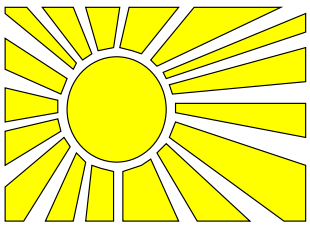
EPCC-SS00-06

Visualisation of 2D and 3D Discrete Element Models
using OpenGL

Dimitrios E. Mitsotakis

Abstract

In many projects involving Discrete Element Models (DEM's), visualisation is very important, not least because animations of DEM's can be extremely visually attractive as well as scientifically informative. In this report, we introduce a simple OpenGL visualiser for DEM's that reads a file containing the particle positions as a function of time and animates them. Moreover we will show how we can add interesting functionality such as colour, lighting, motion etc. We show visualisations of data sets produced by a third-party DEM code that simulates the formation of sand piles. We also present a parallelisation of the serial code using MPI.



Contents

1 Introduction

A Discrete Element Model (DEM) simulates the behaviour of a collection of particles, typically interacting via a short-range force. The simulation works at the level of individual particles and in real cases many millions are typically required to get accurate results. Examples include simulating snooker balls (a 2D DEM with contact forces), the way ball bearings pack as they are emptied into a box (a 3D DEM with complicated contact forces including friction) or the fracturing behaviour of a crystal at the atomic level (a 3D DEM with complicated short-range forces extending at least as far as an atom's nearest neighbours).

In many projects involving DEM's, visualisation is very important, not least because animations of DEM's can be extremely visually attractive as well as scientifically informative.

OpenGL is a simple, portable and efficient standard for 3D graphics, callable from C or Fortran and available under Unix and NT, which is ideal for animation.

1.1 Computer Simulation using Particles

The starting point of the *computational physics* approach to scientific investigation is a mathematical model of the physical phenomenon of interest. The equations of the mathematical model are cast into a discrete algebraic form which is amenable to numerical solution. The discrete algebraic equations describe the *simulation model* which, when expressed as a sequence of computer instructions, provides the computer *simulation program*. The computer plus program then allow the evolution of the model physical system to investigated in *computer experiments*.

Computer simulation may be regarded as the theoretical exercise of numerically solving an initial-value-boundary-value problem. At time $t = 0$, the initial state of the system is specified in some finite region of space (*the computation box*) on the surface of which prescribed boundary conditions hold. The simulation consists of the calculation is the *timestep cycle* in which the state of the physical system is incremented forwards in time by a small timestep, dt. The experimental aspect, and therefore the name computer experiment, arises when we consider the problems of measurements. Even the simplest simulation calculation generates large amounts of data which require an experimental approach to obtain results in a digestible form.

Although the amount of data which can be handled by computers is large, it is nevertheless finite. Much of the ingenuity of computational scientists is devoted to obtaining good simulation models of the physical systems within the constraints of the available finite computer resources. Methods of discretisation used in obtaining simulation models include finite-difference methods (Richtmyer and Morton, 1967), finite-element methods (Strang and Fix, 1973), and the *particle methods*.

"*Particle methods*" is a generic term for the class of simulation models where the discrete representation of physical phenomena involves the use of interacting particles. The name "particle" arose because in most applications the particles may be identified directly with physical objects. Each particle has a set of attributes, such as mass, charge, vorticity, position, momentum. The state of the physical system is defined by the attributes of a finite ensemble of particles and the evolution of the system is determined by the laws of interactions of the particles. A feature which makes particle models computationally attractive is that a number of the particle attributes are conserved quantities and so need no updating as the computer simulation involves in time.

The relationship between the particles of the simulation model and the particles in physical systems is determined largely by the interplay of finite computer resources and the length and timescales of importance in the physical systems.

There are three types of particle simulation model; the particle-particle (PP) model, the particle-mesh (PM) model and the particle-particle-particle-mesh (P^3M) model. The PP model uses the action at a distance formulation of the force law, the PM model treads the force as a field quantity-approximating it on a mesh- and the P^3M model is a hybrid of the PP and PM models. The choice of model is dictated partly by the physics of the phenomenon under investigation and partly by consideration of computational costs. For the very short-ranged forces typical of DEM's, the PP model is the most efficient.

For more information see [5].

1.2 The "Mickey Mouse" Code

Dr Joachim Wittmer (then at the Department of Physics in the University of Edinburgh) created a program that simulates a *Sand Pile Formation*. In this simulation, sand grains are represented by composite *grains*, which are collections of spherical *beads* connected by dissipative springs. This program, called "Mickey Mouse", performs the simulation and creates a set of output data files based on input data taken from the file `bataddGT5`. A typical input data file looks like this.

```
cGT5.260
cGT5.261
12345
0.0 0.5 0.001
0. -0.1 0.0
0.1 10.0
1.2  0.5
6  0.01 5
0.5 0.1 100 0.1
#input file name
#output file name
#iseed
#tstart,tend,dt
#Gx,Gy,Gz
#Drag,Damp      0. 0.0
#RLL,RLLmake
#iJT,Jin,iGT
#R,dR,K,dK
```

The output data files contain the characteristics of the grains which are falling under gravity and creating the sand pile; data includes the position, size, energy etc. These characteristics are very interesting but it is extremely difficult to understand them when presented as raw data. That is why we need to visualise them and see how what these numbers really mean.

We ran the code with $dt = 0.001$ and wrote the output at intervals of $t = 0.5$. New grains were dropped onto the pile at intervals of $t = 2.0$. In total we created 460 data files which we aimed to visualise using OpenGL.

1.3 Starting with OpenGL

OpenGL is a software interface to graphics hardware. This interface consists of about 250 distinct commands (about 200 in the core OpenGL and another 50 in the OpenGL Utility Library) that you use to specify the objects and operations needed to produce interactive three-dimensional applications. In this project we used the freely available Mesa implementation for simplicity.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing task or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you are using. Similarly, OpenGL does not provide high-level commands for describing models of three dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules.

A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modelling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation. For more information see [1], [2], [3].

For all OpenGL applications, we must include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which requires inclusion of the glu.h header file. So almost every OpenGL source file begins with:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

There is also another useful library, the OpenGL Utility Toolkit (GLUT). You can use GLUT for managing your window manager tasks. If you are using GLUT, you should include:

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

Most OpenGL applications also use standard C library

system calls, so it is common to include header files that are not related to graphics, such as:

```
#include <stdlib.h>
#include <stdio.h>
```

In these notes we use C but the FORTRAN programmers should look at [4].

We must also mention that some OpenGL commands accept as many as eight different data types for their arguments. For example there are the GLbyte, GLshort, GLint, GLfloat, GLdouble for signed char, short, int, float and double,respectively to the c language data types.

2 Visualisation of Particles

In this section we discuss the way that we tackle the visualisation of Particles.

We start with a file which contains the data and the other characteristics of the particles. We represent particles with the same characteristics, with spheres of the same colour, so as to easily discriminate between the particles. So we read the file and render every sphere with different colour in terms of its radius.

First of all we define the material properties for the spheres. We will use two different kinds of spheres for the particles. So we will use pink and yellow spheres in terms of its radius. We will also use light so as to give to the image a three dimensional form.

2.1 Colour and Light

The first thing that we must mention is the way in which colours of pixels are stored in graphics hardware known as *bitplanes*. There are two methods of storage. Either the red, green, blue, and alpha (RGBA) values of a pixel can be directly stored in the bitplanes, or a single index value that references a colour lookup table is stored. RGBA-colour-display mode is more commonly used and this is used throughout this project.

What we always do in the beginning is to set the current clearing colour for use in clearing colour buffers in RGBA mode. For this job we use the

```
glClearColor(0.1, 0.2, 0.6,1.0);
```

And when we want to clear specified buffers to their current clearing values we use the **glClear**(GLbitfield *mask*); command. The *mask* argument is a bitwise logical OR combination of the values listed bellow:

Buffer	**Name**
Color buffer	GL_COLOR_BUFFER_BIT
Depth buffer	GL_DEPTH_BUFFER_BIT

Before issuing a command to clear multiple buffers, we have to set the values to which each buffer is to be cleared if we want something other than the default RGBA colour and depth value. The **glClearColor()** and **glClearDepth()** commands set the current values for clearing the colour and depth buffers.

OpenGL allows us to specify multiple buffers because clearing is generally a slow operation, since every pixel in the window (possibly millions) is touched, and some graphics hardware allows sets of buffers to be cleared simultaneously. Hardware that doesn't support simultaneous clears performs them sequentially. The difference between

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

and

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

is that although both have the same final effect, the first example might run faster on many machines. It certainly won't run more slowly. So we use the first one.

In red, green, blue, alpha (RGBA) mode, we use the **glColor*()** command to select a current colour. This command sets the current red, green, blue, and alpha values. It can have up to three suffixes, which differentiate variations of the parameters accepted (see [1], [2]).

The next step is the definition of light. When we look at a physical surface, our eyes perception of the colour depends on the distribution of photon energies that arrive at the trigger of our cone cells. Photons or combinations of sources can be either absorbed or reflected by the surface. In addition, different surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions. Most surfaces are somewhere in between.

OpenGL approximates light and lighting as if light can be broken into red, green and blue components. Thus, the colour of a light source is characterised by the amounts of red, green and blue light it emits, and the material of a surface is characterised by the percentages of incoming red, green and blue components that are reflected in various directions.

The OpenGL lighting model considers the lighting to be divided into four independent components: ambient, diffuse, specular, and emissive. All four components are computed independently and then added together.

Light sources have several properties, such as colour, position, and direction. The command used to specify all properties of light is **glLightfv()**. It takes three arguments: the identity of the light whose property is being specified, the property, and the desired value for that property. So we use the follow commands to specify these properties.

```
GLfloat light_position[]={1.0, 1.7, 3.0, 0.0};
GLfloat white_light[]={0.6, 0.6, 0.6, 0.6};
```

and

```
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
```

With OpenGL we need to enable (or disable) lighting explicitly. To enable lighting we need to explicitly to enable each light source that we define, after we have specified the parameters for that source. We use only one light, GL_LIGHT0:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

2.2 Defining Material Properties

Now that we have created light sources with certain characteristics we need to define the material properties of the objects in the scene: the ambient, diffuse, and specular colours, the shininess,

and the colour of any emitted light. Most of the material properties are conceptually similar to ones we have already used to create the light source. The mechanism for setting them is similar except that the command used called **glMaterial*()**.

The GL_DIFFUSE and GL_AMBIENT parameters set with **glMaterial*()** affect the colours of the diffuse and the ambient light reflected by an object. Diffuse reflectance plays the most important role in determining what you perceive the colour of an object to be. Your perception is affected by the colour of the incident diffuse light and the angle of the incident light relative to the normal direction.

The values used in these definitions are described below:

```
GLfloat mat_ambient1[]={0.0, 0.0, 0.9, 1.0};
GLfloat mat_ambient[]={0.9, 0.0, 0.0 1.0};
GLfloat mat_diffuse[]={ 0.5, 0.5, 0.4, 1.0 };
GLfloat mat_diffuse1[]={ 0.5, 0.3, 0.4, 1.0 };
GLfloat no_shininess[]={10.0};
GLfloat mat_specular[]={1.0 ,1.0, 1.0, 0.15};
GLfloat mat_shininess[]={40.0};
GLfloat mat_emission[]={ 0.7, 0.7, 0.7, 1.0 };
```

where `mat_ambient1[]`, `mat_diffuse1[]` are used to define ambient and diffuse values for the spheres which represent the particles. We use:

```
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION,mat_emission);
```

just to define the material properties for the one kind of spheres. For the other kind of spheres we use the `mat_ambient1[]`, `mat_diffuse1[]` instead of `mat_ambient[]` and `mat_diffuse[]` respectively.

There is also something else that we must describe concerning the material properties. This is the shading model. We specify the desired shading technique with **glShadeModel**. This command sets the shading model. The mode parameter can be either GL_SMOOTH (the default) or GL_FLAT. In this project we use:

```
glShadeModel(GL_SMOOTH);
```

2.3 Creating an Image

First of all, we read the data from the data file. So what we need is the radius **r[i]**, the x-coordinate **x[i]** and the y coordinate **y[i]** of each sphere.

After we have read the data file we render the spheres.

What we use to render a solid sphere is a GLUT function. The function is **glutSolidSphere**. So we produce an approximation to a solid sphere which is made up of a stack of ten layers slices each comprising ten slices. The shading model actually makes this rough approximation look surprisingly smooth.

```
glutSolidSphere(r[i], 10, 10);
```

This command renders a sphere at the origin, so we have to translate those spheres to their proper position. We can do that using the viewing transformation **glTranslate*()**. And so we manage to render the **i**th sphere to its proper position translating **x[i]** across the x-axis, **y[i]** across the y-axis and **0** across the z-axis. For example

```
g_x=(GLfloat)x[i]; g_y=(GLfloat)y[i];
glTranslatef(g_x,g_y, 0.0);
glutSolidSphere((GLfloat)r[i], 10, 10);
```

So we can create a function which will do this rendering called for example

```
void render_image( void );
```

This function does the work that we described above whenever it is called.

2.4 Viewing

A viewing transformation changes the position and orientation of the viewpoint. The viewing transformation positions the camera tripod, pointing the camera toward the model. In this project we use the Utility Library routine **gluLookAt()** to define a line of sight. We also use the perspective projection. The most unmistakable characteristic of perspective projection is foreshortening: the farther an object is from the camera, the smaller it appears in the final image. For this reason we use the **gluPerspective()** routine. Finally we use the bellow routine:

```
void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 200.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0, 1.0, 20.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

2.5 The Main Loop

Finally we create the main loop. First of all we use **glutInit()** to initialise GLUT. The we specify that we want a window with single buffering, the RGBA colour model, and the depth buffer using the

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

Furthermore we specify the screen location for the upper-left corner and the size, in pixels, of our window. And then we create our window and call all the callback routines.

```
glutInitWindowSize(SIZE, SIZE);
glutInitWindowPosition(10, 10);
glutCreateWindow(argv[0]);
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutKeyboardFunc(keyboard);
```

The very last thing we do is call **glutMainLoop()**. All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins, and the registered display callback is triggered.

2.6 Off-Screen Rendering

Every operation we have described up to now uses the frame buffer to actually display an image on the computer screen. That makes the creation of the images a very slow procedure. Not only does the image have to be created in memory but it must also be displayed using the X window system (on a Unix platform). Our aim is to produce a lot of images. If the amount of the images is extremely big then the parallelisation of the serial code would be a very good idea. But if we want to do such a job we want the image to be only in memory and not in the framebuffer; we do not want our parallel code to produce hundreds of windows! For these reasons we use the **Mesa** Off-Screen rendering interface included the **GL/osmesa.h** header file. Although the Mesa implementation is not heavily optimised, the ability to easily perform Off-Screen rendering made it extremely attractive for our purposes.

The Mesa Off-Screen rendering interface is an operating system and window system independent interface to Mesa which allows one to render images into a client-supplied buffer in main memory. Such images are manipulated or saved in whatever way the client wants.

So if we want to use the mesa library first we must create an RGBA-mode context. For this reason we define

```
OSMesaContext ctx;
```

and then we use the **OSMesaCreateContext** to create a new Off-screen Mesa rendering context setting the

```
ctx = OSMesaCreatContext( GL_RGBA, NULL );
```

Then we must allocate the proper image buffer. For this reason we use a

```
void *buffer;
```

and then we set

```
buffer = malloc( WIDTH * HEIGHT * 4 );
```

where WIDTH and HEIGHT is the actual size of our image. After that we bind an OSMesaContext to our image buffer and make the specified context the current one using the **OSMesaMakeCurrent** routine.

```
OSMesaMakeCurrent(ctx,buffer,GL_UNSIGNED_BYTE,WIDTH,HEIGHT);
```

After that we can save the image directly to a simple .ppm file and then convert other formats (eg a .gif file). At the end is quite important to free the image buffer and to destroy the context. We can do this using the commands

```
free( buffer );

OSMesaDestroyContext( ctx );
```

We save the image into a ppm file by following the routine

```
   if (argc>1) {
      /* write PPM file */
      FILE *f = fopen( argv[1], "w" );
      if (f) {
         int i, x, y;
         GLubyte *ptr = (GLubyte *) buffer;
#define BINARY 1
#if BINARY
         fprintf(f,"P6\n");
         fprintf(f,"# ppm-file created by %s\n",  argv[0]);
         fprintf(f,"%i %i\n", WIDTH,HEIGHT);
         fprintf(f,"255\n");
         fclose(f);
         f = fopen( argv[1], "ab" );  /* reopen in binary append mode */
         for (y=HEIGHT-1; y>=0; y--) {
            for (x=0; x<WIDTH; x++) {
               i = (y*WIDTH + x) * 4;
               fputc(ptr[i], f);   /* write red */
               fputc(ptr[i+1], f); /* write green */
               fputc(ptr[i+2], f); /* write blue */
            }
         }

#else /*ASCII*/
         int counter = 0;
         fprintf(f,"P3\n");
         fprintf(f,"# ascii ppm file created by %s\n", argv[0]);
         fprintf(f,"%i %i\n", WIDTH, HEIGHT);
```

```
         fprintf(f,"255\n");
         for (y=HEIGHT-1; y>=0; y--) {
            for (x=0; x<WIDTH; x++) {
               i = (y*WIDTH + x) * 4;
               fprintf(f, " %3d %3d %3d", ptr[i], ptr[i+1], ptr[i+2]);
               counter++;
               if (counter % 5 == 0)
                  fprintf(f, "\n");
            }
         }
#endif
         fclose(f)
```

As you can see we can save the image either as a binary or as an ASCII file with name what ever we want because we use the arg[1] argument. Actually it is better to save it as a binary file because it is smaller.

To convert a ppm file to a gif file we use the system command

```
ppmquant 256 < file.ppm | ppmtogif > image.gif
```

2.7 Producing set of images

Let's say now that we want to add some more functionality. For example, say that we want to create a "flight-path" and travel along this flight-path looking our image from different points of view. We manage to implement such a procedure by using the viewing transformations **glRotate*()** and **glTranslate*()**. As you might suspect, these routines transform an object (or coordinate system, if you are thinking of it in that way) by moving or rotating it. So we use the commands

```
glRotate(-0.6, 0.0, 1.0, 0.0);
glTranslatef(0.0, 0.0, 0.1);
render_image();
```

The first command is a rotation of -0.6 degrees about the y-axis, and the second command does not change the x and y coordinate but changes the z-coordinate about 0.1, and the last command renders the image. So if we want to create a lot of images where every image will contain the same object but rotated and translated we can use the above commands as many times as the amount of images that we want to create. If for example we put the above commands into a for loop, then the object of the image will be transformed every time in terms of the values of the functions.

3 Parallelisation of the Code

After the basic serial OpenGL visualiser was created the next aim was to parallelise it using MPI. The main goal of the parallelisation is to speed up the creation of the images. There are two different ways to do such a job. The first, which is the most simple one, is the trivial parallelisation where each process does the same work as all the others, and the other, which is a little more complicated, is the implementation of the master-slave model.

3.1 Trivial Parallelisation

First of all we create the datafiles which contain the properties of the particles for every timestep. So in the trivial parallelisation any process can read the data from a different datafile and then create an image for a given timestep.

This way is very simple and efficient. For example when we use 4 processes to produce 65 images it takes around 50 sec instead of 180 sec for the serial code.

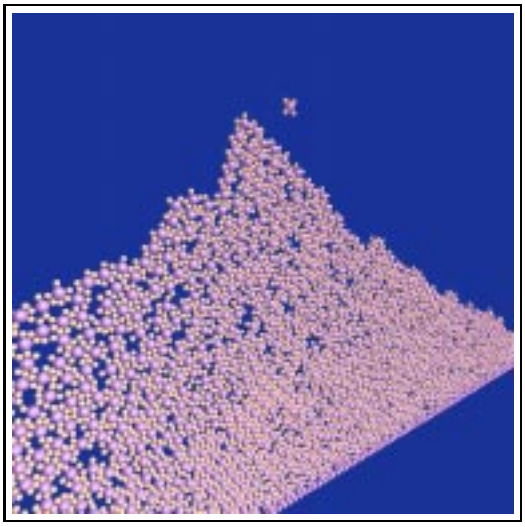
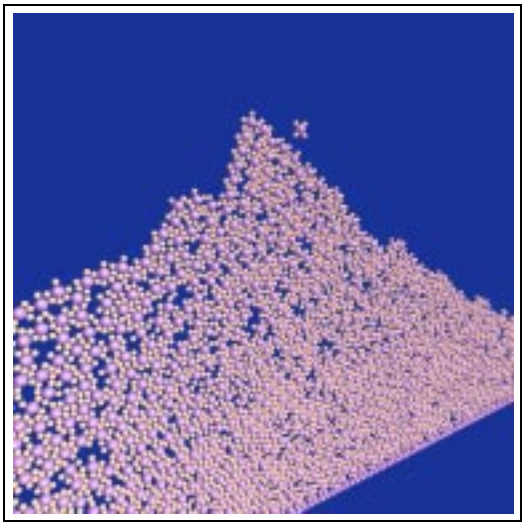
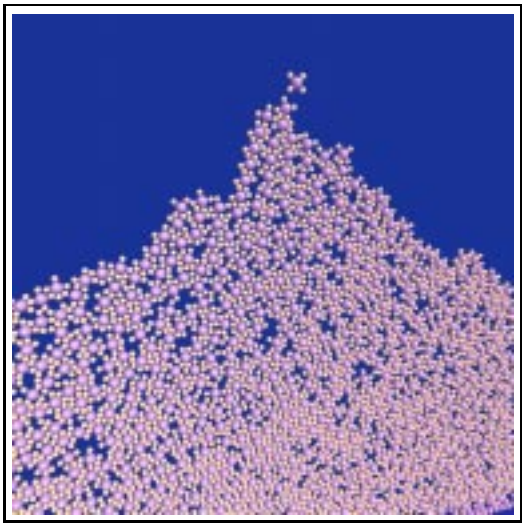
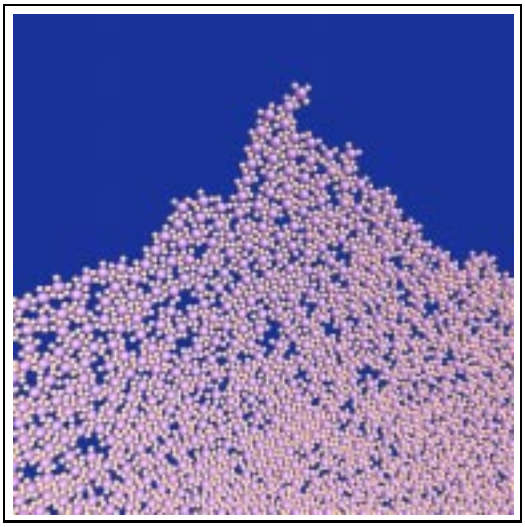
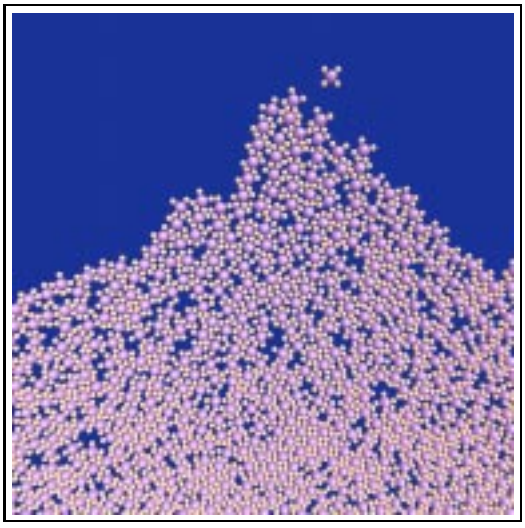
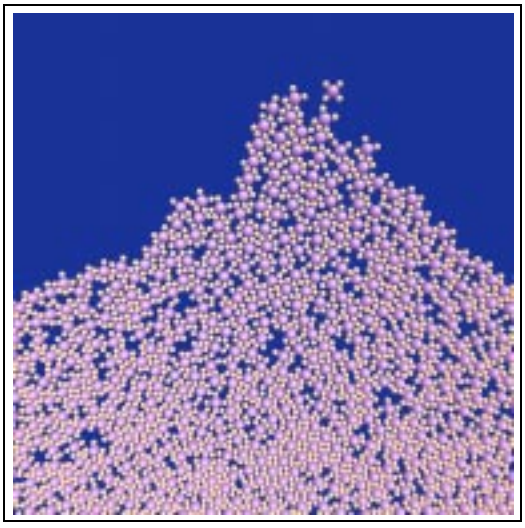
3.2 The Master-Slave Model

In the Master-Slave model there is a process which is called master and all the other processes are called slaves which do different work than the master does. The master reads the data from the data files, renders the image and then checks if there is any slave which is not doing any work. If the master finds any slave who is not working, he sends him work. The work is to produce a ppm file using the Mesa Off-Screen rendering interface and then to convert it into a gif file. When the master finishes with all the data files, he sends a message to the slaves that there is no more work to do.

This way of the parallelisation is faster than the serial implementation but it is not as fast as the trivial parallelisation. For example when we use 4 processes to produce 65 images it takes at around 120 sec. What we can achieve with the master slave model is to share equivalent work between the slaves. However, for our case the images are all of relatively equal complexity and the overhead of the master-slave model outweighs any load-balancing benefits.

4 Some results

Now using all the techniques that we describe above we produce many hundreds of images. This small example will convince you that OpenGL is a very powerful tool.

5 Producing a Video

We can produce simple animated gif files by using the `gifmerge` system command.

```
gifmerge *.gif > animated.gif
```

However, to produce a production quality video we must write an AVI file. We can do this job by using the Perception RT program (a Windows NT application). This also allows us to easily add other effects such as captions, fading between images etc. We use 24-bit TIFF files rather than 8-bit GIF files for high quality images. So first of all we create the TIFF files using the above OpenGL visualiser and a standard PPM to TIFF converter. Every TIFF image must have 720 Width and 576 Height. We can produce those TIFF files by using the system command

```
ppmquant 256 < file.ppm | pnmtotiff > image.tif
```

If we want to produce a CD rom with an AVI file we have to create TIFF files with 800 Width and 600 Height. After that we simply use the Perception RT package to produce the AVI file.

6 Conclusion

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications from modelling to scientific to games. OpenGL fosters innovation and speeds applications development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualisation functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring a wide application deployment. We have shown how it can be used to produce animations of a Discrete Element Model. The authors of the DEM had not previously seen such high-quality animations of their data, and were very interested in our results. The video produced in this project will be shown by Dr Henty at the Supercomputimg 2000 conference in Dallas to illustrate his talk on parallel DEMs [6].

References

[1] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, *OpenGL Programming Guide (Third Edition)The Official Guide to Learning OpenGL, Version 1.2*, Addison Wesley.

[2] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, *OpenGL Reference Manual,The Official Guide to Learning OpenGL, Version 1.2*, Addison Wesley.

[3] Mark J. Kilgard, *The OpenGL Utility Toolkit(GLUT) Programming Interface, API Version 3*, Silicon Graphics Inc, November 13, 1996.

[4] William F. Mitchell, *A Fortran90 Interface for OpenGL: Revised January 1998,NISTIR 6134* U.S. Department of Commerce, January 1998.

[5] R. W. Hockney, J. W. Eastwood, *Computer Simulation using Particles*, Adam Hilger.

[6] D.S. Henty, *Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling*, in proceedings of Supercomputing 2000, Nov. 2000 (to appear).

Dimitrios Mitsotakis is a Mathematician graduated from the Department of Mathematics at the University of Crete in Greece.
His areas of interest are Functional Analysis, Convex Bodies, Geometry of Numbers, Numerical Analysis, Scientific Computations and Code Theory.

This project was supervised by Dr. David Henty (EPCC), who is currently working with a DEM from the Department of Physics.