Using OpenGL with CUDA

Installing OpenGL and GLUT; compiling with nvcc

Basics of OpenGL and GLUT in C

Interoperability between OpenGL and CUDA

- ***OpenGL*** = ***Open Graphic Library*** → creation of 3D graphic primitives
  → rendering through a virtual camera

- ***GLUT*** = ***OpenGL Utility Toolkit*** → creation of a window
  → time loop for real-time animation and rendering

Installing OpenGL and GLUT; compiling with nvcc

installation for Windows 7 - 64 bits and Visual Studio 2008:

- download **glut-3.7.6-bin.zip**
- download **freeglut-MSVC-2.8.0-1.mp.zip**
  from **http://www.transmissionzero.co.uk/software/freeglut-devel/**

- unzip some files and install them on **C:** drive:

glut-3.7.6-bin.zip\glut-3.7.6-bin → **glut32.lib** → **C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\lib\amd64**

freeglut-MSVC-2.8.0-1.mp.zip\freeglut\include\GL → **glut.h , freeglut.h , freeglut_ext.h , freeglut_std.h** → **C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\include\GL**

freeglut-MSVC-2.8.0-1.mp.zip\freeglut\lib\x64 → **freeglut.lib** → **C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\lib\amd64**

freeglut-MSVC-2.8.0-1.mp.zip\freeglut\bin\x64 → **freeglut.dll** → **C:\Windows**

include file:

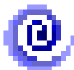
#include <GL/glut.h> → for both GLUT *and* OpenGL *and* GLU

must be written *after* **#include <stdio.h>** and **#include <stdlib.h>**

do *not* include **<gl.h>** and **<glu.h>**

compilation with nvcc 3.2 for a 64-bit executable:

- simply add **freeglut.lib** in the **nvcc** command line instruction


example: compiling **P.cu**

```
call "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat"

"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\bin\nvcc"
        -O  -arch sm_20  -Xcompiler -O2  P.cu  freeglut.lib  -o  P.exe
```

Basics of OpenGL and GLUT in C

window and animation loop with GLUT functions:

- *callback functions:*

window
→ callback function **reshape** is called each time the window's shape is modified

animation loop
→ callback function **evolve** is called each time GLUT is idle
= { update *degrees of freedom*; request a repaint of the window }
→ callback function **display** is called to (re)paint the window

initializations inside **main(int argc , char *argv[])** *:*

start GLUT: **glutInit(argc, argv);**

set the display mode: **glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);**

code value(s) combined with bitwise-OR's
GLUT_DOUBLE → *double-buffering*
GLUT_RGB → *RGBA mode*

set the initial window size: **glutInitWindowSize(w , h);**

width, height

set the initial window position: **glutInitWindowPosition(x , y);**

position of upper-left corner

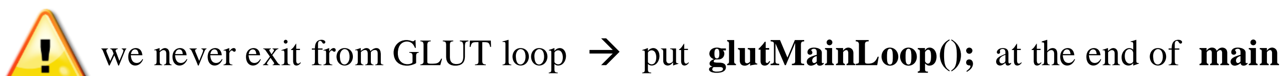
create the window: **glutCreateWindow("window name");**

declare **reshape** as the callback *reshape* function: **glutReshapeFunc(reshape);**

declare **display** as the callback *display* function: **glutDisplayFunc(display);**

declare **evolve** as the callback *idle* function: **glutIdleFunc(evolve);**

enter the *glut loop*: **glutMainLoop();**



we never exit from GLUT loop → put **glutMainLoop();** at the end of **main**

- *callback function* **reshape(int w , int h)** *:*

  → define the window's shape and the optic of the virtual camera

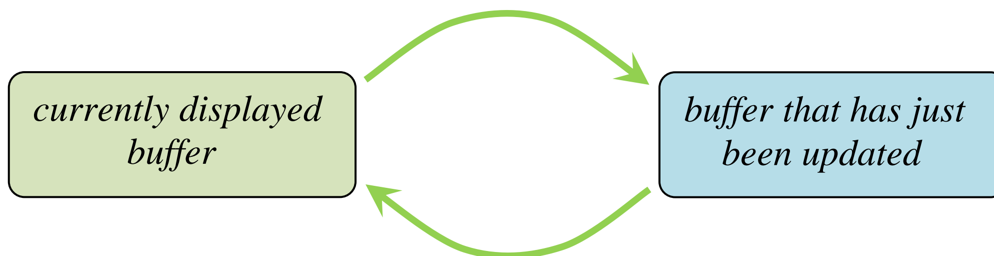
  reshape is first called just after window creation

  window resized by user → OpenGL recalls **reshape** with new size (w , h)

- *callback function* **display()** *:*

  → display the graphic primitives (vertices...)

  at the end of display, swap the two buffers of the window: **glutSwapBuffers();**



- *callback function* **evolve()** *:*

  → update the values of the *degrees of freedom*

  at the end of **evolve**, request **display** to be called again: **glutPostRedisplay();**

no arguments allowed for these callback functions (except w , h for **reshape**)
→ use C external variable(s) to access program data

graphic primitives and rendering with OpenGL:

- *inside* **main(int argc , char *argv[])** *:*

```
glClearColor(r, g, b, a);
```

set the clearing color

- *inside* **reshape** *:*

```
glViewport (0, 0, w, h);

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

glOrtho(X0 , X1 , Y0 , Y1 , Z0 , Z1);
```

window's shape (glViewport)

optic of camera: orthographic projection (no perspective) of $[X_0, X_1] \times [Y_0, Y_1]$ along Z axis (only objects inside $[Z_0, Z_1]$) (glMatrixMode, glLoadIdentity, glOrtho)

- *inside* **display** *:*

```
glClear(GL_COLOR_BUFFER_BIT);

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();
```

clear up window with the clearing color (glClear)

(identity) transformation from camera's coordinate system toward absolute coordinate system (glMatrixMode, glLoadIdentity)

create some vertices between **glBegin(GL_POINTS);** and **glEnd();** :

→ for each vertex (x , y , z) of color (r , g , b) :

```
glColor3f(r , g , b);

glVertex3f(x , y , z);
```

example: animation with a single vertex

```
#include <math.h>
#include <GL/glut.h>

double X = 0.5 , Y = 0.0 , Z = 0.0 , T = 0.0;

void reshape(int w, int h)
{
   glViewport (0, 0,w,  h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-1.0 , 1.0 , -1.0 , 1.0 , 0.0 , 1.0);
}


void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glBegin(GL_POINTS);
      glColor3f(1.0 , 1.0 , 0.0);
      glVertex3f(X , Y , Z);
   glEnd();

   glutSwapBuffers();
}


void evolve(void)
{
   X = 0.5 * cos(T);    Y = 0.5 * sin(T);    Z = 0;    T += 0.01;

   glutPostRedisplay();
}


int main(int argc , char *argv[])
{
   glutInit(&argc, argv);    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize(200, 200);    glutInitWindowPosition(0, 0);
   glutCreateWindow("one vertex");
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glutReshapeFunc(reshape);    glutDisplayFunc(display);    glutIdleFunc(evolve);
   glutMainLoop();
   return 0;
}
```

Interoperability between OpenGL and CUDA

To be developed, see Section 3.2.7 of CUDA Programming Guide

- set the CUDA device with **cudaGLSetGLDevice(index);**

mutually exclusive from **cudaSetDevice(index);**

- OpenGL data can be stored inside the CUDA card's global memory

  → OpenGL vertices can be modified directly and in parallel from a CUDA kernel

  → idem for textures...