A Developer's Guide to the GLUT for Markov Chain Monte Carlo Graphical Interface Version 1.0

Eric C. Anderson*

October 23, 2002

Abstract

The GLUT for Markov chain Monte Carlo (Gf(MC)^2) library provides a system for visualizing the variables involved in Markov chain Monte Carlo. It is designed to provide a relatively simple user interface that allows the management of multiple windows which provide different views of the variables involved in the simulation. It is written in C. The window management system uses calls to Mark Kilgaard's OpenGL Utility Toolkit (GLUT). GLUT is almost platform independent, so the features of Gf(MC)^2 should work in almost the same way across different platforms (*e.g.*, Macintosh, Windows, Linux, *etc.*).

This document explains how a developer can use the Gf(MC)^2 interface to display real-time images that describe the states of variables in a Monte Carlo simulation. A separate document, *A User's Guide to the GLUT for Markov Chain Monte Carlo Graphical Interface*, describes how the end-user of a Gf(MC)^2-based program uses the features available in Gf(MC)^2. It is recommended that developers read that document first to familiarize themselves with the features of Gf(MC)^2. It may also be helpful to read Mark Kilgaard's GLUT documentation available at

`http://www.opengl.org/developers/documentation/glut/`

Contents

*Department of Integrative Biology, University of California, Berkeley, `eriq@u.washington.edu`

1 Incorporating Gf(MC)^2 into an MCMC program

Incorporating Gf(MC)^2 into one's program is usually an easy task if the program has been written in a fairly modular way. There are six functions that the developer must define to make Gf(MC)^2 interface with his program. These functions are:

1. `gfmUserDefd_OneStepByGlobals()`
2. `gfmUserDefd_InitializeChain()`
3. `gfmUserDefd_ResetAllAverages()`
4. `gfmUserDefd_DefineWindows()`
5. `gfmUserDefd_DefineMenus()`
6. `gfmUserDefd_LastWords()`

Functions 1–3 are involved in getting the GLUT event loop to drive the MCMC simulation, to allow restarting of the Markov chain, and to allow resetting of the Monte Carlo averages that are being accumulated, respectively. These functions are described in Section 2, where you will also find a discussion on the necessary changes to function `main()` that allow Gf(MC)^2 to work. Function 4 is the function in which the developer will define the particular windows that will be available. This is described in Section 3. Functions 5 and 6 provide a place for the experienced GLUT user to add specialized menus or add any final code before the GLUT main loop is entered. They are only described briefly in Section 5.

In addition to defining the six functions listed above, the developer must also write the code that instructs the system on what to actually draw in the windows. These "Drawing Functions" should be implemented in the C (or C++) binding of OpenGL. Section 4 describes some very simple OpenGL commands, and then describes how to access the settings maintained by Gf(MC)^2 for each window to incorporate those settings (affecting the color, viewable area, *etc.*) into the material that gets drawn in each window.

Note that the components which are dependent on both Gf(MC)^2 and the developer's MCMC program can all be maintained in a single source code file.

2 Running the MCMC simulation from within the GLUT event loop

In order to appropriately control and run your MCMC program using the Gf(MC)^2 interface, it is necessary to define the three functions:

1. `gfmUserDefd_OneStepByGlobals()`
2. `gfmUserDefd_InitializeChain()`
3. `gfmUserDefd_ResetAllAverages()`.

Section 2.1 describes the GLUT event loop, and how it is that the event loop is used, in conjunction with the above three functions, to perform iterative updates of the chain in an MCMC simulation. Sections 2.2 and 2.3 then describe more specifically how you should define the three functions listed above for your particular MCMC application.

2.1 The GLUT event loop

GLUT is a windowing system and application programming interface that allows user input via the keyboard, mouse, or other devices, while at the same time updating images drawn in OpenGL in the open windows. It accomplishes this by having an event loop. This loop is entered by calling `glutMainLoop()`, which is a function that enters the GLUT event loop and never returns. (Note that the Gf(MC)^2 developer will not call `glutMainLoop()` directly. Rather, that function is called from the Gf(MC)^2 function `gfmInitGFM()`.) When the program enters the GLUT event loop, it continuously cycles, and at each cycle it monitors keyboard and other input devices for any activity. If it records some user input from one of those devices, it executes a function appropriate to the type of input (this, I believe, is called "registering a callback"). When there is no user input, then GLUT registers a callback to its function `glutIdleFunc()`. In Gf(MC)^2 this callback function points to the Gf(MC)^2 function `gfmIdleFunc()` whose definition is given below:

```
/*  this is the function that GLUT calls as its idle function.  */
/*  This one, in turn, calls  */
/*  the gfmUserDefd_OneStepByGlobals function, but all */
/*  the stopping and going of the simulation */
/*  takes place within this function */
void gfmIdleFunc(void)
{
  if(gGo==1  || gDoASingleSweep == 1 )  {
    gfmUserDefd_OneStepByGlobals();
    gNumSweeps++;
    gNumSweepsAfterBurnIn++;

    gDoASingleSweep = 0;
  }
  gfmUpdateImages();
}
```

This is a simple function. It says that if the global variable `gGo` is 1, then the function `UserDefd_One-StepByGlobals()` should be called. This is a function that advances the MCMC algorithm one iteration. The variable `gDoASingleSweep` is a global variable that is used for being able to proceed through the simulation a single sweep at a time. `gNumSweeps` and `gNumSweepsAfterBurnIn` are global variables that count the number of sweeps since the Markov chain was initialized, and the number of sweeps after the burn in period was discarded. Finally, regardless of whether or not the Markov chain is running in the program, the function `gfmUpdateImages()` is called to cycle over all the open windows and draw the graphics images in them.

Notice that there are a lot of global variables running around there. In order to be platform independent (I guess) the GLUT interface does not support very much argument passing back-and-forth, and so it is necessary to define a few global variables to be able to pass information from GLUT to the developer's program and back. That is just the way it goes. I have followed the convention that global variable names begin with a lower-case `g` followed by an upper-case letter included in the rest of the variable's name.

2.2 Function `gfmUserDefd_OneStepByGlobals()`

As should be apparent from above, the function `gfmUserDefd_OneStepByGlobals()` is how the GLUT event loop drives your MCMC simulation. It is very easy to define if already have a function

that does a single iteration of your chain. Let's say you do, and it is called `DoASingleSweep()` and it takes as an argument a pointer to all the variables involved in the simulation (let us say that is a pointer to a data structure of type "`struct chain_vars`"). Let us also imagine that you have established a global variable named `gC` which is a pointer to the `chain_vars` struct that holds all variables in the simulation. Then, you might define `gfmUserDefd_OneStepByGlobals()` to look something like:

```
void gfmUserDefd_OneStepByGlobals(void)
{
  DoASingleSweep(gC);
  IncrementValues(gC);

  if(gNumSweepsAfterBurnIn%300==0 && gNumSweepsAfterBurnIn > 0) {
    OutputHistograms(gC);
  }
}
```

Note that `gfmUserDefd_OneStepByGlobals` takes no arguments, it has the prototype

```
void gfmUserDefd_OneStepByGlobals(void)
```

(and this is why you need the global pointer to your variable structures). The function above does a sweep of the chain, then it calls `IncrementValues(gC)`, which, in this case, calculates the running Monte Carlo average of the variables of interest (so, those averages must be accessible by the pointer `gC`, or some other pointer that is a global variable). The `if` statement says that every 300 iterations after the burn-in period, histograms of the values of the variables should be output to a file (for example, if the user desires some text output). Pretty straightforward stuff.

Notice that if you have programmed your simulation in C++, and you have a class called `all_variables` (or something like that) with a method called `OneIter` that does one iteration of the chain. Then, you will need to declare a global pointer, say `gPtrToAll`, to type `all_variables`, and assign to it the address of the pointer that your own program maintains to the instance of the class that holds all the variables. Then you could define `gfmUserDefd_OneStepByGlobals()` something like the following:

```
void gfmUserDefd_OneStepByGlobals(void)
{
  gPtrToAll->OneIter;

  /* and if you had other methods for incrementing the averages, etc. */
  /* maybe you would also do */
  gPtrToAll->ComputeRunningAverages;
}
```

It's all pretty straightforward.

2.3 Initializing the chain, and resetting averages

The next two user-defined functions have prototypes:
`void gfmUserDefd_InitializeChain(void)` and
`void gfmUserDefd_ResetAllAverages(void)`.
The names should tell you what they do. If you don't have a separate function that initializes

your MCMC simulation from certain starting values, then you should. And, it would be good, if possible, to let that function initialize the chain to random, and possibly "overdispersed" starting values. Let's imagine that you do have such a function, and you've called it `InitMyChain()`. Then you'd define something like:

```
void gfmUserDefd_InitializeChain(void)
{
  InitMyChain(gC);
}
```

Likewise, if you don't already have a separate function to reset the variables that hold the Monte Carlo averages in your simulation, you probably should... call it `ResetMyAves()`. Then, of course, you would define something like:

```
void gfmUserDefd_ResetAllAverages(void)
{
  ResetMyAves(gC);
}
```

if the function `ResetMyAves()` can access those averages through the pointer `gC`. If you store your averages in some other variable, then you'd have to declare a global variable that points to them (say, `gMyAves`) and define the function:

```
void gfmUserDefd_ResetAllAverages(void)
{
  ResetMyAves(gMyAves);
}
```

or something along those lines.

The function `gfmUserDefd_ResetAllAverages()` gets called when GLUT registers a callback from the user pressing the e key (see the Gf(MC)^2 *User's guide*). At the same time `gNumSweepsAfterBurnIn` gets set to zero by Gf(MC)^2.

When the user presses s, then Gf(MC)^2 calls both `gfmUserDefd_ResetAllAverages()` and `gfmUserDefd_InitializeChain()` and sets both `gNumSweepsAfterBurnIn` and `gNumSweeps` to zero. There is only one more detail—when the user presses alt-s, then, before `gfmUserDefd_InitializeChain()` gets executed, the global variable `gUseSameSeeds` gets set to 1 (otherwise, if the user just pressed s, `gUseSameSeeds` gets the value 0). This allows the developer to restart the simulation with the same set of random seeds that the last chain initialization used. The slightly more involved definition of the function might then look something like:

```
void gfmUserDefd_InitializeChain(void)
{
  long temp1, temp2;

  if(gUseSameSeeds == 1) {
     setall((long)gC->Seed1, (long)gC->Seed2);
  }
  else {
    getsd(&temp1, &temp2);
    gC->Seed1 = temp1;
    gC->Seed2 = temp2;
```

```
 }
 InitMyChain(gC);
}
```

In the above listing, `setall()` is a function that seeds the random number generator using two variables of type `long`, and the function `getsd()` retrieves the current state of the random number generator, (in effect giving seeds with which one may start from this state at a later time, by calling `setall()`). The exact implementation will, of course, depend on the random number generator you use. Note that if you wish to start from the seeds you used to set the chain for the very first time, you will have to record that separately (see the following section on the function `main()`).

2.4 Necessary changes to function `main()`

Broadly, you may have to make three or four changes to your function `main()` and the source code file in which it resides. First, you must define a macro to ensure that the global variables used by Gf(MC)^2 are properly declared. Second, your function `main()` must have the prototype:

```
int main(int argc, char **argv)
```

This allows the Unix and Linux versions to take command line options that affect GLUT. Third you must initialize your MCMC simulation. And finally, you must add two lines to enter the GLUT event loop.

2.4.1 Header files and `UN_EXTERN` for Global Variables

As noted earlier, there are many global variables in Gf(MC)^2. Quite a number of these variables have scope that extends beyond a single source file—they are declared in header files that appear in several different source files. In order to deal with this, I have declared them using statements like:

```
GLOB int gGlobalVariable;
```

where `GLOB` is a macro that expands to `extern` if the macro `UN_EXTERN` is not defined, and expands to nothing if `UN_EXTERN` is defined. I tend to make all of these global variables declared as `extern` except where they are declared in the file in which function `main()` resides. For this reason, the file in which your own function main resides should begin (before including any header files, *etc.*) with the statement:

```
#define UN_EXTERN
```

Then, of course, you will want to include the header file `GFMCMC.h`.

2.5 Initializing the chain before entering the GLUT loop

When you run your MCMC program, you have to get the data that you are analyzing, then you have to allocate memory to all of your variables and set all the settings for the MCMC run. And finally you have to initialize all the variables and start the chain. Using Gf(MC)^2 is almost the same, except, after initializing all the variables, you have to:

1. Assign the global pointer to all your variables (`gC` in previous examples) to the address of the block in memory that holds all those variables

2. Initialize the GLUT window system by issuing the statement[1]

```
glutInit(&argc, argv);
```

3. Initialize the Gf(MC)^2 system and enter the GLUT event loop with the line:

```
gfmInitGFM();
```

Hence, an example function `main()` might look like:

```
#define UN_EXTERN
#include "GFMCMC.h"
...
struct chain_vars *gC;  /* global pointer */
...
int main(int argc, char **argv)
{
  struct chain_vars *Variables;

  /* get the input, allocate memory, and set parameters */
  Variables = GetSettingsAndData("InputFile.txt");

  /* Initialize the variables */
  InitMyChain(Variables);

  /*  then do the three steps mentioned above:  */
  gC = Variables;
  glutInit(argc,argv);
  gfmInitGFM();

  /*  put a return here, even though you'll never reach it  */
  return(0);

}
```

3 Defining the Windows

While executing all the steps outlined in Section 2 will allow you to run your MCMC simulation using the GLUT event loop, it won't actually let you see anything happening on your screen. In order to take advantage of the visualization features that Gf(MC)^2 offers, you must define a number of window types. Once the program has begun, you may then open those windows and view their contents which may be updated during the course of the MCMC simulation.

It is in the definition of the function `gfmUserDefd_DefineWindows()` that you will actually tell the Gf(MC)^2 system what each window will contain, what its initial boundaries will be, what its initial color scheme will be, *etc.* I have written a number of #defined macros to make it easier

[1] I have not used Gf(MC)^2 with any programs that require command line options. However, reading the GLUT manual, it appears that if you have command line options for your program *and* you wish to use command line options to affect GLUT, then you should call `glutInit(&argc, argv);` first thing in your program, and GLUT will then remove all the command line options specific to GLUT, leaving you with an `argc` and an `argv` suitable for processing the command line options specific to your own program.

to input all these things.[2] All these macros start with a lowercase `gs`, standing for "Gf(MC)^2 shortcut." Following is a hypothetical (though it follows closely a portion of the definition in my program `NewHybrids`) definition of `gfmUserDefd_DefineWindows()`. In the following subsections, I will go through the different parts of it, broken down by their function.

```
void gfmUserDefd_DefineWindows(void)
{
  /*  define the prefix for file names */
  gsOUTPUT_FILE_PREFIX("NewHybrids");

  /*  define the console window */
  gsCONSOLE("\"Info\" Window");
    gsDRAW_FUNC(DrawMyConsoleWindow);

  /* define another window */
  gsNEW_WINDOW(1,"Category Probablities");
    gsDRAW_FUNC(DrawCatProbs);
    gsCOLOR_KEYS(gC->CategNames);
    gsNUM_COLOR_KEYS(&gC->NumCategories);
    gsCOLOR_SCHEME(DEEP_BLUE);

  /* define another window with explicit boundaries in the clipping volume  */
  gsNEW_WINDOW(7,"Observed Data");
    gsDRAW_FUNC(DrawObservedData);
    gsXLO(-.15 * 3 * gC->NumDataCols);
    gsXHI(1.15 * 3 * gC->NumDataCols);
    gsYLO(-.15 * gC->NumDataRows);
    gsYHI(1.15 * gC->NumDataRows);
```

3.1 File Prefix Name

It is a bit of a kluge to be defining this in the function for defining windows, but this is the way it is done. You should issue the statement

```
gsOUTPUT_FILE_PREFIX("PrgName");
```

at the beginning of the function with "PrgName" replaced by the name that you want appended to the beginning of the files that this program will use to store information about its views and color schemes. (*i.e.*, they come out with file names like `PrgName_PreDefdViews.txt`).

3.2 The Console Window

Every Gf(MC)^2 based program has one window that must remain open all the time. This is what I call the console window. In it, I find it useful to list the name of the program, the author name, maybe a few options that are in effect or the data file that is being analyzed, and then the values of the `gNumSweeps` and `gNumSweepsAfterBurnIn` variables. The window is created by issuing the statement

```
gsCONSOLE("WindowName");
```

[2] These macros are found in the header file `GFMCMC_Shortcuts.h` which automatically gets included from `GFMCMC.h`.

where "WindowName" is a string that can be whatever you want it to be (enclosed in double quotation marks, of course). It is the title of the window (which will typically be displayed on the top bar of the frame around the window).

The next statement in the listing above is

```
gsDRAW_FUNC(DrawMyConsoleWindow);
```

This tells Gf(MC)^2 that the drawing commands that should be executed in the window that was just defined (in this case it is the console window) are in the function `DrawMyConsoleWindow()` which must also, of course be defined separately. The prototype for all such drawing functions is

```
void DrawingFunction(void)
```

The contents of such DrawingFunctions will typically be OpenGL graphics commands. These will be described in Section 4.

3.3 The `gsNEW_WINDOW` Macro

`gsNEW_WINDOW` is a macro that takes two arguments. The first is an integer and should be the unique identifier of this type of window (it will be used by Gf(MC)^2 to open new windows of this type and to set all their settings as appropriate). The second argument is a string which is the title of the window. Hence it should be clear what the

```
gsNEW_WINDOW(1,"Category Probablities");
```

statement does. And of, course, the following `gsDRAW_FUNC(DrawCatProbs)` assigns the drawing function `DrawCatProbs()` to the window.

3.4 Defining the Legend Entries

The next two lines in the listing deal with the legend that will be attached to the "Category Probabilities" window.

```
gsCOLOR_KEYS(gC->CategNames);
```

means that the names held in the variable `gC->CategNames` (letting `gC` be our global pointer to lots of variables relevant to the MCMC simulation) are the names that will be linked to the different colors in the window. It is necessary that `gC->CategNames` be of type `char **`. (A pointer to pointers to character, *i.e.*, an array of strings).

The next line in the listing is where the number of different legend items is defined. There is a bit of a vestige here in that I assign the address of the variable rather than the value. At some point I will probably change that back to be just the value. At any rate,

```
gsNUM_COLOR_KEYS(&gC->NumCategories);
```

is what sets the number of items that will appear in the legend. The argument of the macro must be an `int *` (pointer to `int`).

The names for the legend entries and the number of them will depend on what sort of problem that you are dealing with by MCMC. For example, if you are doing a genetics problem in which this particular window is displaying the currently inferred genetic sequences of a collection of ancestral chromosomes, then the color keys might be a pointer to the array of strings: "`A`", "`C`", "`G`", and "`T`", with each different color representing a different nucleotide base. On the other hand, you might be doing a model-based clustering problem with `gNumClusters` clusters, and you are plotting a constantly-updated density estimate of the values of the mixing proportions for those clusters, with each cluster given a different color. In that case, the color keys might be a pointer to the array of strings "`Cluster1`", "`Cluster2`", *etc.*

3.5 Setting the Color Scheme

The initial color scheme of a window is set with, for example,

```
gsCOLOR_SCHEME(DEEP_BLUE);
```

This is the color scheme that will be applied to the window. Currently, the color scheme of a window does not get reset in a pre-defined view or user-defined view. There are four pre-defined color schemes, which can be arguments to `gsCOLOR_SCHEME`. These are `FISHER_PRICE`, `OL_GRAYBACK`, `DEEP_BLUE`, and `GILBERT_AND_SHERMAN`. The default color scheme is `FISHER_PRICE`.

3.6 Setting the clipping volume

When you draw things with OpenGL, you do so by placing scenes in 3-space. That scene in 3-space has to get transferred to a 2-D window. There are several different steps involved here that I won't go into (pick up any book on OpenGL), but one of the important ones is defining the region in 3-space that will actually be drawn. This is called the clipping volume. Currently Gf(MC)^2 only deals with orthographic projections and doesn't have any features for rotating 3-D images, *etc.* Those could be added easily, but I don't know when I will get around to it. So, for the most part, for now we are only interested in the boundaries of the clipping volume in the x and y directions. These can be set with the `XLO`, `XHI`, `YLO`, and `YHI` macros. (There are also `ZLO`, `ZHI` macros, but they will do little until I incorporate more 3-D features into Gf(MC)^2.) The example in the listing is in the definition for the "Observed Data" window:

```
gsXLO(-.15 * 3 * gC->NumDataCols);
gsXHI(1.15 * 3 * gC->NumDataCols);
gsYLO(-.15 * gC->NumDataRows);
gsYHI(1.15 * gC->NumDataRows);
```

So, for example, if there are 20 columns of data, then the clipping volume will extend from $-.15 \times 60$ on the left hand side to 1.15×60 on the right hand side. This would be useful if each column of data is depicted by some figure that is of width 3 units. The y dimension is similar...

3.7 Adjusting the Axis Settings

If you wish to have axes on your figure, you can control how they will appear with a number of different macros:

4 Drawing Functions

4.1 An Example of a Drawing Function

4.2 Some simple OpenGL Commands

4.3 Some Gf(MC)^2 Drawing Utilities

4.4 Fitting to Extrema

5 Advanced Control for Menus, *etc.*

5.1 Function `gfmUserDefd_DefineMenus()`

This is another function which gets called when Gf(MC)^2 is setting up its variables and initializing itself (this is quite a different thing than initializing the variables in the Markov chain). It is a place where one can issue GLUT commands to declare some menus. I won't say much about it here, but will describe its use in one of my programs in which it is defined as:

```
void gfmUserDefd_DefineMenus(void)
{
  gUserDefdMenus[ARC] = glutCreateMenu(HandleARC);
  glutAddMenuEntry("Open Histogram View For Selected Locus", 5);

  gUserDefdMenus[CRC] = glutCreateMenu(HandleCRC);
  glutAddMenuEntry("Open Histogram View For Selected Individual", 0);
}
```

Briefly, this creates two menus that will be available to be attached to one of the three mouse clicks (left, right, and center) for certain window types, should the developer see fit. `ARC` and `CRC` are #defined macros that expand to 0 and 1. They stand for "Allele right click" and "Category right click" because I intend to attach them to the allele frequency and the category frequency windows of my program `NewHybrids`. The function `glutCreateMenu` is a GLUT function that creates a menu, returns the integer identifier of that menu (which gets stored in the Gf(MC)^2 global array of ints, `gUserDefdMenus`). The menu that gets created is one that will register a callback to GLUT when the menu is selected that tells it to call the function `HandleARC`, which, in turn is a function that takes a single integer as an argument. The function `glutAddMenuEntry()` adds an entry to the "current" menu (which, in this case, is the one just created by `glutCreateMenu`). In the first instance above, it is saying to add a menu item named "Open Histogram View For Selected Locus" with the menu callback number 5. This means that if you select the item "Open Histogram View For Selected Locus" from that menu, GLUT will call the function `HandleARC`, passing to it the argument "5".

All the foregoing has merely created some menus, which may later be attached to certain mouse clicks for certain windows. This will be described briefly in the next section. For the most part, you typically won't need to create any new menus unless you know what you are doing with GLUT and want to add some features to your Gf(MC)^2 program. In such cases it suffices to define `gfmUserDefd_DefineMenus()` as

```
void gfmUserDefd_DefineMenus(void)
{
  return;
}
```

A Window Settings

B System Requirements (to Run the Programs)

GLUT is a windowing system that has been designed to be portable across most major operating systems. Some configuration of the system may still be necessary. Information about the requirements for running OpenGL (necessary for rendering the graphics in Gf(MC)^2) on different platforms is found at

http://www.opengl.org/users/downloads/index.html

Libraries necessary for GLUT are in different locations, as indicated below for each platform. The information page for GLUT is at:

http://www.opengl.org/developers/documentation/glut.html

B.1 Macintosh

As you can read on the OpenGL website:

> OpenGL ships with OS 9 and OS X. You can also optain the latest software version on the Apple OpenGL web site. You also need an OpenGL hardware accelerator driver. All new PowerMac G4, iMac, iBook and PowerBook computers ship with built in hardware acceleration and include the correct hardware driver. If you buy a different or additional hardware board, you can obtain the driver from each board manufacturer's web site.

So if you have OS 9 or higher you shouldn't have to do anything other than run the program. Otherwise, or if there is a problem, it is probably because the system does not have the necessary GLUT and/or OpenGL libraries. These may be found at

`http://docs.info.apple.com/article.html?artnum=120000`

As far as I can tell from the website this stuff will be compatible with OS 8.1 or greater.

B.2 Microsoft Windows

Again, from the OpenGL site:

> OpenGL v1.1 software runtime is included as part of operating system for WinXP, Windows 2000, Windows 98, Windows 95 (OSR2) and Windows NT. So you only need to download this if you think your copy is somehow missing. The OpenGL v1.1 libraries are also available as the self-extracting archive file from the Microsoft Site via HTTP or FTP.
>
> OpenGL v1.2 & 1.3 are included with the drivers for your OpenGL video cards. So you only need to make sure you have the latest OpenGL driver for your video card. If you do not have the latest driver with OpenGL 1.2 or 1.3 support, either use GLSetup (for Win95/98 only) or contact the video card manufacturer directly and ask them for an OpenGL 1.2/1.3 driver for your card and OS.
>
> You do need an OpenGL hardware driver for your particular 3D hardware accelerator board. For consumer-level boards under WIndows 95 or Windows 98 run Glsetup to automatically download and install the latest & greatest video driver. Glsetup is a

program written specifically to analyze your hardware and install working OpenGL drivers for that hardware. Currently (Jan 2001), Glsetup supports cards using the following chipsets:

- 3Dfx Voodoo, Voodoo2, Voodoo Rush, Banshee, Voodoo3, Voodoo3 3500TV, Voodoo5
- 3Dlabs Permedia 2 and Permedia 3
- ATI Rage 128, Rage 128 Pro, Rage Fury MAXX, Rage Pro, Radeon
- Intel i740, i810 and i815
- Matrox G200, G400 and G450
- NVidia Riva 128/128ZX and Riva TNT/TNT2/GeForce1&2/Quadro1&2
- Rendition Verite 2200
- S3 Savage3D, Savage4, and Savage2000

For cards with chipsets not listed, for WinNT, Win2000 or WinXP or for professional workstation graphic cards you can obtain OpenGL drivers directly from each board manufacturer's web site.

To run the GLUT portion of Gf(MC)^2 you have to be able to link the the dynamic library glut32.dll. Go to the website

http://www.opengl.org/developers/documentation/glut/index.html#windows

and follow the directions. Basically you have to download a file, unzip it, and put glut32.dll in your system folder.

B.3 Linux

Will deal with this later. In the meantime, if you are using Linux, you probably already know what you need to do.

B.4 Unix

Will deal with this later.

C Required Libraries and Headers for Code Development

Check for you particular platform.

D Software Agreement

REGENTS SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY, PROVIDED HEREUNDER IS PROVIDED "AS IS". REGENTS HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.