EG UK Computer Graphics & Visual Computing (2017)
Tao Ruan Wan and Franck Vidal (Editors)

Supplementary Material for Real-Time Rendering of Molecular Dynamics Simulation Data: A Tutorial

N.Alharbi[1] , M. Chavent[2] and RS. Laramee[1]

[1]Department of computer science, Swansea University, UK
[2] Institute of Pharmacology and Structural Biology (IPBS), Toulouse, France

1. Data Processing

We process two types of files (gro and xtc). The first one, GRO, is used to describe the molecules structure and the XTC file stores the actual dataset.

gro Files contain a molecular structure that describe a trajectory by simply concatenating files [GRO09a]. However, in our example the GRO file is just used to obtain the molecule structure. A sample is included below:

```
SELF-ASSEMBLY
336260
    1ALA CA 1 2.892 3.409 2.771
    2PRO CA 2 3.076 3.738 2.707
    2PRO CB 3 3.033 3.900 2.767
    3LYS CA 4 3.344 3.706 2.460
    3LYS CB 5 3.254 3.391 2.339
    3LYS CG 6 3.231 3.204 2.403
    4ASP CA 7 3.706 3.775 2.437
    4ASP CB 8 3.778 3.727 2.211
    ..
    ..
    ..
    37698POPG C4B75914 94.693 104.448 6.085
    37698POPG C5B75915 94.632 104.578 5.616
116.01860 116.01860 10.13590
```

Lines contain the following information (top to bottom):

- title string (free format string, optional time in ps after 't=')
- number of atoms (free format integer)
- one line for each atom (fixed format, see below)
- box vectors (free format, space separated reals), values: v1(x) v2(y) v3(z) v1(y) v1(z) v2(x) v2(z) v3(x) v3(y), the last 6 values may be omitted (they will be set to zero).

This format is fixed, ie. all columns are in a fixed position. Columns contain the following information (from left to right):

- residue number (5 positions, integer)
- residue name (5 characters)
- atom name (5 characters)
- atom number (5 positions, integer)
- position (in nm, x y z in 3 columns, each 8 positions with 3 decimal places)
- velocity (in nm/ps (or km/s), x y z in 3 columns, each 8 positions with 4 decimal places)

We provide a class *GROManager* contains a method *fetchStructure()* that derives the molecules structure from the GRO file.

xtc Files is a portable format for trajectories [GRO09b]. It uses the xdr routines for writing and reading data which was created for the Unix NFS system. The trajectories are written using a reduced precision algorithm which works in the following way: the coordinates (in Nanometres) are multiplied by a scale factor, typically 1000, so that coordinates are in Picometers. These are rounded to integer values. Then several other customization are performed, for instance making use of the fact that atoms close in sequence are usually close in space too (e.g. a water molecule). To this end, the xdr library is extended with a special routine to write 3-D float coordinates.
All the data is stored using calls to xdr routines and can be read using the same xdr routines. We utilize the XDR liblary from GROMACS.
The variable described below are used to obtained data from the XTC file
int magic a magic number, for the current file version its value is 1995.
int natoms the number of atoms in the trajectory.
int time step the simulation step.
float real time in the simulation.
float box[3][3] the computational box which is stored as a set of three basis vectors, to allow for triclinic Periodic boundary condition (PBC).
For a rectangular box the box edges are stored on the diagonal of the matrix.
3dfcoord x[natoms] the coordinates themselves stored in reduced precision. Please note that when the number of atoms is smaller than 9 no reduced precision is used.

XTC Library The XTC library is an open source library under the lesser GNU public license [GRO09c]. It is design to help developers reading and writing xtc, edr and trr files. We utilized this library to read our xtc file. Reading data from XTC file requires three steps: 1) opening the xtc file, 2) looping through the xtc file, and 3) closing the xtc file. The xtc file can be opened via *xdrfile_open()*. This function has two parameters *FileName* and *AccessType*. Then *read_xtc()* can be used inside a logic loop to read frames until no frame found in the file. Finally, the xtc file must be closed via *xdrfile_close()*. We provide a class *XTCManager* that interfaces the xtc file reading functionality. The *XTCManager* class has only one main function *FetchFrames(const unsigned int start, const unsigned int end, short stride=0)*. The start and end parameters hold the first frame and the last frame to be read respectively, and the stride parameter holds the number of frames that we must skip. The stride parameter can be used to get a time-sampled data-set.

2. Context Properties Configuration for OpenGL Interoperability

In order to use OpenGL interoperability, the OpenCL context properties must be configured with respect to the operating system. The *cl_context_properties* is a data structure and it has three fields. The first and the second fields specify the the property name to be configured and its value respectively while the third field must be set to 0.

Configure OpenCL shared context on Apple Platform On Mac OS, the cl_context_properties structure requires only one property [Sca11, MGMG11]: CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE. However, the value associated with this property must have the data type CGLShareGroupObj, and it can be acquired via the function *CGLGetShareGroup()*. This function requires a *CGLContextObj* structure, which can be obtained by calling *CGLGetCurrentContext()*. The following code shows how these functions work together:

```
 CGLContextObj glContext = CGLGetCurrentContext();
 CGLShareGroupObj shGroup = CGLGetShareGroup(glContext);
 cl_context_properties prop[] = {
   CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,
   (cl_context_properties)shGroup,
   0};
```

Configure OpenCL shared context on Linux Platform The configuration of the cl context property on Linux requires three structures: CL_GL_CONTEXT_KHR, CL_GLX_DISPLAY_KHR, and CL_CONTEXT_PLATFORM. There are three functions that can be used to acquire the associated values with these properties *glXGetCurrentContext()* for the first property CL_GL_CONTEXT_KHR and *glXGetCurrentDisplay()* for the second property CL_GLX_DISPLAY_KHR and *clGetPlatformIDs()* to acquire the platform id.

```
clGetPlatformIDs( 1, &cpPlatform, NULL );
 cl_context_properties prop[] = {
   CL_GL_CONTEXT_KHR,
   (cl_context_properties)glXGetCurrentContext(),
   CL_GLX_DISPLAY_KHR,
   (cl_context_properties)glXGetCurrentDisplay(),
   CL_CONTEXT_PLATFORM,
   (cl_context_properties)cpPlatform,
   0};
```

Configure OpenCL shared context on Windows Platform On Windows platform the CL context properties configuration requires setting three properties: CL_GL_CONTEXT_KHR, CL_WGL_HDC_KHR, and CL_CONTEXT_PLATFORM. Similar to Linux, on Windows there are three functions that can be used to acquire the associated values with these properties *wglGetCurrentContext()* for the first property CL_GL_CONTEXT_KHR and *wglGetCurrentDC()* for the second property CL_WGL_HDC_KHR and *clGetPlatformIDs()* to acquire the platform ID.

```
clGetPlatformIDs( 1, &cpPlatform, NULL );
```

The following code shows how to configure the CL context properties for Windows:

```
 cl_context_properties prop[] = {
   CL_GL_CONTEXT_KHR,
   (cl_context_properties)wglGetCurrentContext(),
   CL_WGL_HDC_KHR,
   (cl_context_properties)wglGetCurrentDC(),
   CL_CONTEXT_PLATFORM,
   (cl_context_properties)cpPlatform,
 0};
```

Creating a Shared Context Once the CL context properties have been set, *clCreateContext()* function can be used to create the shared context. This function can create a context with one or more devices. It utilizes the CL context properties that we have configured to create a context with the selected device. The following code shows how to create a context with one device:

```
 sharedGPUContext = clCreateContext(prop, 1, &ourDevices,
NULL, NULL, &erNum);
```

Creating OpenCL memory object From OpenGL Buffer If the shared OpenCL context was successfully created then we can create OpenCL memory objects by invoking *clCreateFromGLBuffer()*. This function requires the OpenCL context to be created from an OpenGL one and it requires the OpenGL buffer to be exist before the function is called. The following example shows how to achieve this goal via *clCreateFromGLBuffer()*:

```
 cl_mem sharedCLGLBuffer = clCreateFromGLBuffer(sharedContext, CL_MEM_WRITE_ONLY,
glBufferID, &errCode);
```

The function returns a pointer to the OpenCL memory object stored in *sharedCLGLBuffer*. The first parameter of the function is a shared OpenCL context. The second parameter is a flag to specify the access permission (CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE). The third parameter specifies the identity of the OpenGL buffer that needs to be set shareable. The last parameter is used to return an appropriate error code.

References

[GRO09a] GROMACS: gro file format, 2009. URL: http://manual.gromacs.org/online/gro.html. 1

[GRO09b] GROMACS: xtc file format, 2009. URL: http://manual.gromacs.org/online/xtc.html. 1

[GRO09c] GROMACS: Xtc library, 2009. URL: http://www.gromacs.org/Developer_Zone/Programming_Guide/XTC_Library. 2

[MGMG11] MUNSHI A., GASTER B., MATTSON T. G., GINSBURG D.: *OpenCL programming guide*. Pearson Education, 2011. 2

[Sca11] SCARPINO M.: *OpenCL in Action: How to Accelerate Graphics and Computations*. Manning Publications, Nov. 2011. URL: http://amazon.com/o/ASIN/1617290173/. 2