices and lines.
- Defines a virtual camera position and viewing angle.

The GPU can then identify and update the display pixels as the data and/or viewing position changes.

Continuing with the camera analogy, the required transforms for 3D data are:

1. Position and point the camera at the scene (a view transformation)
2. Arrange the scene composition (a model transform)
3. Adjust the camera zoom (a projection transform)
4. Choose the final size (a viewport transform)

OpenGL transforms and coordinate systems require very detailed thinking and explanation. Suffice it to say that defining the projection transform is all that was needed to adapt simplePBO.cpp so 3D VBO data can be displayed.

Song Ho Ann has an excellent set of tutorials and visual aids to help understand the details of OpenGL transforms, the OpenGL rendering pipeline (including differences between pixel and geometry rendering), the OpenGL projection matrix, and much more. I recommend these tutorials or one of the many other excellent resources available on the web including the online version of Chapter 3 of the OpenGL Red Book.

The following is the complete source code for the VBO version of simpleGLmain.cpp.

```
// simpleGLmain.cpp (Rob Farber)

/*
   This wrapper demonstrates how to use the Cuda OpenGL bindings to
   dynamically modify data using a Cuda kernel and display it with opengl.

*/

// includes, GL
#include <GL/glew.h>

// includes
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cutil_gl_error.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

// The user must create the following routines:
void initCuda(int argc, char** argv);

// GLUT specific variables
const unsigned int window_width = 512;
const unsigned int window_height = 512;

unsigned int timer = 0; // a timer for FPS calculations

// Forward declaration of GL functionality
CUTBoolean initGL(int argc, char** argv);

// Rendering callbacks
void fpsDisplay(), display();
void keyboard(unsigned char key, int x, int y);
void mouse(int button, int state, int x, int y);
void motion(int x, int y);

// Main program
int main(int argc, char** argv)
{
  // Create the CUTIL timer
  cutilCheckError( cutCreateTimer( &timer));

  if (CUTFalse == initGL(argc, argv)) {
    return CUTFalse;
  }

  initCuda(argc, argv);
  CUT_CHECK_ERROR_GL();

  // register callbacks
  glutDisplayFunc(fpsDisplay);
  glutKeyboardFunc(keyboard);
  glutMouseFunc(mouse);
  glutMotionFunc(motion);

  // start rendering mainloop
  glutMainLoop();

  // clean up
  cudaThreadExit();
  cutilExit(argc, argv);
}

// Simple method to display the Frames Per Second in the window title
void computeFPS()
{
  static int fpsCount=0;
  static int fpsLimit=100;

  fpsCount++;

  if (fpsCount == fpsLimit) {
    char fps[256];
    float ifps = 1.f / (cutGetAverageTimerValue(timer) / 1000.f);
    sprintf(fps, "Cuda GL Interop Wrapper: %3.1f fps ", ifps);

    glutSetWindowTitle(fps);
    fpsCount = 0;

    cutilCheckError(cutResetTimer(timer));
  }
}

void fpsDisplay()
{
  cutilCheckError(cutStartTimer(timer));

  display();

  cutilCheckError(cutStopTimer(timer));
  computeFPS();
}
```

```
float animTime = 0.0;    // time the animation has been running

// Initialize OpenGL window
CUTBoolean initGL(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutInitWindowSize(window_width, window_height);
  glutCreateWindow("Cuda GL Interop Demo (adapted from NVIDIA's simpleGL");
  glutDisplayFunc(fpsDisplay);
  glutKeyboardFunc(keyboard);
  glutMotionFunc(motion);

  // initialize necessary OpenGL extensions
  glewInit();
  if (! glewIsSupported("GL_VERSION_2_0 ")) {
    fprintf(stderr, "ERROR: Support for necessary OpenGL extensions missing.");
    fflush(stderr);
    return CUTFalse;
  }

  // default initialization
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glDisable(GL_DEPTH_TEST);

  // viewport
  glViewport(0, 0, window_width, window_height);

  // set view matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // projection
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, (GLfloat)window_width / (GLfloat) window_height,
         0.10, 10.0);

  return CUTTrue;
}
```

The simpleVBO.cpp File

The Part 15 simplePBO.cpp file was modified in several ways to create the simpleVBO.cpp file shown below. The only significant change is to the display routine to utilize both 3D vertex and color data as discussed below. The remaining changes are straight-forward refactoring to support the CUDA 3.0 and pre-3.0 graphics interoperability APIs as well as two graphics interoperability buffers `colorVBO` and `vertexVBO`. Note that `colorVBO` is created to hold a uchar4 color array and `vertexVBO` a float4 array of vertex values.

The changes to simplePBO.cpp can be summarized as follows:

- The call to `launch_kernel` now includes parameters to pass both vertex and color arrays.
- `createVBO` was refactored to use a `typedef mappedBuffer_t` structure. This allows float4 and uchar4 arrays to be created as well as facilitating CUDA 3.0 and pre-3.0 graphics interoperability APIs.
- `cleanupCuda` frees both `vertexVBO` and `colorVBO` structures.
- `runCuda` maps and unmaps both `vertexVBO` and `colorVBO` objects as well as passing the appropriate pointers to `launch_kernel`.

```
// simpleVBO.cpp (Rob Farber)

// includes, GL
#include <GL/glew.h>
#include <GL/gl.h>
#include <GL/glext.h>

// includes
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

//#define USE_CUDA3

extern float animTime;

////////////////////////////////////////////////////////////////////////////////
// VBO specific code
#include <cuda_runtime.h>
#include <cutil_inline.h>

// constants
const unsigned int mesh_width = 128;
const unsigned int mesh_height = 128;
const unsigned int RestartIndex = 0xffffffff;

typedef struct {
  GLuint vbo;
  GLuint typeSize;
```

```
#ifdef USE_CUDA3
  struct cudaGraphicsResource *cudaResource;
#else
  void* space;
#endif
} mappedBuffer_t;

extern "C"
void launch_kernel(float4* pos, uchar4* posColor,
                   unsigned int mesh_width, unsigned int mesh_height, float time);

// vbo variables
mappedBuffer_t vertexVBO = {NULL, sizeof(float4), NULL};
mappedBuffer_t colorVBO =  {NULL, sizeof(uchar4), NULL};

////////////////////////////////////////////////////////////////////////////////
//! Create VBO
////////////////////////////////////////////////////////////////////////////////
//void createVBO(GLuint* vbo, unsigned int typeSize)
void createVBO(mappedBuffer_t* mbuf)
{
  // create buffer object
  glGenBuffers(1, &(mbuf->vbo) );
  glBindBuffer(GL_ARRAY_BUFFER, mbuf->vbo);

  // initialize buffer object
  unsigned int size = mesh_width * mesh_height * mbuf->typeSize;
  glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);

  glBindBuffer(GL_ARRAY_BUFFER, 0);

#ifdef USE_CUDA3
  cudaGraphicsGLRegisterBuffer( &(mbuf->cudaResource), mbuf->vbo,
                                cudaGraphicsMapFlagsNone );
#else
  // register buffer object with CUDA
  cudaGLRegisterBufferObject(mbuf->vbo);
#endif
}

////////////////////////////////////////////////////////////////////////////////
//! Delete VBO
////////////////////////////////////////////////////////////////////////////////
//void deleteVBO(GLuint* vbo)
void deleteVBO(mappedBuffer_t* mbuf)
{
  glBindBuffer(1, mbuf->vbo );
  glDeleteBuffers(1, &(mbuf->vbo) );

#ifdef USE_CUDA3
  cudaGraphicsUnregisterResource( mbuf->cudaResource );
  mbuf->cudaResource = NULL;
  mbuf->vbo = NULL;
#else
  cudaGLUnregisterBufferObject( mbuf->vbo );
  mbuf->vbo = NULL;
#endif

}

void cleanupCuda()
{
  deleteVBO(&vertexVBO);
  deleteVBO(&colorVBO);
}

////////////////////////////////////////////////////////////////////////////////
//! Run the Cuda part of the computation
////////////////////////////////////////////////////////////////////////////////
void runCuda()
{
    // map OpenGL buffer object for writing from CUDA
    float4 *dptr;
    uchar4 *cptr;
    uint *iptr;
#ifdef USE_CUDA3
    size_t start;
    cudaGraphicsMapResources( 1, &vertexVBO.cudaResource, NULL );
    cudaGraphicsResourceGetMappedPointer( ( void ** )&dptr, &start,
                                          vertexVBO.cudaResource );
    cudaGraphicsMapResources( 1, &colorVBO.cudaResource, NULL );
    cudaGraphicsResourceGetMappedPointer( ( void ** )&cptr, &start,
                                          colorVBO.cudaResource );
#else
    cudaGLMapBufferObject((void**)&dptr, vertexVBO.vbo);
    cudaGLMapBufferObject((void**)&cptr, colorVBO.vbo);
#endif

    // execute the kernel
    launch_kernel(dptr, cptr, mesh_width, mesh_height, animTime);

    // unmap buffer object
#ifdef USE_CUDA3
```

```
    cudaGraphicsUnmapResources( 1, &vertexVBO.cudaResource, NULL );
    cudaGraphicsUnmapResources( 1, &colorVBO.cudaResource, NULL );
#else
    cudaGLUnmapBufferObject(vertexVBO.vbo);
    cudaGLUnmapBufferObject(colorVBO.vbo);
#endif
}

void initCuda(int argc, char** argv)
{
  // First initialize OpenGL context, so we can properly set the GL
  // for CUDA.  NVIDIA notes this is necessary in order to achieve
  // optimal performance with OpenGL/CUDA interop.  use command-line
  // specified CUDA device, otherwise use device with highest Gflops/s
  if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") ) {
    cutilGLDeviceInit(argc, argv);
  } else {
    cudaGLSetGLDevice( cutGetMaxGflopsDeviceId() );
  }

  createVBO(&vertexVBO);
  createVBO(&colorVBO);
  // make certain the VBO gets cleaned up on program exit
  atexit(cleanupCuda);

  runCuda();

}

void renderCuda(int drawMode)
{
  glBindBuffer(GL_ARRAY_BUFFER, vertexVBO.vbo);
  glVertexPointer(4, GL_FLOAT, 0, 0);
  glEnableClientState(GL_VERTEX_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER, colorVBO.vbo);
  glColorPointer(4, GL_UNSIGNED_BYTE, 0, 0);
  glEnableClientState(GL_COLOR_ARRAY);

  switch(drawMode) {
  case GL_LINE_STRIP:
    for(int i=0 ; i < mesh_width*mesh_height; i+= mesh_width)
      glDrawArrays(GL_LINE_STRIP, i, mesh_width);
    break;
  case GL_TRIANGLE_FAN: {
    static GLuint* qIndices=NULL;
    int size = 5*(mesh_height-1)*(mesh_width-1);

    if(qIndices == NULL) { // allocate and assign trianglefan indicies
      qIndices = (GLuint *) malloc(size*sizeof(GLint));
      int index=0;
      for(int i=1; i < mesh_height; i++) {
        for(int j=1; j < mesh_width; j++) {
          qIndices[index++] = (i)*mesh_width + j;
          qIndices[index++] = (i)*mesh_width + j-1;
          qIndices[index++] = (i-1)*mesh_width + j-1;
          qIndices[index++] = (i-1)*mesh_width + j;
          qIndices[index++] = RestartIndex;
        }
      }
    }
    glPrimitiveRestartIndexNV(RestartIndex);
    glEnableClientState(GL_PRIMITIVE_RESTART_NV);
    glDrawElements(GL_TRIANGLE_FAN, size, GL_UNSIGNED_INT, qIndices);
    glDisableClientState(GL_PRIMITIVE_RESTART_NV);
  } break;
  default:
    glDrawArrays(GL_POINTS, 0, mesh_width * mesh_height);
    break;
  }

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_COLOR_ARRAY);
}
```

As can be seen, the call to `renderCuda` has been modified to support several drawing modes based on a parameter `drawMode`. Once inside the `renderCuda` routine, the following code tells OpenGL to bind the buffer `vbo` as a float4 vertex array:

```
  glBindBuffer(GL_ARRAY_BUFFER, vertexVBO.vbo);
  glVertexPointer(4, GL_FLOAT, 0, 0);
  glEnableClientState(GL_VERTEX_ARRAY);
```

Similarly, the `colorVBO` buffer is bound as uchar4 array. This requires the CUDA kernel specifiy the color at each vertex in RGB format according to `glColorPointer`.

```
  glBindBuffer(GL_ARRAY_BUFFER, colorVBO.vbo);
  glColorPointer(4, GL_UNSIGNED_BYTE, 0, 0);
  glEnableClientState(GL_COLOR_ARRAY);
```

A `switch` statement based on `BdrawMode` is used to draw the image according to the user's input.

Drawing colored points is requires only a single call:

```
    glDrawArrays(GL_POINTS, 0, mesh_width * mesh_height);
```

The most straightforward way to draw lines was utilized by looping over the rows of the mesh:

```
case GL_LINE_STRIP:
   for(int i=0 ; i < mesh_width*mesh_height; i+= mesh_width)
     glDrawArrays(GL_LINE_STRIP, i, mesh_width);
   break;
```

As discussed earlier in this article, primitive restart was utilized to render colored surfaces. This example takes the unusual approach of declaring the pointer to the `qIndices` array as a static pointer within the case statement that only gets allocated and initialized once during the life of the program. This was purposely done to help the reader experiment with alternative drawing methods, index layout, and drawing modes by keeping the allocation, initialization, and usage local to the drawing mode. As a general rule, such use of a static variable should be avoided.

```
  case GL_TRIANGLE_FAN: {
    static GLuint* qIndices=NULL;
    int size = 5*(mesh_height-1)*(mesh_width-1);

    if(qIndices == NULL) { // allocate and assign trianglefan indicies
      qIndices = (GLuint *) malloc(size*sizeof(GLint));
      int index=0;
      for(int i=1; i < mesh_height; i++) {
        for(int j=1; j < mesh_width; j++) {
          qIndices[index++] = (i)*mesh_width + j;
          qIndices[index++] = (i)*mesh_width + j-1;
          qIndices[index++] = (i-1)*mesh_width + j-1;
          qIndices[index++] = (i-1)*mesh_width + j;
          qIndices[index++] = RestartIndex;
        }
      }
    }
    glPrimitiveRestartIndexNV(RestartIndex);
    glEnableClientState(GL_PRIMITIVE_RESTART_NV);
    glDrawElements(GL_TRIANGLE_FAN, size, GL_UNSIGNED_INT, qIndices);
    glDisableClientState(GL_PRIMITIVE_RESTART_NV);
  } break;
```

Finally the OpenGL client state machine is informed that the vertex and color arrays are disabled and `renderCuda` returns.

```
  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_COLOR_ARRAY);
```

The callbacksVBO.cpp File

The file callbacksVBO.cpp defines keyboard, mouse and display events.

The keyboard routine is very simple. Basically it allows the user to toggle through the display modes (point, line, surface) by pressing the "d" or "D" key.

The mouse and motion routines work in concert with each other modify the values of the `rotate_x` and `rotate_y` variables based on user mouse movements and the state of the mouse buttons.

Most of the work occurs in the display routine that defines the view transforms as shown below:

```
  // set view matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, translate_z);
  glRotatef(rotate_x, 1.0, 0.0, 0.0);
  glRotatef(rotate_y, 0.0, 1.0, 0.0);

  // run CUDA kernel to generate vertex positions
  runCuda();

  // render the data
  renderCuda(drawMode);
```

The CUDA kernel is then called to create the data with `runCuda` and render it with `renderCuda`.

The buffers are swapped so the latest version can be made visible and GLUT is informed that the display needs to be updated. The animation time is also incremented.

```
  glutSwapBuffers();
  glutPostRedisplay();

  animTime += 0.01;
```

The complete source for callbacksVBP.cpp is as follows:

```
//callbacksVBO.cpp (Rob Farber)
// includes, GL
#include <GL/glew.h>

// includes
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

extern float animTime;

// The user must create the following routines:
void initCuda(int argc, char** argv);
void runCuda();
void renderCuda(int);

// Callbacks

int drawMode=GL_TRIANGLE_FAN; // the default draw mode

// mouse controls
int mouse_old_x, mouse_old_y;
int mouse_buttons = 0;
float rotate_x = 0.0, rotate_y = 0.0;
float translate_z = -3.0;

//! Display callback for GLUT
//! Keyboard events handler for GLUT
//! Display callback for GLUT
void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // set view matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, translate_z);
  glRotatef(rotate_x, 1.0, 0.0, 0.0);
  glRotatef(rotate_y, 0.0, 1.0, 0.0);

  // run CUDA kernel to generate vertex positions
  runCuda();

  // render the data
  renderCuda(drawMode);

  glutSwapBuffers();
  glutPostRedisplay();

  animTime += 0.01;
}

//! Keyboard events handler for GLUT
void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case(27) :
    exit(0);
    break;
  case 'd':
  case 'D':
    switch(drawMode) {
    case GL_POINTS: drawMode = GL_LINE_STRIP; break;
    case GL_LINE_STRIP: drawMode = GL_TRIANGLE_FAN; break;
    default: drawMode=GL_POINTS;
    } break;
  }
  glutPostRedisplay();
}

// Mouse event handlers for GLUT
void mouse(int button, int state, int x, int y)
{
  if (state == GLUT_DOWN) {
    mouse_buttons |= 1<<button;
  } else if (state == GLUT_UP) {
    mouse_buttons = 0;
  }

  mouse_old_x = x;
  mouse_old_y = y;
  glutPostRedisplay();
}

void motion(int x, int y)
{
```

```
  float dx, dy;
  dx = x - mouse_old_x;
  dy = y - mouse_old_y;

  if (mouse_buttons & 1) {
    rotate_x += dy * 0.2;
    rotate_y += dx * 0.2;
  } else if (mouse_buttons & 4) {
    translate_z += dy * 0.01;
  }

  mouse_old_x = x;
  mouse_old_y = y;
}
```

A Simple CUDA Kernel

The first example kernel draws a time-varying sinusoidal surface using a simple adaptation of the NVIDIA kernel provided in the simpleGL SDK example. Essentially, each height value stored in `pos` is updated based on position in the mesh, a frequency and the animation time passed via the parameter `time`.

```
    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u*freq + time) * cosf(v*freq + time) * 0.5f;
```

The position and height information is stored in the float4 position array:

```
    // write output vertex
    pos[y*width+x] = make_float4(u, w, v, 1.0f);
```

Similarly, the colors are calculated based on position in the mesh and the animation time:

```
    // write the color
    colorPos[y*width+x].w = 0;
    colorPos[y*width+x].x = 255.f *0.5*(1.f+sinf(w+x));
    colorPos[y*width+x].y = 255.f *0.5*(1.f+sinf(x)*cosf(y));
    colorPos[y*width+x].z = 255.f *0.5*(1.f+sinf(w+time/10.f));
```

The complete listing for kernelVBO is as follows:

```
// kernelVBO.cpp (Rob Farber)
// Simple kernel to modify vertex positions in sine wave pattern
__global__ void kernel(float4* pos, uchar4 *colorPos,
                       unsigned int width, unsigned int height, float time)
{
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

    // calculate uv coordinates
    float u = x / (float) width;
    float v = y / (float) height;
    u = u*2.0f - 1.0f;
    v = v*2.0f - 1.0f;

    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u*freq + time) * cosf(v*freq + time) * 0.5f;

    // write output vertex
    pos[y*width+x] = make_float4(u, w, v, 1.0f);
    colorPos[y*width+x].w = 0;
    colorPos[y*width+x].x = 255.f *0.5*(1.f+sinf(w+x));
    colorPos[y*width+x].y = 255.f *0.5*(1.f+sinf(x)*cosf(y));
    colorPos[y*width+x].z = 255.f *0.5*(1.f+sinf(w+time/10.f));
}

// Wrapper for the __global__ call that sets up the kernel call
extern "C" void launch_kernel(float4* pos, uchar4* colorPos,
                              unsigned int mesh_width, unsigned int mesh_height, float time)
{
  // execute the kernel
  dim3 block(8, 8, 1);
  dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);
  kernel<<< grid, block>>>(pos, colorPos, mesh_width, mesh_height, time);
}
```

The program can be compiled under Linux with the following after the complete source of each of the files has been saved in correctly named files:

```
#/bin/bash
SDK_PATH= PATH_TO_NVIDIA_GPU_Computing_SDK

nvcc -O3 -L $SDK_PATH/C/lib -I $SDK_PATH/C/common/inc simpleGLmain.cpp simpleVBO.cpp callbacksVBO.cpp kernelVBO.cu -lglut -lGLEW -lcutil_x
```

Microsoft users should consult the GPUcomputing.net website for information about building the examples with Microsoft Visual Studio.

Figures 10, 11, and 12 are images created with this program. You will likely see something similar, but the colors and shape of the surface do dynamically change with time. Also, the orientation of the surface in three-space also affects what is shown on the screen.

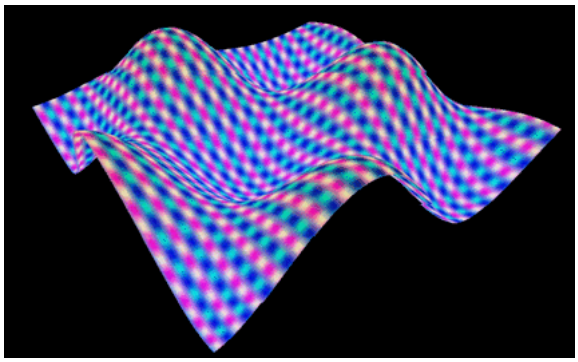

Figure 10: Example surface sinusoidal surface.

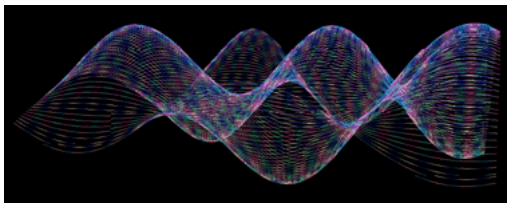

Figure 11: Wire-frame image.

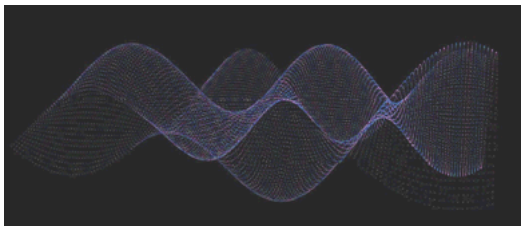

Figure 12: Points drawn in 3D.

Artificial Terrain With a Perlin Noise Generator

Now let's have some fun and create a 3D virtual terrain world!

The following example adapts the Improved Perlin Noise generator from Part 15 to generate 3D coordinates and color based on elevation. For simplicity, `fBm` (Fractal Brownian Motion) was chosen to generate the fractal terrain. Better methods exist to create more realistic landscapes as noted here.

Essentially four changes were made to the original Part 15 code:

- The call to `k_perlin` was modified so the position and color buffers can be passed to the kernel.
- `launch_kernel` was modified to pass the position and color buffers.
- The routine `fBm` was added to calculate Fractal Brownian Motion.
- The terrain was colored according to elevation.

```
colorPos[idx] = colorElevation(w);
float u = ((float) (idx%width))/(float) width;
float v = ((float) (idx/width))/(float) height;
u = u*2.f - 1.f; // center the mesh
v = v*2.f - 1.f;
w = (w>0.f)?w:0.f; // don't show region underwater
pos[idx] = make_float4( u, w, v, 1.0f);
```

The complete listing for perlinKernelVBO.cu is as follows:

```
// perlinKernelVBO.cu (Rob Farber)
#include <cutil_math.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>

float gain=0.75f;
float xStart=2.f;
float yStart=1.f;
```

```
float zOffset = 0.0f;
float octaves = 2.f;
float lacunarity = 2.0f;
#define Z_PLANE 50.f

__constant__ unsigned char c_perm[256];
__shared__ unsigned char s_perm[256]; // shared memory copy of permuation array
unsigned char* d_perm=NULL; // global memory copy of permutation array
// host version of permutation array
const static unsigned char h_perm[] = {151,160,137,91,90,15,
   131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
   190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
   88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
   77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
   102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
   135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
   5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
   223,183,170,213,119,248,152,2,44,154,163, 70,221,153,101,155,167, 43,172,9,
   129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
   251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
   49,192,214, 31,181,199,106,157,184,84,204,176,115,121,50,45,127, 4,150,254,
   138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
   };

__device__ inline int perm(int i) { return(s_perm[i&0xff]); }
__device__ inline float fade(float t) { return t * t * t * (t * (t * 6.f - 15.f) + 10.f); }
__device__ inline float lerpP(float t, float a, float b) { return a + t * (b - a); }
__device__ inline float grad(int hash, float x, float y, float z) {
  int h = hash & 15;                      // CONVERT LO 4 BITS OF HASH CODE
  float u = h<8 ? x : y,                  // INTO 12 GRADIENT DIRECTIONS.
    v = h<4 ? y : h==12||h==14 ? x : z;
  return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
}

__device__ float inoise(float x, float y, float z) {
  int X = ((int)floorf(x)) & 255, // FIND UNIT CUBE THAT
    Y = ((int)floorf(y)) & 255,   // CONTAINS POINT.
    Z = ((int)floorf(z)) & 255;
  x -= floorf(x);               // FIND RELATIVE X,Y,Z
  y -= floorf(y);               // OF POINT IN CUBE.
  z -= floorf(z);
  float u = fade(x),            // COMPUTE FADE CURVES
    v = fade(y),                // FOR EACH OF X,Y,Z.
    w = fade(z);
  int A = perm(X)+Y, AA = perm(A)+Z, AB = perm(A+1)+Z, // HASH COORDINATES OF
    B = perm(X+1)+Y, BA = perm(B)+Z, BB = perm(B+1)+Z; // THE 8 CUBE CORNERS,

  return lerpP(w, lerpP(v, lerpP(u, grad(perm(AA), x  , y  , z   ), // AND ADD
                             grad(perm(BA), x-1.f, y  , z   )),   // BLENDED
                        lerpP(u, grad(perm(AB), x  , y-1.f, z   ),    // RESULTS
                             grad(perm(BB), x-1.f, y-1.f, z   ))),     // FROM  8
               lerpP(v, lerpP(u, grad(perm(AA+1), x  , y  , z-1.f ),  // CORNERS
                             grad(perm(BA+1), x-1.f, y  , z-1.f )),    // OF CUBE
                        lerpP(u, grad(perm(AB+1), x  , y-1.f, z-1.f ),
                             grad(perm(BB+1), x-1.f, y-1.f, z-1.f ))));
#ifdef ORIG
  return(perm(X));
#endif

}

__device__ float fBm(float x, float y, int octaves,
                     float lacunarity = 2.0f, float gain = 0.5f)
{
  float freq = 1.0f, amp = 0.5f;
  float sum = 0.f;
  for(int i=0; i<octaves; i++) {
    sum += inoise(x*freq, y*freq, Z_PLANE)*amp;
    freq *= lacunarity;
    amp *= gain;
  }
  return sum;
}


__device__ inline uchar4 colorElevation(float texHeight)
{
  uchar4 pos;

  // color textel (r,g,b,a)
       if (texHeight < -1.000f) pos = make_uchar4(000, 000, 128, 255); //deeps
  else if (texHeight < -.2500f) pos = make_uchar4(000, 000, 255, 255); //shallow
  else if (texHeight < 0.0000f) pos = make_uchar4(000, 128, 255, 255); //shore
  else if (texHeight < 0.0125f) pos = make_uchar4(240, 240, 064, 255); //sand
  else if (texHeight < 0.1250f) pos = make_uchar4(032, 160, 000, 255); //grass
  else if (texHeight < 0.3750f) pos = make_uchar4(224, 224, 000, 255); //dirt
  else if (texHeight < 0.7500f) pos = make_uchar4(128, 128, 128, 255); //rock
  else                          pos = make_uchar4(255, 255, 255, 255); //snow

  return(pos);
}

void checkCUDAError(const char *msg) {
```

```
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}

//Simple kernel fills an array with perlin noise
__global__ void k_perlin(float4* pos, uchar4 *colorPos,
                         unsigned int width, unsigned int height,
                        float2 start, float2 delta, float gain, float zOffset,
                        unsigned char* d_perm, float octaves, float lacunarity)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float xCur = start.x + ((float) (idx%width)) * delta.x;
  float yCur = start.y + ((float) (idx/width)) * delta.y;

  if(threadIdx.x < 256)
    // Optimization: this causes bank conflicts
    s_perm[threadIdx.x] = d_perm[threadIdx.x];
  // this synchronization can be important if there are more that 256 threads
  __syncthreads();


  // Each thread creates one pixel location in the texture (textel)
  if(idx < width*height) {
    float w = fBm(xCur, yCur, octaves, lacunarity, gain) + zOffset;

    colorPos[idx] = colorElevation(w);
    float u = ((float) (idx%width))/(float) width;
    float v = ((float) (idx/width))/(float) height;
    u = u*2.f - 1.f;  // center the mesh
    v = v*2.f - 1.f;
    w = (w>0.f)?w:0.f; // don't show region underwater
    pos[idx] = make_float4( u, w, v, 1.0f);
  }
}


uchar4 *eColor=NULL;
// Wrapper for the __global__ call that sets up the kernel call
extern "C" void launch_kernel(float4 *pos, uchar4* posColor,
                              unsigned int image_width, unsigned int image_height,
                              float time)
{
  int nThreads=256; // must be equal or larger than 256! (see s_perm)
  int totalThreads = image_height * image_width;
  int nBlocks = totalThreads/nThreads;
  nBlocks += ((totalThreads%nThreads)>0)?1:0;

  float xExtent = 10.f;
  float yExtent = 10.f;
  float xDelta = xExtent/(float)image_width;
  float yDelta = yExtent/(float)image_height;

  if(!d_perm) { // for convenience allocate and copy d_perm here
    cudaMalloc((void**) &d_perm,sizeof(h_perm));
    cudaMemcpy(d_perm,h_perm,sizeof(h_perm),cudaMemcpyHostToDevice);
    checkCUDAError("d_perm malloc or copy failed!");
  }

  k_perlin<<< nBlocks, nThreads>>>(pos, posColor, image_width, image_height,
                                   make_float2(xStart, yStart),
                                   make_float2(xDelta, yDelta),
                                   gain, zOffset, d_perm,
                                   octaves, lacunarity);

  // make certain the kernel has completed
  cudaThreadSynchronize();
  checkCUDAError("kernel failed!");
}
```

Minor changes were also made to the callbacksVBO.cpp file to so the mouse movements with the middle button pressed translate the image along the Z-axis. In addition support for the keyboard commands in Table 2 was added.

Key	Action
+	Lower the ocean level
-	Raise the ocean level
k	Vi type key command to move terrain up
J	Vi type command to move terrain down
h	Vi type command to move terrain left
l	Vi type command to move terrain right
d	Toggle draw mode
D	Toggle draw mode
I	Increase gain by 0.25
i	Decrease gain by 0.25
O	Increase octaves (i.e. number of frequencies in the fBm) by 1
o	Decrease octaves (i.e. number of frequencies in the fBm) by 1
P	Increase lacunarity (or the gap between successive frequencies) by 0.25
p	Decrease lacunarity (or the gap between successive frequencies) by 0.25

Table 2: Keyboard commands.

The following is the complete source for perlinCallbacksVBO.cpp.

```
// perlinCallbacksVBO.cpp (Rob Farber)
// includes, GL
#include <GL/glew.h>

// includes
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

extern float animTime;

// The user must create the following routines:
void initCuda(int argc, char** argv);
void runCuda();
void renderCuda(int);

// Callbacks

int drawMode=GL_TRIANGLE_FAN; // the default draw mode

// mouse controls
int mouse_old_x, mouse_old_y;
int mouse_buttons = 0;
float rotate_x = 0.0, rotate_y = 0.0;
float translate_z = -3.0;

//! Display callback for GLUT
//! Keyboard events handler for GLUT
//! Display callback for GLUT
void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // set view matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, translate_z);
  glRotatef(rotate_x, 1.0, 0.0, 0.0);
  glRotatef(rotate_y, 0.0, 1.0, 0.0);

  // run CUDA kernel to generate vertex positions
  runCuda();

  // render the data
  renderCuda(drawMode);

  glutSwapBuffers();
  glutPostRedisplay();

  animTime += 0.01;
}

extern float xStart,yStart,zOffset;
extern float gain, octaves, lacunarity;

//! Keyboard events handler for GLUT
void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case(27) : // exit
  case('q') :
    exit(0);
    break;
  case '+': // lower the ocean level
    zOffset += 0.01;
    zOffset = (zOffset > 1.0)? 1.0:zOffset; // guard input
    break;
  case '-': // raise the ocean level
    zOffset -= 0.01;
    zOffset = (zOffset < -1.0)? -1.0:zOffset; // guard input
    break;
  case 'k': // move withing Perlin function
    yStart -= 0.1;
    break;
  case 'j':
    yStart += 0.1;
    break;
  case 'l':
    xStart += 0.1;
    break;
  case 'h':
    xStart -= 0.1;
    break;
  case 'd':
  case 'D':
    switch(drawMode) {
```

```
    case GL_POINTS: drawMode = GL_LINE_STRIP; break;
    case GL_LINE_STRIP: drawMode = GL_TRIANGLE_FAN; break;
    default: drawMode=GL_POINTS;
    } break;
  case 'I': // change gain
    gain += 0.25;
    break;
  case 'i': // change gain
    gain -= 0.25;
    gain = (gain < 0.25)?0.25:gain; // guard input
    break;
  case 'O': // change octaves
    octaves += 1.0f;
    octaves = (octaves > 8)?8:octaves; // guard input
    break;
  case 'o': // change octaves
    octaves -= 1.0f;
    octaves = (octaves<2)?2:octaves; // guard input
    break;
  case 'P': // change lacunarity
    lacunarity += 0.25;
    break;
  case 'p': // change lacunarity
    lacunarity -= 0.25;
    lacunarity = (lacunarity<0.2)?0.2:lacunarity; // guard input
    break;
  }
  glutPostRedisplay();
}

// Mouse event handlers for GLUT
void mouse(int button, int state, int x, int y)
{
  if (state == GLUT_DOWN) {
    mouse_buttons |= 1<<button;
  } else if (state == GLUT_UP) {
    mouse_buttons = 0;
  }

  mouse_old_x = x;
  mouse_old_y = y;
  glutPostRedisplay();
}

void motion(int x, int y)
{
  float dx, dy;
  dx = x - mouse_old_x;
  dy = y - mouse_old_y;

  if (mouse_buttons & 1) {
    rotate_x += dy * 0.2;
    rotate_y += dx * 0.2;
  } else if (mouse_buttons & 4) {
    translate_z += dy * 0.01;
  }
 rotate_x = (rotate_x < -60.)?-60.:(rotate_x > 60.)?60:rotate_x;
 rotate_y = (rotate_y < -60.)?-60.:(rotate_y > 60.)?60:rotate_y;

  mouse_old_x = x;
  mouse_old_y = y;
}
```

The code can be built with the following command after the complete source has been saved to appropriately named files:

```
#/bin/bash
SDK_PATH=PATH_TO_NVIDIA_GPU_Computing_SDK

nvcc -O3 -L $SDK_PATH/C/lib -I $SDK_PATH/C/common/inc simpleGLmain.cpp simpleVBO
.cpp perlinCallbacks.cpp perlinKernelVBO.cu -lglut -lGLEW -lcutil_x86_64 -o test
GL
```

Figure 13 is an example terrain that can be created after using the defined keys and mouse movements to make a pleasing picture.

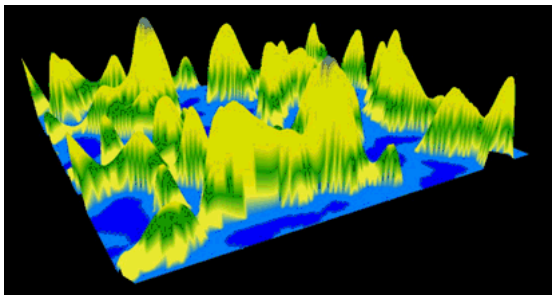

Figure 13: An example 3D surface created using Perlin Noise

Summary

Mixing CUDA and visualization opens tremendous opportunities for commercial games and visual products as well as scientific applications. The examples in this article demonstrate that the current generation of CUDA-enabled graphics processors can both render and generate very complex data at hundreds of frames per second.

In particular, this article attempts to point the way to an extraordinarily simple and flexible way for CUDA developers to generate and render 3D images using the OpenGL standards compliant primitive restart capability so that minimal host processor interaction is required. As a result, PCIe bottlenecks and latencies can be avoided to deliver high-performance high-quality graphics even when the images require irregular meshes and/or computationally expensive data generation. Of course, this is of interest when generating very realistic images on high-end GPUs but do not forget that this same technique can enable product penetration into the mid- and lower-performance markets as well!

Finally, the simple OpenGL software framework introduced in Part 15 of this series that was initially used to generate and view 2D pictures has been adapted -- with minor modification -- to support fully animated and interactive 3D images and landscapes. Hopefully, this software will provide the starting point for many new projects.

As shown in Part 17 of this series, this same software framework can easily be adapted to exploit the full class and inheritance capabilities of C++ plus a vast collection of Cg libraries and other existing visualization software!

References

- CUDA, Supercomputing for the Masses: Part 17
- CUDA, Supercomputing for the Masses: Part 16
- CUDA, Supercomputing for the Masses: Part 15
- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.