Visualizing Molecular Dynamics I—Windowing

Graphic Library

OpenGL

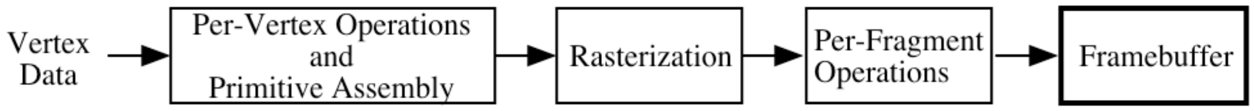
- **OpenGL**: A standard, hardware-independent interface to graphics hardware (e.g., SGI InfiniteReality2 Graphics Engine). A set of graphics routines defined on the OpenGL virtual graphics machine.
- **Windowing system**: To achieve portability, OpenGL does not include commands for windowing tasks (such as creating a window) or obtaining user input (such as a mouse click). Instead one has to work with whatever windowing system which controls the particular hardware being used. On SGI workstations, we can use either the OpenGL Extension to X window systems (**GLX**) or the OpenGL Utility Toolkit (**GLUT**), a window system-independent toolkit for window APIs. GLUT is also available on PC and Macintosh platforms.
- For information on OpenGL and software download, see `http://www.opengl.org`.

REFERENCE

- D. Shreiner, *OpenGL Programming Guide, 7th Ed.* (Addison-Wesley, Reading, MA, 2009).

GRAPHICS PIPELINE

- **Framebuffer**: A collection of buffers that store data for screen pixels (screen is, for example, 1280 pixels wide and 1024 pixels high) such as color, depth information for hidden surface removal, etc.
- **Rendering pipeline**: A typical graphics application creates a set of polygons, which must be converted into pixel information by various transformations. These transformations include: projection of the 3D world onto the screen and clipping. These transformations are pipelined by a special hardware called graphic pipeline.



> **Vertex data**: Our polygon data consist of vertices.

> **Per-vertex operations**: Translations and rotations are performed for some vertices. Positions in the 3D world are projected onto positions on the screen. Lighting calculations are performed using the vertices, surface normal, light source position, and material properties.

> **Primitive assembly**: Clipping eliminates portions of geometry, which fall outside the screen.

> **Rasterizations**: Conversion of geometric data into **fragments**. Each fragment square corresponding to a pixel in the framebuffer. Color and depth (z coordinate) values are assigned.

> **Per-fragment operations**: Hidden surface removal using the depth buffer (z buffer) or alpha blending for transparent materials.

- **OpenGL as a state machine**: You put it into various states (such as the current color and the current model transformation matrix), then the states remain in effect until you change them.

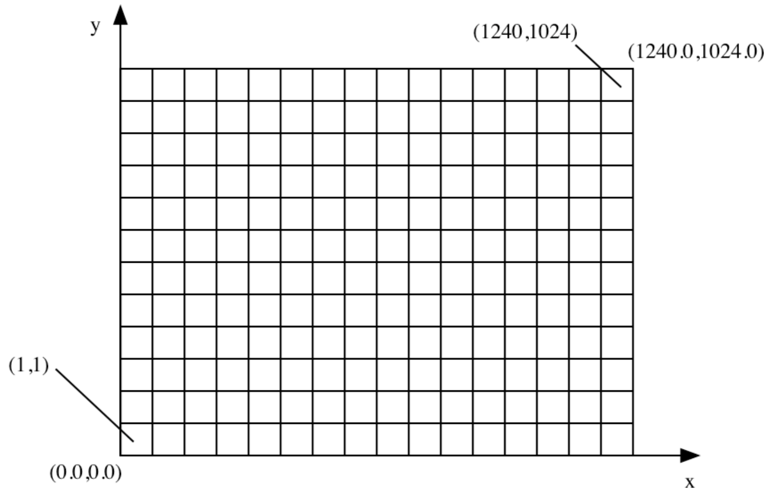
USING OpenGL

- To use any one of the GL-library calls (every GL routine has a prefix `gl`, e.g., `glColor3f`), include the following line in your program:
  `#include <OpenGL/gl.h>`
- **The OpenGL Utility Library (GLU)**: Contains several routines that use low-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations (GLU routines use the prefix `glu`, e.g., `gluLookat`). To use a GLU-library call, include:
  `#include <OpenGL/glu.h>`
- **The OpenGL Utility Toolkit (GLUT)**: A window-system-independent toolkit for windowing, written by Mark Kilgard (GLUT routines use the prefix `glut`, e.g., `glutCreateWindow`). You can download GLUT also from `www.opengl.org`. To use a GLUT-library call, include:
  `#include <GLUT/glut.h>`
- Compiling an OpenGL program is platform-specific and will be explained in the lecture.

WINDOW MANAGEMENT

Initializing a Window

- `void glutInitDisplayMode(unsigned int mask);`
  Specifies a display mode (such as RGBA or color-index, or single- or double-buffered—to be explained later) for windows created when `glutCreateWindow()` is called. The mask argument is a bitwise ORed combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer-enabling flags such as GLUT_DEPTH, e.g., `glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE).`
- `void glutInitWindowsize(int width, int height);`
  Requests windows created by `glutCreateWindow()` to have an initial horizontal (x) and vertical (y) size, `width` and `height` pixels, e.g., `glutInitWindowsize(640,480)`. The pixels are indexed as follows:



- `void glutCreateWindow(char *name);`
  Opens a window with previously set characteristics (display mode, sizes, etc.). The string `name` will appear in the title bar, e.g., `glutCreateWindow("Lennard-Jones MD")`. (The window is not actually displayed until `glutMainLoop()` is called.)

Event-Handling Loop

- The GLUT system uses event-handing-type programming style. You will provide event handlers for specific events. You will write functions that specify, e.g., what to do when the mouse is clicked, or what to draw when the GLUT run-time system decides to draw something on a window. The GLUT system will endlessly repeat an event loop, waiting for an event to occur, and when an event occurs, calls the corresponding event-handling routine you have provided.
- `void glutDisplayFunc(void (*func)(void));`
  Specifies the function that is called whenever the contents of the window need to be redrawn—when the window is initially opened, when the window is popped and window damage is exposed, etc. For example, if you write a drawing program in
  ```
  void display() {…};
  ```
  then you register this function as a callback function as
  ```
  glutDisplayFunc(display);
  ```
- `void glutMainLoop(void);`
  Enters the GLUT "event-processing loop". Registered callback functions will be called when the corresponding events occur.

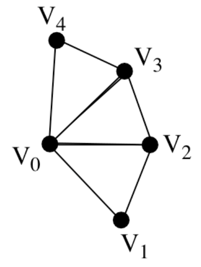
Visualizing Molecular Dynamics II—Static Visualization

In the following, the program `atomv.c` is used as a sample application. The program visualizes MD atoms as spheres.

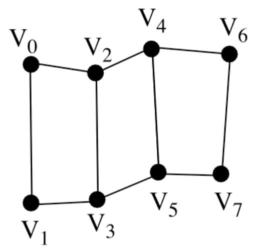
Polygonal Surfaces

OpenGL GEOMETRIC DRAWING PRIMITIVES

We construct a scene as a set of polygons. To create a polygonal surface from a set of vertices, bracket the set of vertices between a call to `glBegin()` and `glEnd()`. The argument passed to `glBegin()` determines what sort of geometric primitive is constructed from the vertices.



GL_TRIANGLE_FAN



GL_QUAD_STRIP

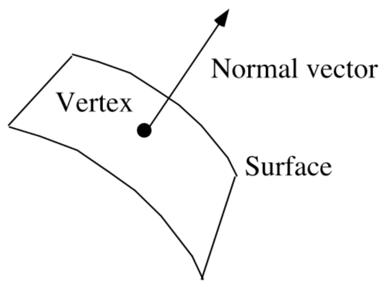
- `void glBegin(GLenum mode);`
  Marks the beginning of a vertex-data list that describes a geometric primitive. The type of primitive is indicated by mode, a few examples of which are given in the table below.

Value	Meaning
GL_TRIANGLE_FAN	Linked fan of triangles
GL_QUAD_STRIP	Linked strip of quadrilaterals

- `void glEnd(void);`
  Marks the end of a vertex-data list.

(Example)
```
float normal_vector[MAX_VERTICES][3],vertex_position[MAX_VERTICES][3];
glBegin(GL_QUAD_STRIP);
  for (i=0; i<number_of_vertices; i++) {
    glNormal3f(normal_vector[i]);
    glVertex3f(vertex_position[i]);
  }
glEnd();
```

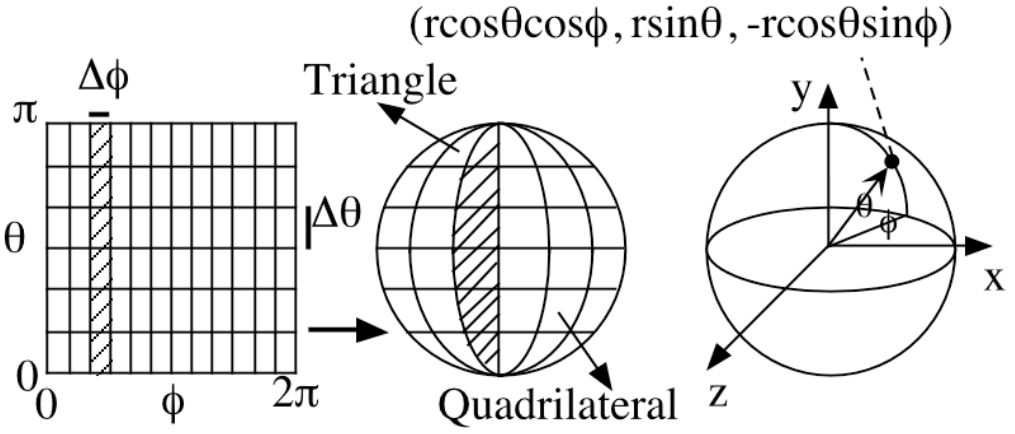
The quadrilateral-mesh-construct creates a surface from a sequence of vertices. The first 4 vertices in the sequence is used to draw a first quadrilateral strip; each time two new vertices are encountered, the system draws a quadrilateral strip containing them and the last two vertices before them.



- `glVertex3f(float x, float y, float z);`
  Specifies a vertex for use in describing a geometric object. (x, y, z) is the 3D coordinate of the vertex.
- **Normal vector**: Points outward perpendicular to the surface at the current vertex position. The normal information is used for implementing lighting effects.
- `glNormal3f(float nx, float ny, float nz);`
  Sets the current normal vector as specified by the arguments.

POLYGONAL SPHERE

In `atomv.c`, a unit sphere is represented by `nlon` (longitudinal direction) by `nlat` (lateral direction) quadrilateral strips. Since all lateral points at the south pole (also at the north pole) are degenerated, we use a linked fan of 3 triangles each at the south- and north-poles, instead.



The following code in `makeFastNiceSphere()` defines the vertices (and normals in this case are parallel to the vertices) on a spherical surface with a given `radius`. Note that `M_PI` (= π = 3.14159…) is defined in the math library. The radius in Lennard-Jones unit is defined as `float atom_radius = 0.5` in `atomv.h` and is passed to `makeFastNiceSphere()` as an argument.

```
int nlon=3, nlat=2;
loninc = 2*M_PI/nlon; /* Δϕ */
latinc = M_PI/nlat;   /* Δθ */

/* South-pole triangular fan */
glBegin(GL_TRIANGLE_FAN);
  glNormal3f(0,-1,0);
  glVertex3f(0,-radius,0);
  lon = 0;
  lat = -M_PI/2 + latinc;
  y = sin(lat);
  for (i=0; i<=nlon; i++) {
    x = cos(lon)*cos(lat);
    z = -sin(lon)*cos(lat);
    glNormal3f(x,y,z);
    glVertex3f(x*radius,y*radius,z*radius);
    lon += loninc;
  }
glEnd();
```

```
/* Quadrilateral strips to cover the sphere */
for (j=1; j<nlat-1; j++) {
  lon = 0;
  glBegin(GL_QUAD_STRIP);
    for (i=0; i<=nlon; i++) {
      x = cos(lon)*cos(lat);
      y = sin(lat);
      z = -sin(lon)*cos(lat);
      glNormal3f(x,y,z);
      glVertex3f(x*radius,y*radius,z*radius);
      x = cos(lon)*cos(lat+latinc);
      y = sin(lat+latinc);
      z = -sin(lon)*cos(lat+latinc);
      glNormal3f(x,y,z);
      glVertex3f(x*radius,y*radius,z*radius);
      lon += loninc;
    }
  glEnd();
  lat += latinc;
}
/* North-pole triangular fan */
glBegin(GL_TRIANGLE_FAN);
  glNormal3f(0,1,0);
  glVertex3f(0,radius,0);
  y = sin(lat);
  lon = 0;
  for (i=0; i<=nlon; i++) {
    x = cos(lon)*cos(lat);
    z = -sin(lon)*cos(lat);
    glNormal3f(x,y,z);
    glVertex3f(x*radius,y*radius,z*radius);
    lon += loninc;
  }
glEnd();
```

Display Lists

When we draw a number of atoms in an MD configuration, we repeatedly reuse the sphere-drawing operations many times. OpenGL allows us to store this set of operations as a "display list" and invoke it anytime it is needed.

- **Display list**: A group of OpenGL commands that have been stored for later execution. When a display list is invoked, the commands in it are executed in the order in which they were issued. Display lists may improve performance by caching commands, which are reused many times. (Some graphics hardware may store display lists in dedicated memory.)

  Each display list is identified as a unique, system-generated integer ID. For example

```
GLuint sphereid = glGenLists(1);
```

generates one new display-list ID and store it in variable `sphereid`.

- `GLuint glGenLists(GLsizei range);`
  Allocates `range` number of contiguous, previously unallocated display list indices. The integer returned is the index that marks the beginning of a contiguous block of empty display list indices.

Next, a routine for drawing a sphere (the code segment shown above) is bracketed by `glNewList()` and `glEndList()`.

```
glNewList(sphereid, GL_COMPILE);
  (the above code to draw a sphere)
glEndList();
```

- `void glNewList(GLuint list, GLenum mode);`
  Specifies the start of a display list. `list` is a nonzero positive integer that uniquely identifies the display list. The possible values for `mode` are GL_COMPILE and GL_COMPILE_EXECUTE. Use GL_COMPILE if you don't want OpenGL commands executed as they are placed in the display list.

- `void glEndList(void);`
  Marks the end of a display list.

  To execute a display list, which has been created previously,
  `glCallList(sphereid);`

- `void glCallList(GLuint list);`
  Executes the display list specified by `list`.

Modeling Transformation

MODELING TRANSFORMATION FUNCTIONS

- **Instance**: In MD visualization, a series of spheres must be drawn at different positions. We can reuse the sphere list, which stores a single-sphere-drawing routine defined in `makeFastNiceSphere()` for drawing spheres at various positions. Each sphere is an instance of a generic sphere class. In order to do this, graphical objects must be manipulated using modeling transformation function, `glTranslatef()`.

- `void glTranslatef(float x, float y, float z);`
  Moves the object origin to the point specified in the current object coordinate system. The object coordinate system is a state of the graphic machine; all objects drawn after `glTranslatef()` execution will be translated. Multiplies the current matrix (a state variable) by a matrix that moves an object by given x-, y-, and z-values.

- `void glRotatef(float angle, float x, float y, float z);`
  Rotates graphical objects by `angle` (in degrees) around the rotation axis specified by vector (`x`, `y`, `z`). As you look down the positive axis to the origin, a positive rotation is counterclockwise.

- `void glScalef(float x, float y, float z);`
  Shrinks and expands objects by specified magnifying factors in the x, y and z directions. All objects drawn after `glScalef` execution will be scaled.

TRANSFORMATION MATRIX

- **Transformation matrix**: Specifies the amount by which the object's coordinate system is to be rotated, scaled, or translated. Each time you specify a transformation such as rotate or translate, the software automatically generates a transformation matrix for the rotation or translation. The current transformation matrix (which is a state variable) is then premultiplied by the generating matrix, effecting the desired transformation.

- **Affine transformation**: Rotation, scaling and translation are represented as a single affine transformation, $\vec{r}' = \overleftrightarrow{A}\vec{r} + \vec{b}$. The affine transformation is represented by a 4 by 4 matrix:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- **Matrix stack**: The graphics system maintains a stack of transformation matrices. At the top of the stack is the current transformation matrix. Initially the transformation matrix is the identity matrix (do nothing):
  ```
  Matrix Identity = {1, 0, 0, 0,
                     0, 1, 0, 0,
                     0, 0, 1, 0,
                     0, 0, 0, 1};
  ```
- `void glPushMatrix(void);`
  Pushes all matrices in the current stack down one level. The topmost matrix (current transformation matrix) is copied, so its contents are duplicated in both the top and second-from-the-top matrix.
- `void glPopMatrix(void);`
  Eliminates the matrix on the top (destroying the contents of the popped matrix) to expose the second-from-the-top matrix in the stack.

DRAWING MULTIPLE ATOMS

The function `glTranslatef()` is used to draw all the atoms in an MD configuration in `makeAtoms()`. (In fact `makeAtoms()` merely makes a display list `atomsid` to be drawn later in `drawScene()`.

Data Structures in Program `atomv.c`

- `int natoms`: Number of atoms

-
  ```
  typedef struct {
    float crd[3];
  } AtomType;
  ```
  Atom data type. `crd[0|1|2]` is the x|y|z coordinate of the atom in the Lennard-Jones unit.
- `AtomType *atoms;`
  Array of atoms (dynamically allocated).
- `float min_ext[3], max_ext[3];`
  [`min_ext[0|1|2]`,`max_ext[0|1|2]`] is the range of x|y|z coordinate of atoms. `max_ext[0|1|2]` is basically the MD-box length in the x|y|z direction.

The atomic coordinates are input from a file in `readConf()`:

```
fscanf(fp,"%d",&natoms);
atoms = (AtomType *) malloc(sizeof(AtomType)*natoms);
for (l=0; l<3; l++) fscanf(fp,"%f%f",&min_ext[l],&max_ext[l]);
for (j=0; j<natoms; j++)
  fscanf(fp,"%f %f %f",&(atoms[j].crd[0]),&(atoms[j].crd[1]),
                       &(atoms[j].crd[2]));
```

An input file can be prepared by inserting the following lines in the `md.c` program.

```
double min_ext = 0.0;
fprintf(fconf,"%d\n",nAtom);
for (l=0; l<3; l++)
   fprintf(fconf,"%f %f\n",min_ext,Region[l]);
for (j=0; j<nAtom; j++)
   fprintf(fconf,"%f %f %f\n",r[j][0],r[j][1],r[j][2]);
```

The following code draws a series of MD atom as scaled and translated instances of the unit-sphere object. Note that the transformation matrix is a state variable. If we apply scale and translation operations for one atom, the effect will remain for the next atom. Since the next atom has its own scale and translation, the previous transformation must be eliminated.

```
for (i=0; i<natoms; i++) {
  glPushMatrix();
  glTranslatef(atoms[i].crd[0],atoms[i].crd[1],atoms[i].crd[2]);
  glCallList(sphereid);
  glPopMatrix();
}
```

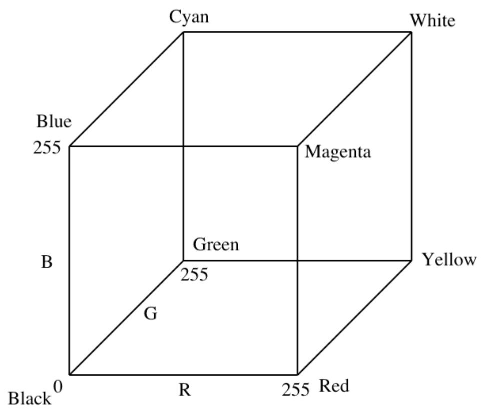
* Note that `sphereid` routine draw a sphere around the origin, (0, 0, 0). By changing the current transformation matrix by `glTranslatef()`, the next call to `sphereid` will draw a sphere around (`atoms[i].crd[0],atoms[i].crd[1],atoms[i].crd[2]`) instead.

Lighting and Materials

The Graphics Library lighting model determines how the incident light is modified when it reflects from objects in the scene, thereby giving a sense of reality.

COLOR DISPLAY

The standard monitor has three color guns, which sweep out the entire screen area 60 times per second. Each pixel on the screen is composed of three different phosphors that glow red, green, or blue. Each color gun activates only one of the phosphors. Each color gun can be set to 256 different intensity levels, ranging from completely off to completely on. The intensities of the red, green and blue guns at the pixel determine its color.



- **RGB(A) mode**: One way to specify color is to provide 8 bits (between 0 and 255) each for red, green and blue intensities. Some function instead specifies normalized intensity in the range [0.0,1.0]. The fourth color component (Alpha) is optional, see below.
- **Alpha component**: Specifies the opacity of a material. The highest alpha value (int 255 or float 1.0) of an object means that you cannot see other objects behind it. An object with the lower alpha value (0 or 0.0) is completely transparent and invisible.
- **Color functions**: Color state variable is specified by various color functions. Everything after a color function call will be drawn with the color until the color-state is changed by another color function call. (Note that color values are state variables.) An example of color functions is:

  ```
  glColor3f(1.0, 0.0, 0.0);        /* Red color */
  ```

  Note the three vector components are arranged from left to right: red, green, and blue. If an alpha value is not supplied, it is automatically set to 1.0.

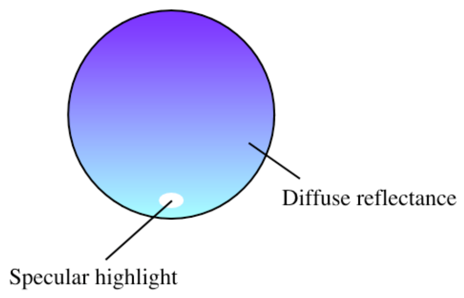
- `void glColor3f(float r, float g, float b)`
  Sets the current red, green, and blue values.
- `glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH);`
  The second bit mask, `GLUT_RGBA`, sets the display mode to RGBA mode.

LIGHT MODEL

OpenGL defines a few different types of light.

- **Diffuse component**: Gives the appearance of a matter or flat reflection from an object's surface. The direction of the light as it falls on the surface determines how bright the surface's diffuse reflectance is. Diffuse reflection is brightest when the incident light strikes perpendicular to the surface. Once it hits a surface, however, it is scattered equally in all directions so that it appears equally bright no matter where the eye is located.

- **Ambient illumination**: Simulates light reflected from other objects in the scene, rather than directly from the light source. It has been scattered so much by the environment that its direction is impossible to determine—it seems to come from all directions. (We can still see things under a desk, though it is not illuminated directly by a light.)
- **Specular light**: Creates highlights. It is like the glare in a rearview mirror from the headlights of a car behind you. (The best example is a well-collimated laser beam bouncing off a high-quality mirror.)



- **Emission**: Material may have an emitted color component. This is used only for simulating the appearance of lights in the scene.

MATERIAL DEFINITION

- **Refelectance**: The material characteristics of an object determine how the object reflects light. Material is characterized by ambient, diffuse, specular reflectance (in addition emissive color if it itself is a light.) In addition, alpha value specifies material's opacity. For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light that strikes it. Material's ambient reflectance is combined with the ambient component of each incoming light source, etc. to define material properties.
- **Color material mode**: To define material properties, `glMaterialf()` function can be used. Alternatively, color material mode may be activated as
  ```
  glEnable(GL_COLOR_MATERIAL);
  ```
  In this mode, the current color specified by `glColor*()` will change the ambient and diffuse reflectance of materials

LIGHTING SOURCE

Light Source Definition

We can have up to 8 light sources that illuminate the scene. A light source is defined in terms of a set of properties such as ambient, diffuse, and specular light components as well as position. These properties are set by using the `glLightfv()` function.

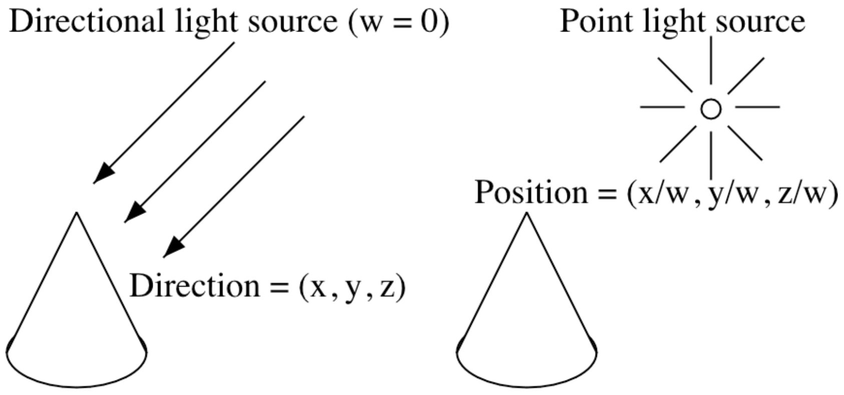
- `void glLightfv(GLenum light, GLenum pname, float *param);`
  Creates the light specified by `light`, which can be GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT_7. The characteristic of the light being set is defined by `pname`, which is specified by a named parameter, some of which are listed in the table below. `param` is an array containing the values to be set.

Parameter Name	Default Values	Meaning
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) for light 0	diffuse intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light

(Example)
```
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);   light_diffuse[4] = {1.0,1.0,1.0,1.0}
glLightfv(GL_LIGHT0, GL_POSITION, light_position); light_position[4] = {0.5,0.5,1.0,0.0}
```

Directional and Point Light Sources

If w of the light position is 0.0, then the light source is a directional one and the (x, y, z) values describe its direction. Else, it is a point light source: the (x, y, z) vector represents the position of an omnidirectional point light source, and its position is (x/w, y/w, z/w).



Enabling Lighting

You need to enable lighting explicitly. Otherwise, the current color is simply mapped onto the current vertex, and no calculations concerning normals, light sources, or materials are performed. To enable lighting:

```
glEnable(GL_LIGHTING);
```

You also need to enable each light source you define:

```
glEnable(GL_LIGHT0);
```

HIDDEN SURFACE REMOVAL

- **Hidden surface removal**: Draws only those surfaces that are nearest to the eye for opaque objects. All other surfaces are obscured by those nearer the eye. By default, the OpenGL system does not do hidden surface removal. Figures are rendered on the screen in the order they are drawn.
- **Z buffer**: Keeps track the z coordinate (distance from the eye) of the nearest surface for each pixel.
- `glEnable(GL_DEPTH_TEST);`
  Enables z-buffer.

Clearing a Window

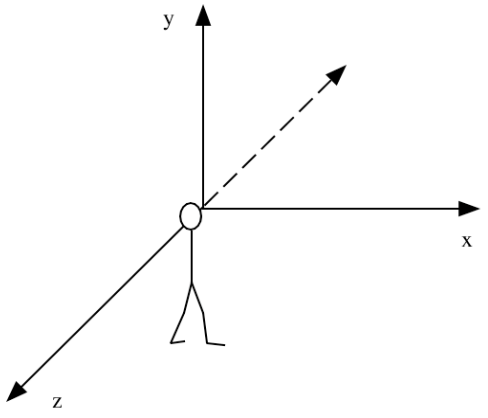
Before drawing a scene, the color buffer and the z-buffer must be initialized. This is, for example, necessary when the view angle changes, since the surfaces that are hidden will then change.

- `void glClear(Glbitfield mask);`
  Clears the specified buffers to their current clearing values. The mask argument is a bitwise OR combination of the values in the table below, e.g, `glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT)`.

Buffer	Name	Default values
Color buffer	GL_COLOR_BUFFER_BIT	(0, 0, 0, 0), i.e., black
Depth buffer	GL_DEPTH_BUFFER_BIT	1.0, i.e., the maximum value in [0, 1]

Viewing Transformation

In addition to the 2-dimensional window-coordinate system, OpenGL understands 3D coordinates through the **eye coordinate system**. The viewer is at the origin, and is looking toward the negative z axis. The viewer's "up" direction is (001).

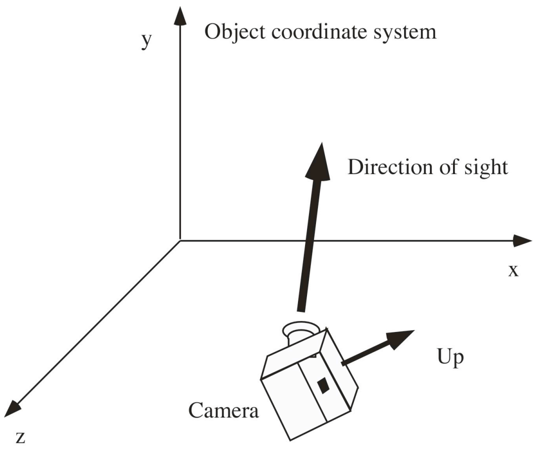


The transformation from the eye coordinate system to the window coordinate system will be described later. We first describe how our MD box, $0 \le x < L_x$, $0 \le y < L_y$, $0 \le z < L_z$, in the `md.c` program is transformed to the eye coordinate system. In the OpenGL terminology, the original coordinate system prior to any OpenGL transformation is called the **object coordinate system**.

VIEWING TRANSFORMATION

OpenGL achieves viewing transformation by setting a transformation matrix, corresponding to the specified viewer information by the `gluLookat()` routine.

- **Viewing transformation:** OpenGL has utility functions to help transform object coordinates to eye coordinates. One of them is `gluLookat()`, which transforms object coordinates relative to the viewer's eye position.
- `void gluLookat(eyx, eyey, eyz, centerx, centery, centerz, upx, upy, upz);`
  A GLU routine to specify the viewer (or camera) position. The first three arguments specify where the camera is. The next three arguments specify where the camera is aimed at (usually the center of the scene). Finally the last three argument specify which direction is up. By default (if `gluLookat()` is not called), the camera is at the origin, (0, 0, 0), points down to the negative z axis, and has an up-vector of (0, 1, 0). All coordinates are defined as type `GLdouble`.



In `atomv.c`, we have chosen to look at the MD system from the positive z direction and at the x-y position, which is the center of the MD box. The distance between the eye position and the center of the MD box is chosen to be the diagonal length of the MD box:

$$\texttt{dis = sqrt}(L_x^2 + L_y^2 + L_z^2).$$

i.e., $(eyx, eye, eyz) = (centerx, centery, centerz + \sqrt{L_x^2 + L_y^2 + L_z^2})$ and $(upx, upy, upz) = (0, 1, 0)$.

Projection Transformation

Specifying projection transformation is like choosing a lens for a camera (the position and direction of which has been specified by a call to the `gluLookat()` function). For example, either a wide-angle or normal-angle camera can be used to choose a view angle. Any object that lies outside the view angle then needs to be "clipped" away.

$$\text{object coordinates} \xrightarrow[\text{model-view transformation}]{} \text{eye coordinates} \xrightarrow[\text{projection transformation}]{} \text{2D screen coordinates}$$

Since the projection-transformation matrix is a state variable, and projection transformation functions work on the current transformation matrix (i.e., by premutiplying the matrix to achieve the specified transformation to the current state variable), you need to initialize it to an identity matrix before specifying a projection-transformation matrix. This is done by calling `glLoadIdentity()` before calling `gluPerspective()`.

- `void glLoadIdentity(void);`
  Sets the current matrix to the identity matrix (do nothing) as defined previously.

CLIPPING

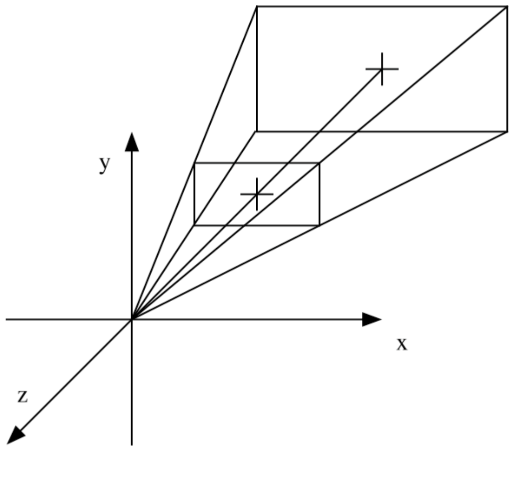
Viewing items on the computer screen is like looking through a rectangular piece of transparent glass at the items.

The collection of all the lines leaving your eye and passing through the glass would form an infinite four-sided pyramid with its apex at your eye. Anything outside the pyramid would not appear on the glass, so the four planes passing through your eye and the edges of the glass would clip out invisible items.

In addition to these left, right, bottom and top clipping planes, the graphic system also provides two other clipping planes that eliminate anything two far or too near the eye. Thus the visible region looks like an Egyptian pyramid with the top sliced off, called **frustum**.

- `void gluPerspective(fovy,aspect,near_clip,far_clip);`
  Defines a projection matrix that maps a frustum of eye coordinates so that it exactly fills the unit cube. The four arguments are: the field of view in the y direction (in degrees); the aspect ratio (the x dimension—width—of the glass divided by its y dimension—height); and the distances to the near and far clipping planes. All arguments are of type GLdouble.

Since the items in the frustum will be shown in a window, the aspect ratio is usually that of the window. In order to obtain the window-size information (note that the window size specified in the `glutInitWindowSize()` is merely a suggestion, and the actual window size determined by the operating system may be different), we place the perspective operation in the callback function to the window-resize event.



- `void glutReshapeFunc(void (*func)(int width, int height));`
  Specifies the function that is called whenever the window is resized or moved. The argument `func` is a pointer to a function that expects two arguments, the new width and height of the window in pixels.

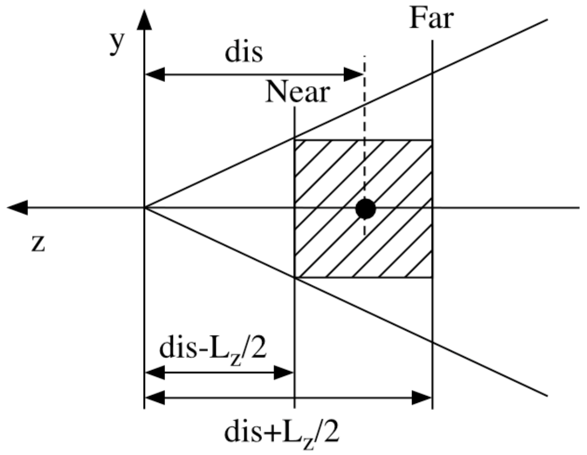
In our callback function, `reshape()`, we first define a 2D space onto which the frustum is projected before `gluPerspective()` is called.

- `void glViewPort(GLint x, GLint y, GLsizei width, GLsizei height);`
  Defines a pixel rectangle in the window in which the final image is mapped. The (`x`, `y`) parameters specify the lower left corner of the viewport, and `width` and `height` are the size of the viewport rectangle. By default, the initial viewport values are (0, 0, winWidth, winHeight), where winWidth and winHeight specify the size of the window.

The aspect ratio of the viewport should generally equal the aspect ratio of the viewing volume. The latter is specified in `gluPerspective()`. In `gluPerspective()`, we use the width and height of the current window, which are passed to the `reshape()` function by the GLUT system, to set aspect = width/height.

The following code defines the other parameters to specify a frustum that contains our normalized MD system.

```
near_clip = dis-0.5*dif_ext[2];
far_clip  = dis+0.5*dif_ext[2]);
fovy = 0.5*dif_ext[1]/(dis-0.5*dif_ext[2]);
fovy = 2*atan(fovy)/M_PI*180.0;
```



Note that the viewing angle = 2*tan^{-1}(L_z/(dis-L_z/2)). In the actual code, we give some margin so that the objects do not completely fill the screen: `fovy = (GLdouble)(1.2*fovy)`.

- **Current matrix mode**: The graphics system maintains the model-view matrix, which transforms coordinates from object coordinates to eye coordinates (i.e., which does modeling and viewing transformations). This is one of three matrices the graphics system maintains; the others are the projection matrix (which does projection transformation) and the texture matrix.

Before setting the projection matrix with `gluPerspective()`, we need to switch to the projection matrix mode so that the matrix stack holds the current projection matrix (and save the model-view matrix somewhere else), initializes the projection matrix as the unit matrix. After setting the projection matrix, we need to switch back to the model-view matrix mode.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fovy,aspect,near_clip,far_clip);
glMatrixMode(GL_MODELVIEW);
```

- `void glMatrixMode(GLenum mode);`
  Specifies which of the three matrices is the current matrix: ModelView (`GL_MODELVIEW`), Projection (`GL_PROJECTION`), or Texture (`GL_TEXTURE`) for `mode`. By default, the model-view matrix is the one that can be modified.

Visualizing Molecular Dynamics III—Animation

Animation

DOUBLE BUFFER MODE

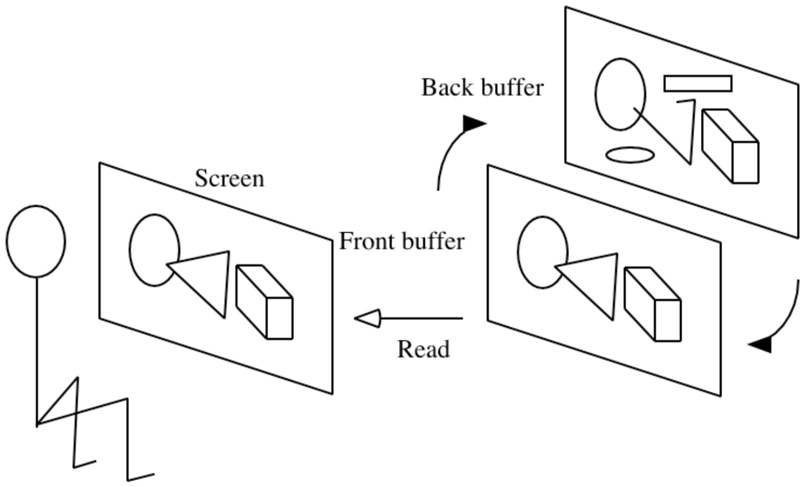
In double-buffer mode, two bitplanes (front and back planes) are maintained. Only the front bitplane is displayed, and drawing routines update only the back plane. After the drawing is completed, the front and back planes are swapped to expose the new scene.

Double buffer mode is used for smooth animation. With double buffering, the new scene replaces the previous one instantaneously; otherwise, rendering process would be seen on the screen.

- `glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);`
  The first bit mask, `GLUT_DOUBLE`, sets the display mode to double buffer mode.

On a standard monitor, an electron gun sweeps from the top of the screen to the bottom, refreshing all pixels 60 times each second. If the content of the visible frame buffer is changed, the next time the refresh hardware reads a changed pixel, the system draws the new value instead of the old one.

- `void glutSwapBuffers(void);`
  Exchanges the front and back buffers in double buffer mode during the next vertical retrace.



BACKGROUND PROCESS MANAGEMENT

For an MD animation within the GLUT event-loop model, we update atomic coordinates by integrating Newton's equation for one time step while no event is being processed. This is achieved by registering the MD single-step-update function as a handler to the "nothing-to-process event". After updating atomic coordinates, we then need to force the system to redraw the scene.

- `void glutIdleFunc(void (*func)(void));`
  Specifies the function, func, to be executed if no other events are pending.

- `void glutPostRedisplay(void);`
  Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by `glutDisplayFunc()` will be called.

Combining md.c and atomv.c

We will use atomv.c as a starting point, and add functions in md.c in appropreate places. First of all, there are some discrepancies in data structures between the two programs, and this must be unified.

atomv.c	md.c
`int natoms;` (number of atoms)	`int nAtom;`
`AtomType *atoms;` (atomic coordinates)	`double r[NMAX][3];`

1. Initialize atomic coordinates, velocities, accelerations, and step count before entering the GLUT main loop in `main()`.

```
   InitParams();   /* Read and initialize MD parameters */
   InitConf();
   ComputeAccel(); /* Compute initial accelerations */
   stepCount = 1;  /* Initialize the MD step count */
```

2. Register the single-MD-step update function, say `animate()`, as `glutIdleFunc()` callback function.

```
   glutIdleFunc(animate);
```

3. In the callback function, `animate()`:

```
   if the stepCount ≤ StepLimit
       (i) velocity-Verlet integration for one MD step, SingleStep() followed by
           ApplyBoundaryCond();
       (ii) make a display list for a collection of atoms with the updated atomic coordinates, i.e., call
           makeCurframeGeom();
       (iii)   call glutPostRedisplay() to redraw the updated scene;
       (iv) increment the time step, stepCount;
   endif
```