LittleFe: GalaxSee Image Processing at Polytechnic University of Puerto Rico

Ricardo J. Negrón Fontanez
Master of Engineering in Electrical Engineering
Dr. Luis Vicente
Electrical & Computer Engineering & Computer Science Department
Polytechnic University of Puerto Rico

Abstract *— The purpose of this project is to manage images in the current LittleFe project in the GalaxSee program. This program is installed in a Linux base Bootable Cluster CD version 3.3.1. This Linux version doesn't contain any library that is capable to manage and process external images, but it's able to manage pixels and pixels colors by the Xlib library, available in this Linux version. By programing a class that is able to open a bitmap file in a byte format, then by removing the image header we obtain all the bytes that contain the image colors. These bytes where divided in group of three, each group contained 24 bits. This 24 bits reflects the Red, Green and Blue pixel color values also known as the RGB triplet. Each channel represents an 8 bit value that is used to provide the intensity of each color channel. By reading the bitmap file in a interval of 24 bits we manage to read each pixel color. By storing all the color values into a three dimensional array variable we were able to include the bitmap file pixels in the GlaxSee program and the bitmap file pixel colors.*

Key Terms *— C++, Image Processing, Libraries, UNIX.*

WHAT IS LITTLEFE?

LittleFe is a portable multi-node computational cluster that supports shared memory parallelism (OpenMP), distributed memory parallelism (MPI) and GPGPU parallelism (CUDA) designed for educational purposes associated with high performance computing (HPC) and computational science teaching key concepts such as speedup efficiency, and load balancing being much effectively done on a parallel platform. [7]

WHAT IS GALAXSEE?

GalaxSee is a program that is installed in a BCCD Linux environment that simulates a galaxy using Newtonian law of physics. It can manage any amount of stars; manage the mass of each star and the number of years that will go through before exiting automatically the program. [3]

GALAXSEE PHYSICS

One of the grand challenge problems in astronomy is the evolution and structure of the universe and galaxies. Galaxies often have a spiral structure that is difficult to explain. Space is not occupied by a homogeneous fluid, but by discrete particles that interact through gravity over long ranges. [1]

This is often modeled as discrete bodies interacting through gravity. The gravitational force is given by:

$$F_g = G\, M_1 M_2 / D_2, \tag{1}$$

Where M_1 and M_2 are two objects masses and D is the distance between them. The acceleration of an object is given by the sum of the forces acting on that object divided by its mass:

$$a = \Sigma F/M \tag{2}$$

If you know the acceleration of each mass, you can calculate the change in velocity

$$a = \Delta v/\Delta t \tag{3}$$

If you know the velocity, you can calculate the change in position

$$v = \Delta x/\Delta t \tag{4}$$

The algorithm can be loosely described as **NEW = OLD + CHANGE**. This is applied to each particle. To apply this to each particle, you need to know the acceleration, but the acceleration is determined by the sum of all of the forces.

What this means is that the more objects you have, the more forces you need to calculate. What's worse, every object needs to know about every other object. [1]

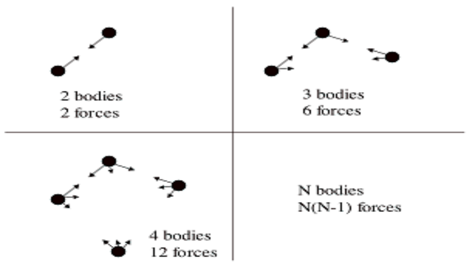


Figure 1
N Bodies Forces

GalaxSee Code Original State

The GalaxSee code is a simple implementation of parallelism. Since most of the time in a given N-Body model is spent calculating the forces, we only parallelize that part of the code. "Client" programs that just calculate accelerations are fed every particle's information, and a list of which particles that client should compute. A "server" runs the main program, and sends out requests and collects results during the force calculation. [1]

You could think of the total running time in the following way:

- A: It takes some time to send out information on N particles to P-1 processors each of S time steps.
- B: It takes some time for each processor to calculate N/P * N interactions each of every S time steps.

As long as you run the model for enough time steps that not much time is spent "setting up" the program, a reasonable model for how long the GalaxSee program will take to run on a given cluster is

$$\text{Time} = A*N*(P-1) + B*N*N/P \qquad (5)$$

Running GalaxSee Program in Original State

To run the program, first move into the **GalaxSee** directory by executing **cd ~/GalaxSee**. Next the executable needs to be "**made**" by running make. This will create the executable **GalaxSee.**

Next we need to copy this executable to all the nodes that will be running it. To do so first in your terminal run **bccd-allowall**, and then run **bccd-snarfhosts**.

To run the program on one node of your cluster, enter the following command

```
mpirun –np 1 –machinefile ~/machines-openmpi
./GalaxSee.cxx-mpi 500    400    1000.0        (6)
```

For running it on multiple processors, make the appropriate changes to the number of processors (**-np**), follow the instruction above. [1]

Objective

Analyze the current program, study its behavior and how it works. Then identify the code part that manages the pixel mass placements and improve the code by including any necessary libraries, classes or functions.

Currently the program shows each mass as a pixel and displays it in a window as it appears in the following image:

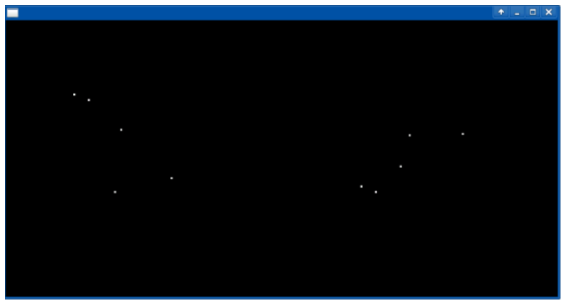


Figure 2
GalaxSee Masses Display

The following images are going to be displayed in substitution of the pixels demonstrated on the previous image.

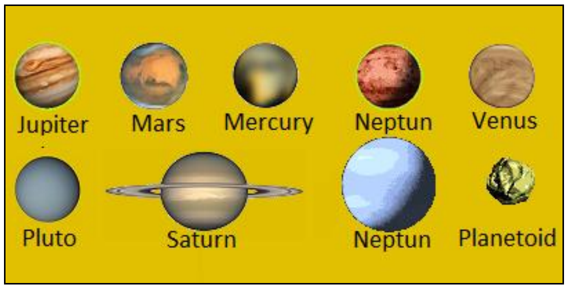


Figure 3
Planets

RESEARCH RESULTS

During the analysis of the GalaxSee programmed code I found that the program doesn't manages images but is able to manage pixels and their colors using the **X11/Xlib** library that is included in the BCCD Operating System as many other libraries.

In the process on verifying if the BCCD OS contained any pre-installed library that could manage a image format file, we found that it didn't had at the moment a known library that could manage a image file. Having this constraint we pursue to find and install a library that could ease our objective. [4]

In this search we found the following libraries: **libPNG**, **JPEGlib**, **Magik++**, **IMGlib2**, **EasyBMP**. The **libPNG** and **Magik++** were installed successfully but it didn't compile correctly given an error of compatibility with the Math library being only compatible with Intel compilers. The **JPEGlib** wasn't able to install the successfully in the lib directory. The **IMGlib2++** didn't contained a configure file required to perform the make file that is needed to perform the installation of the library. Finally we installed the **EasyBMP** successful but wasn't able to execute the library codes correctly because it was programmed to manage Windows bitmap images, generating for this reason errors during its usage.

By having problems in the installation of the previously mentioned libraries we directed our search on managing the image files in a native form.

BITMAP FILE FORMAT

In our research on selecting an adequate image file format we selected the Bitmap file format. This format is capable of storing 2D digital images of arbitrary width, height, resolution, color depths and color profiles.

The bitmap image file consists of a fixed size structure known as a header and a variable-size structure appearing in a predetermined sequence.

Table 1
Bitmap File Composed Structure

Structure Name	Optional	Size	Purpose	Comments
Bitmap File Header	No	14 Bytes	To store general information about the Bitmap image File	Not needed after the file is loaded in memory
DIB Header	No	Fixed-size (however 7 different versions exist)	To store detailed information about the bitmap image and define the pixel format	Immediately follows the Bitmap File Header
Extra bit masks	Yes	3 or 4 DWORDs[6] (12 or 16 Bytes)	To store detailed information about the bitmap image and define the pixel format	Immediately follows the Bitmap File Header
Color Table	Semi-optional	Variable-size	To define colors used by the bitmap image data (Pixel Array)	Mandatory for color depths <= 8
Gap1	Yes	Variable-size	Structure alignment	An artifact of the File Offset to PixelArray in the Bitmap File Header

Pixel Array	No	Variable-size	To define the actual values of the pixels	The pixel format is defined by the DIB Header or Extra bit masks. Each row in the Pixel Array is padded to a multiple of 4 bytes in size
Gap2	Yes	Variable-size	Structure alignment	An artifact of the ICC Profile Data offset field in the DIB Header
ICC Color Profile	Yes	Variable-size	To define the color profile for color manageme nt	Can also contain a path to an external file containing the color profile. When loaded in memory as "non-packed DIB", it is located between the color table and gap1.

BITMAP FILE HEADER

This block of bytes are in the start of the file and are used to identify the file. The GalaxSee program proceed to first open the image file and verify if its headers corresponds to a BMP file of 24 bits. The data that should be obtain on the header would be of two bytes, containing the characters 'B' on the first byte and in the second byte the character 'M'. All of the integer values are stored in little-endian format. [6]

DIB HEADER (BITMAP INFORMATION HEADER)

This block of bytes tells the application detailed information about the image, which will be used to display the image in the screen. The block also matches the header used internally by Windows and OS/2 and has several different variants. All of them contain a dword field, specifying their size, so that an application can easily determine which header is used in the image.

The GalaxSee program ignores these bytes because there are not needed for displaying the image pixels colors in the BCCD window.

COLOR TABLE

The color table is contained after the BMP file header, the DIB header and after an optional three red, green and blue bitmasks if the BITMAPINFOHEADER header with BI_BITFIELDS option is being used. The number of entries in the color table is 2^n or a smaller number specified in the header.

The color table is a block of bytes listing the colors used by the image. Each pixel in an indexed color image is described by a number of bits (1, 4 or 8) which is an index of a single color described by the table. The purpose of the color table in indexed color bitmaps is to inform the application about the actual color that each of these index values corresponds to. The purpose of the color table in a non-indexed bitmaps is to list the colors used by the bitmap for the purposes of optimization on devices with limited color display capability and to facilitate future conversion to different pixel formats. [6]

The colors in the color table are specified in the 4-byte per entry known as RGBA32 format. The color table used with OS/2 BITMAPCOREHEADER uses the 3-byte per entry known as RGB24 format. Knowing this, we were able to identify the correct Bitmap image format needed to obtain only the image pixel colors. The RGB24 format is the correct bitmap image format that is needed to obtain the pixel color bites. The first 8 bites will store the Red color values, the next 8 bites will store the Green color values and the last 8 bites will store the Blue color values, each color values will contain a range from 0 to 255 color value (RED, GREEN,BLUE), (0 to 255, 0 to 255, 0 to 255). [6]

PIXEL STORAGE

The bits representing the bitmap pixels are packed in rows. The size of each row is rounded up to a multiple of 4 bytes by padding. For images with height > 1, multiple padded rows are stored consecutively, forming a Pixel Array. The total number of bytes necessary to store one row of pixels can be calculated as:

$$\text{RowSize} = \left\lfloor \frac{\text{BitsPerPixel} \cdot \text{ImageWidth} + 31}{32} \right\rfloor \cdot 4, \quad (6)$$

The total amount of bytes necessary to store an array of pixels in an *n* bits per pixel (bpp) image, width 2^n colors, can be calculated by accounting for the effect of rounding up the size of each row to a multiple of a 4 bytes, as follows:

$$\text{PixelArraySize} = \text{RowSize} \cdot |\text{ImageHeight}| \quad (7)$$

PIXEL ARRAY

Is a block of 32-bit DWORD that describe the image pixel by pixel. This pixels are stored upside down with respect to normal image raster scan order, starting in the lower left corner, going from left to right, and then row by row from the bottom, when the Image Height value is negative. [6]

INCLUDING BMP FILES IN GALAXSEE

We previously explained how a bitmap image files work. But we need to first include the following libraries to complete this task:

- **Sstream**: Is a part of the C++ Standard Library that contains a header file that provides templates and types that enable interoperation between stream buffer and string objects. [9]
- **String.h**: Is a contiguous sequence of code units terminated by the first zero code. There are two types of strings: strings, which is sometimes called **byte string** which uses the type chars as code units and **wide string** which uses the type *wchar_t* as code units. [8]
- **Cmath**: Provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. [5]

Then we proceed to create a structure that will contain the color table of each planet images and their image sizes. The structure is called ***Planet*** and it contains:

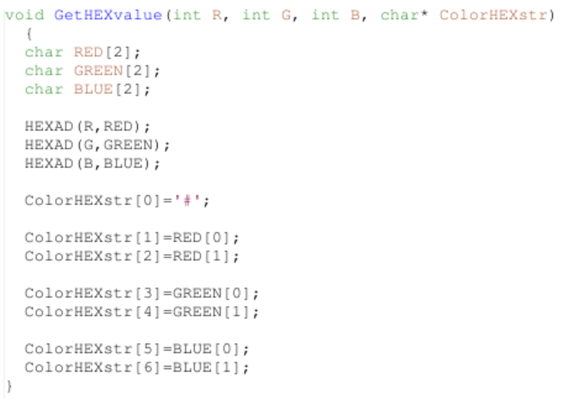
- **Char *** HEXcolorArray**: 3 dimensional char that will collect the hexadecimal color values of each pixel.
- **Char ** XColor:** 2 dimencional char that will collect a PAM tuple type color value used in the GalaxSee program to identify the color value of the pixel.
- **Int width:** It will store the image width in an integer.
- **Int height:** It will store the image width in an integer.

Table 2
PAM Tuple Types

TUPLTYPU	MAXVAL	DEPTH	Comments
BLACKANDWHITE	1	1	Special case of GRAYSCALE
GRAYSCALE	2…65535	1	2 bytes per pixel for MAXVAL > 255
RGB	1…65535	3	6 bytes per pixel for MAXVAL > 255
BLACKANDWHITE_ALPHA	1	2	2 bytes per pixel for MAXVAL > 255
GRAYSCALE_ALPHA	2…65535	2	4 byes per pixel for MAXVAL > 255
RGB_ALPHA	1…65535	4	8 bytes per pixel for MAXVAL > 255

We declared 10 variables with the previously mentioned structure called “Planet”. The created

Planet variables are: Earth, Jupiter, Neptun, Uranus, Mars, Mercury, Pluto, Saturn, Venus and Planetoid. Each Planet will open their BMP file by a standard ifstream that are being stored at the following directory: */bccd/home/bccd/GlaxSee/Imagenes/*. When the Planet image is opened then we verify if the header is from a BMP file by verifying if the image type is of 19778 and if its bits counts to 24, if the image contains this bits counts and the correct header then we proceed to read each 3bytes and convert them from 24 bits to a Hexadecimal value by calling the function ***GetHEXvalue*** and store them in the ***HEXcolorArray*** that is in each Planet Structure. [2]

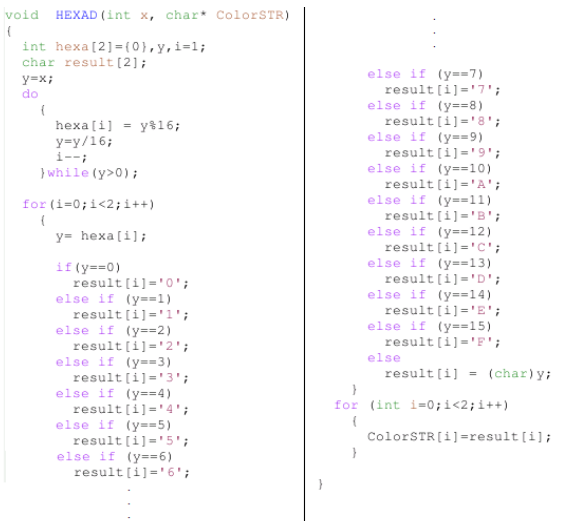


```
void GetHEXvalue(int R, int G, int B, char* ColorHEXstr)
  {
  char RED[2];
  char GREEN[2];
  char BLUE[2];

  HEXAD(R,RED);
  HEXAD(G,GREEN);
  HEXAD(B,BLUE);

  ColorHEXstr[0]='#';

  ColorHEXstr[1]=RED[0];
  ColorHEXstr[2]=RED[1];

  ColorHEXstr[3]=GREEN[0];
  ColorHEXstr[4]=GREEN[1];

  ColorHEXstr[5]=BLUE[0];
  ColorHEXstr[6]=BLUE[1];
}
```

Figure 4
GetHEXvalue Function

Now that we acquired each pixel color in a Hexadecimal value we proceed to send those pixel colors to the bccd window by first calling the **XParseColor()** function. The ***XParseColor()*** function looks up the string name of color with respect to the screen associated with the specified color map. It returns the exact color value. This color value is stored in the *Planet structure* and used in the ***XAllocColor()*** function.

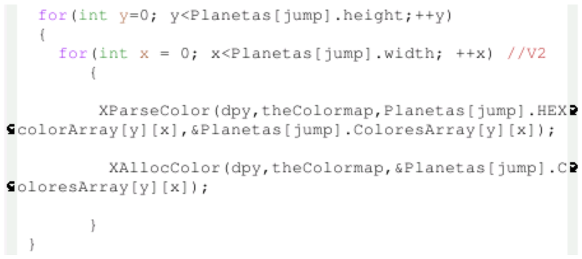
Each Planet image contains pixel colors that are not going to be place in the bccd Windows. The pixels that are not going to be used are the background of each image. To remove the background color we select a unique color as the background. The unique color has a hexadecimal value of “E0C000”. This means that each pixel color is verified and if it contains the hexadecimal value of “E0C000” it would not be placed in the bccd Window.



```
void  HEXAD(int x, char* ColorSTR)
{
  int hexa[2]={0},y,i=1;
  char result[2];
  y=x;
  do
    {
      hexa[i] = y%16;
      y=y/16;
      i--;
    }while(y>0);

  for(i=0;i<2;i++)
    {
      y= hexa[i];

      if(y==0)
        result[i]='0';
      else if (y==1)
        result[i]='1';
      else if (y==2)
        result[i]='2';
      else if (y==3)
        result[i]='3';
      else if (y==4)
        result[i]='4';
      else if (y==5)
        result[i]='5';
      else if (y==6)
        result[i]='6';
                .
                .
                .
      else if (y==7)
        result[i]='7';
      else if (y==8)
        result[i]='8';
      else if (y==9)
        result[i]='9';
      else if (y==10)
        result[i]='A';
      else if (y==11)
        result[i]='B';
      else if (y==12)
        result[i]='C';
      else if (y==13)
        result[i]='D';
      else if (y==14)
        result[i]='E';
      else if (y==15)
        result[i]='F';
      else
        result[i] = (char)y;
    }
  for (int i=0;i<2;i++)
    {
      ColorSTR[i]=result[i];
    }

}
```

Figure 5
HEXAD Function

The ***XAllocColor()*** function allocates a read-only color map entry corresponding to the closest RGB value supported by the hardware. It returns the pixel value of the color closest to the specified RGB elements supported by the hardware and returns the RGB value actually used. This value is stored in the Planet structure in the 2 dimensional ***XColor*** array called ***ColoresArray***. [2]



```
  for(int y=0; y<Planetas[jump].height;++y)
  {
    for(int x = 0; x<Planetas[jump].width; ++x) //V2
        {

          XParseColor(dpy,theColormap,Planetas[jump].HEX
colorArray[y][x],&Planetas[jump].ColoresArray[y][x]);

           XAllocColor(dpy,theColormap,&Planetas[jump].C
oloresArray[y][x]);

        }
 }
```

Figure 6
Store Supported RGB Colors on Planet Structure

To show the pixel value we need to call the XSetForeground() function to specify the foreground we want to set for the specified GC. A GC defines how the new destination bits are to be computed from the source bit and the old destination bits. In other words it removes the previous image pixel color location to a new location on the bccd Window. Then we call the ***XDrawPoint*** function to draw a single pixel in the bccd Window. The ***XDrawPoint*** function uses the foreground pixel and function components of the GC to draw a single point into the specified drawable. [2]

These changes were included in the GalaxSee project specifically in the Gal.cpp code. This c++ file manages each created star and includes it in the foreground at the bccd Window. With these changes being done to the Gal.cpp file we can now include BMP image files and give the GlaxSee program a more attractive effect on the Galaxy simulation.

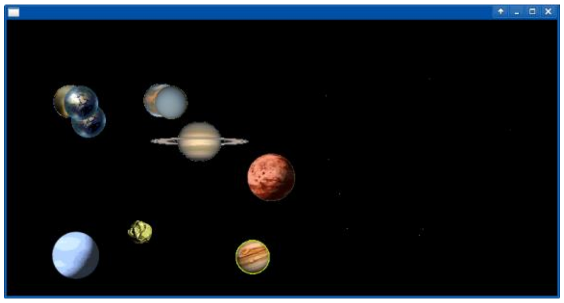


Figure 7
GalaxSee Program Simulation

REFERENCES

[1] *BCCD Version 3*, February 12, 2013. Retrieved on May 4, 2014 from GalaxSee: http://bccd.net/wiki/index.php/GalaxSee.

[2] *Chapter 6: Color Management Functions*, (n.d.). Retrieved May 5, 2014, from Tronche: http://tronche.com/gui/x/xlib/color/.

[3] *LittleFe*, February 21, 2011. Retrieved from LF WIKI: https://littlefe.net/wiki/index.php/LittleFe_Manual#GalaxSee_.28Located_in_Gal.29.

[4] *LittleFe Parallel and ClusterComputing Education On The Move*, 2012. Retrieved on May 4, 2012, from LittleFe: http://littlefe.net/.

[5] *Mathematical functions for complex numbers*. (n.d.). Retrieved May 4, 2014, from Python: https://docs.python.org/2/library/cmath.html.

[6] *Microsoft Windows Bitmap File Format Summary*. (n.d.). Retrieved May 4, 2014, from FileFormatInfo: http://www.fileformat.info/format/bmp/egff.htm.

[7] Robert M. Panoff, P. J. (2006). *LittleFe Overview*. Retrieved May 4, 2014, from Shodor: http://www.shodor.org/media/content//petascale/materials/general/presentations/littlefe-overview_pdf.pdf.

[8] *Safe C Library*, February 13, 2009. Retrieved on May 4, 2014, from Safe C Library: http://safeclib.sourceforge.net/.

[9] *Std:basic_stringstream*, August 15, 2013. Retrieved on May 4, 2014 from cppreference: http://en.cppreference.com/w/cpp/io/basic_stringstream.