University of Nevada
Reno

Image Maker

A professional paper submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science.

by

GuoLiang Sun

Dr. Frederick C. Harris, Jr., advisor

May 1998

Abstract

Image Maker is a program which runs on the Be[9] Operating system (BeOS). It allows the user to create a new image based upon available images. Users can load many kind of image files, select the interesting areas (small images), move them to where they want them, and then save them. When the user wants the new image, **Image Maker** renders all small images into a combined image and displays it. A Binary Tree and a parallel algorithm are used in **Image Maker**'s design. **Image Maker** can be used to create a Web page or serve as the basis for animation game. It can be treated as an independent software package or plugged into other software as a new feature.

Acknowledgments

As my formal education at the University of Nevada comes to a close, it is only appropriate that I thank those who have helped me make it to this point in my life.

My wife and my son for their encouragement and support of my success. Mr. William E. Bull, for his cooperation and helpfulness, as well as his BeBOX. Dr. Ed Wishart and Dr. Bruce Johnson, who served as committee members and spent time reviewing and critiquing this project as well as providing support and encouragement along the way. And finally, Dr. Frederick C. Harris, Jr., my advisor, for his support and encouragement for the completion of this project as well as the guidance throughout my graduate program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Combining images into a complex image is very popular and useful. Reusing available images to make a new image should be easy, efficient, and fast. There are lots such images in Web sites and advertisements. Many software packages let you create a new image based upon available images. For example, consider Microsoft Word[3]. You can use it as a text editor, a graphic editor, and an image editor. Figure 1.1 shows an image created by someone using Microsoft Word. It consists of three images: *rabbit*, *turtle*, and *donkey*. The three images have an order or a layering. The order or the layer identifies their overlapping behavior. The layer is very important in combined image processing. The upper layer images can shadow the lower layer images when they are overlapping.

Microsoft Word allows user to insert images anywhere, to change the sizes of images, and to keep the order as they are inserted. When user inserts an image, the whole image is composed, as a single atomic whether the user likes the whole image or not. The user cannot select only interesting areas from an image.

PhotoShop[10] is a full image processing software package which allows the user to select any area in an image. Figure 1.2 shows a PhotoShop screen. In the image

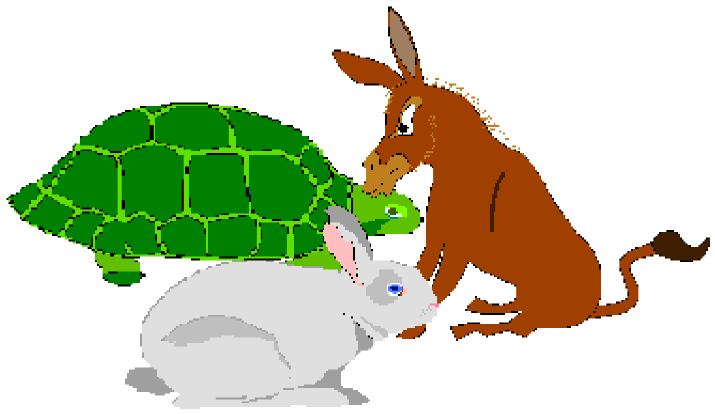
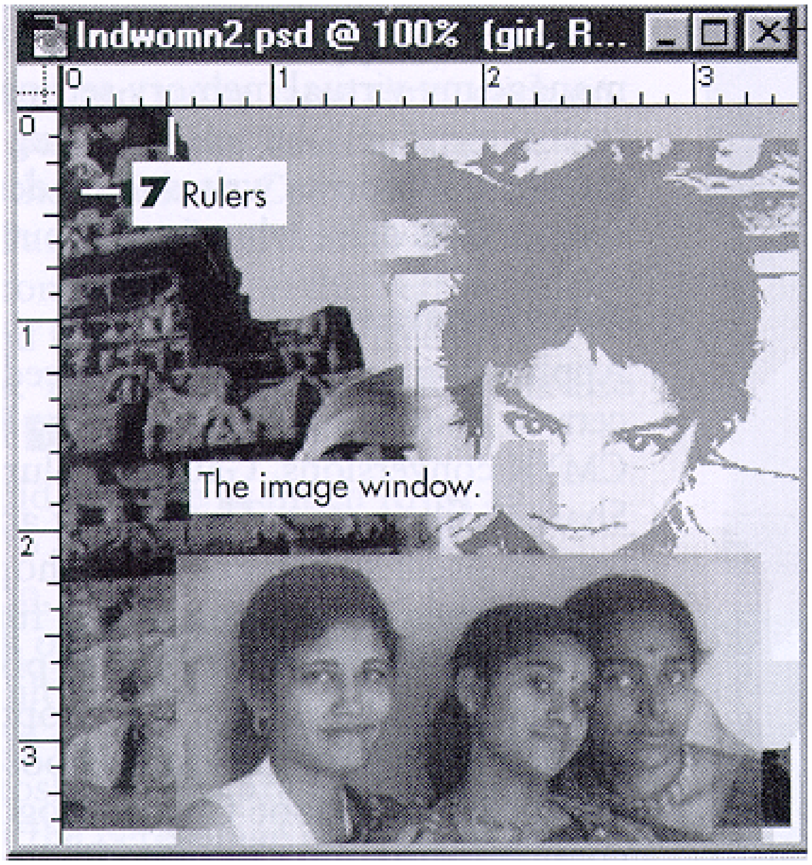


Figure 1.1: Combined image using Microsoft Word.

window, we can see the image, which consists of a background image, an image of a girl, and an image of a few women. Each image has a layer, which indicates its layering location among these images.

PhotoShop runs on an Intel 80386 or higher processor; a hard drive with at least 20 megabytes of available space after loading PhotoShop's 20 megabytes of program data; and at least 16 megabytes of RAM allocated to an application. Figure 1.3 shows another capability of PhotoShop where an area of an image which is selected and a copy of that selection was moved to another place in the image.

We implemented a project we called **Image Maker**. Our goal was to render many small images into an image efficiently. **Image Maker** was developed on the BeOS (Be Operating System). The BeOS is a multitasking, heavily multi-threaded system. The structure of the BeOS is optimized for dealing with real-time, high-bandwidth data types. Image Maker has a graphic user interface (GUI)[5], allows users to load different format image files, select unlimited number small images from these images, and render all small images into a new image.



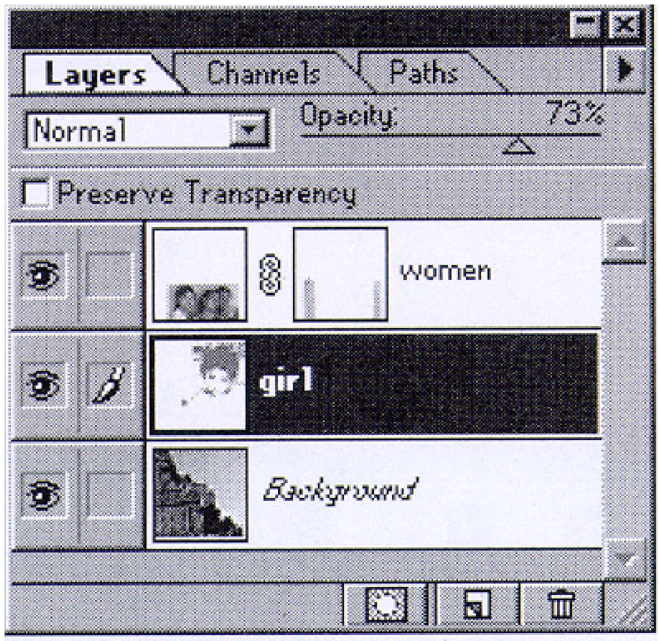


Figure 1.2: A PhotoShop screen.

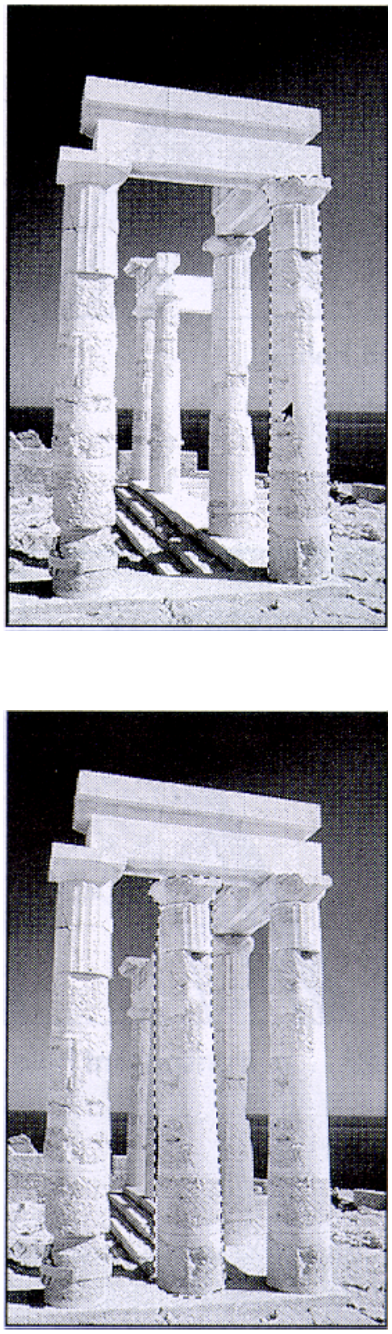


Figure 1.3: Drag and copy a selection

Chapter 2 discusses the method used in **Image Maker**. **Image Maker** consists of loading various format image files, selecting small images (tiles) from these images, storing these tiles in a Binary Search Tree (BST), rendering the BST into a new image, and editing the new image. The parallel algorithm used in the rendering portion of **Image Maker** is also explained in this chapter. Chapter 3 presents the functionality overview and menu options, and Conclusion and Future Works are presented in Chapter 4.

Chapter 2

Implementation

In this chapter, we explain the overall method of **Image Maker**. From loading image files to rendering the new image, an object-oriented design methodology was used throughout the whole design process of **Image Maker**. Figure 2.1 presents the model we used for **Image Maker**. First a window object receives all events. The window object then passes these events one a time to Bitmapview object. A simple message passing algorithm was used to implement the communication between Window object and Bitmapview object. Finally all events are handled in Bitmapview object.

2.1 Loading Image Files

There are many kind image files such as TIFF, GIFF, PEG, etc., and each kind of image file has a different format. Some formats have identifiers that are in the headers of image files, and some have the identifiers in the tail of image files, and some are in both the headers and the tails. In order to load a variety of image files, we must solve the various format problems.

The BeOS (Be Operating System) version 1.6.1 or later version provides the Be datatype library (`libdatatypes.so`) which deals with various data types. Using

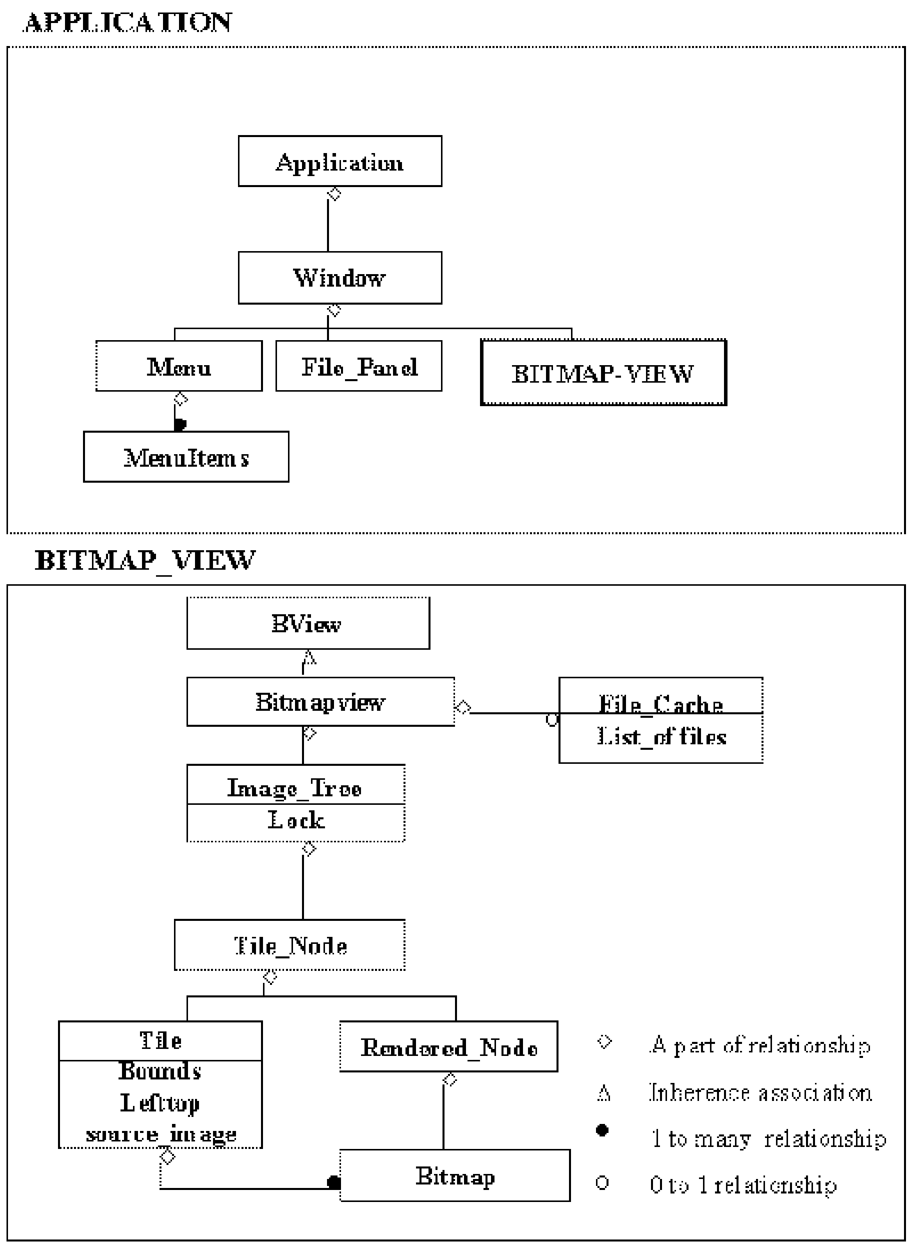


Figure 2.1: Image Maker Model.

`libdatatypes.so` the programmer can convert various image files such as "TIFF", "GIFF", "JPEG", "PEG", and a few others, into bitmap[7] objects. There are two APIs in this library: the program API and the Translator API. The program API is used by normal applications, and contains functions for identifying data and translating into another type that has been registered. The translator API is used by the writer of the data types, and allows the user to write the add-on "handlers" that the library dynamically loads to handle various data types.

In **Image maker**, all image files are converted into bitmap objects when they are loaded. The format of a bitmap file consists of a header followed by bitmap data. The header is big_endian and consists of the following data fields:

- `uint32 magic`; DATA_BITMAP
- `Rect bounds`; The bounds of the bitmap
- `uint32 rowbytes`; The number of bytes (not pixels) stored per row in the image data.
- `color_space colors`; either B_COLOR_8_BIT or B_RGB_32_BIT.

The data follows in the format specified by the `colors` member. Typically, the data is unsigned bytes for B_COLOR_8_BIT or RGBA_Interleaved unsigned bytes which uses 32 bits per pixel for B_RGB_32_BIT. The data is then stored row by row, from the top to the bottom of the image.

After converting an image into the bitmap format, a bitmap object is created to represent the data, and we can access the bitmap data through this bitmap object. We also have the capability of creating a bitmap object without initializing its data. The bitmap data can also be modified anytime after it is created.

2.2 Creating a New Tile

A tile is small image with rectangular bounds. It contains a bitmap object, which represents the small image, and some other information associated with it. Figure 2.2 shows all data information associated with a tile. In Figure 2.2, `filename` and `source_image` indicate the source data of a tile; `bounds` which is a rectangular shape gives the location and the area of a tile in the `source_image`, and `lefttop` gives the location of the upper left corner of a tile in the new image.

```
class Tile{
    public:
        Tile (char *filename, Rect * rect);
        Tile (char *filename);
        ~Tile();
        char  filename[20];
        Point  *lefttop;
        Rect   *bounds;
        Bitmap *image;
        Bitmap *source_image;
};
```

Figure 2.2: Tile.h.

When an image file is loaded, a tile is created only with the image filename, and the function `get_tile(Tile *&tile)` (see Figure 2.3) is called. This function checks the tile's `source_image` first. If the `source_image` is not found, a function using the filename reads the image file and then converts it into a bitmap object. If the tile's `bounds` are not given, the `source_image` is returned. Otherwise the tile's bitmap object is returned. This function is used to load an image file first and then get the tile data from that image when the tile's `bounds` are given.

After the `source_image` is displayed on the window, the user can drag any area within the window to define a tile. A rectangle displayed on the window traces the new tile's bounds when the user drags across the window using the mouse. After the mouse is released, the new tile's bounds are selected. If user does not like it, user can drag again and again until the desired tile is selected. The user can also move the selected tile to where they want. This movement changes the `lefttop` point of the tile. The user can then store the tile which will add it to the image tree. If user does not wish to save the current tile, the next dragging will cancel it. To save a tile, a new node (Figure 2.4) is created using the tile and is added into the BST by calling `insert(Tile_Node *new node)` (see Figure 2.3 Image_Tree.h). It is important to note that the tile's bitmap object still is empty until the BST is rendered (see Section 2.3).

2.3 Binary Search Tree (BST)

A Binary Search Tree (BST) is a special kind of tree. Each node in BST has a data item, and for a given node, the data items of the left subtree of that node are less than the data item in that node; and the data items in the right subtree of the node are greater than the data item in that node. Using a BST, we can achieve efficiency in both searching, adding, or deleting nodes[6].

Why did we choose a BST for use in **Image Maker**? We had many choices: arrays, lists, trees etc. When we set out to design **Image Maker**, we had a few things in mind for the data structure chosen:

- It had to store tile information.
- It had to keep the order in which the tiles were added to it.

- It had to be easy to add or delete a tile.
- It made searching for data fast.
- It was suitable for multi-threaded rendering.

We found that the BST was the best data structure to solve these problems. Another good feature of the BST is its ability to allow parallel rendering of a new image. Figure 2.3 shows the interface of our BST used in **Image Maker**. In this Figure, `root` is the root of BST; `lock` is a semaphore [8], is used for multi-threaded synchronization during the rendering of the BST; and `Rendered` indicates whether of the BST has been rendered.

In Figure 2.3, `Tile_Node` is the important data type which defines a node of our BST. The interface of a node is shown in Figure 2.4. Each node in the BST has a tile, an integer value, a `Rendered_Tile` object, and three `Tile_Node` pointers referencing the parent node, the left child node, and the right child node respectively. The integer `z_value` is the search key in BST, and represents the layer in a new image when a new node is created. The new node's `z_value` is greater than the previous node's `z_value`, and each node has a unique `z_value`. `Rendered_Tile` is an object which contains a bitmap object. The bitmap object holds the rendered image below this node. More detail of `Rendered_Tile` object will be covered in Section 2.4.

2.4 Rendering the BST

Rendering the BST means combining all the tiles created so far into one image. There are three phases in this process.

- Getting the tile's data from its source image. A tile is a new bitmap object. It has bounds and location in its source image. Figure 2.5 shows the algorithm of

```
class Image_Tree{
    public:
        Image_Tree();
        ~Image_Tree();

        void blend(Tile_Node *nd1, Tile_Node *nd2, int order);
        int  retrieve(int key, Tile_Node &search_Tile);
        int  remove(int key, Tile_Node *node);
        bool swap(Tile_Node *go_down, Tile_Node *go_up);
        void      insert(Tile_Node *node);
        void      reset(Tile_Node *node);
        Tile_Node *select_tile(Point *point);
        Bitmap    *show_Image(Tile_Node *node);
        bool      render(Tile_Node *node);
        Bitmap    *get_tile( Tile *tile);

        bool      Rendered;

    private:
        Tile_Node *root;
        Tile_Node *sel_tile;
        sem_id    lock;
};
```

Figure 2.3: Image_Tree.h

```
class Tile_Node {
    public:
        Tile_Node( Tile *newtile, int value);
        ~Tile_Node( );
        Tile          *tile;
        int           z_value;
        bool          Available_job;
        Rendered_Tile  *render_tile;
        Tile_Node     *parent;
        Tile_Node     *lift_child;
        Tile_Node     *right_child;
};
```

Figure 2.4: Tile_Node.h

how to select a tile's bitmap data from the source image into the tile's bitmap data.

- Combining the two tiles into one combined bitmap image. When a tile is saved, a new node, which contains that tile, is created and is added into the BST. A node has a `z_value` which indicates the layer of a tile in the BST. To combine two tile nodes, we first determinate which node holds the combined bitmap object based on `z_value`s in them. Then a new bitmap object is created in it. The bounds of the new bitmap object cover two tile's bounds; no bigger, no smaller. Figure 2.6 shows the size relationship between the bounds of the new object and the bounds of two tiles.

  Just like the first item in this list, we copy the data in the small `z_value` node into the new bitmap object's data, and then the tile's data in the big `z_value` node. If two tile bounds overlap, the upper tile shadows the lower one. The area which is not covered by the tiles is set to be transparent.

- Combining a tile with a combining image object, or two combined image objects. A tile or a combined image object contains a bitmap object, which we use when rendering the new image. Rending two such bitmap objects is like rendering two tile bitmap objects. The rendering algorithm is the same as the algorithm used in the previous item.

In summary, each node gets the tile image data from the tile's source image first, and then combines two nodes each time from the leaf up to the root of a BST until the root node is rendered.

```
Procedure get_tile

Input  : A Tile object: tile
Output : The tile's bitmap object if its bounds were given
          Otherwise the bitmap object of the tile's source image

begin
   if tile->source_image == NULL
       AppMakeMap( ..., tile->source_image, ...) // load image
   if tile->image == NULL
       if tile->bounds == NULL
           return tile->source_image
       else
           tile->image = Bitmap( tile->bounds, B_RGB_32_BIT)
           Height = tile->bounds->heigh()
           width1 = tile->image->BytesPerRow()
           width  = tile->source_image->BytePerRow()
           data1  = (unsigned char* )tile->image->Bits()
           data   = (unsigned char* )tile->source_image->Bits()
           for j = 0; j < Height; j++
               for i = 0; i < width1; i++
                   data1[(j*width1) + i] = data[(j +
                         tile->lefttop->y)*width +
                         tile->lefttop->x*4 + i]
            return tile->image
    else
        return tile->image
 end
```

Figure 2.5: Algorithm for selecting tile's data.

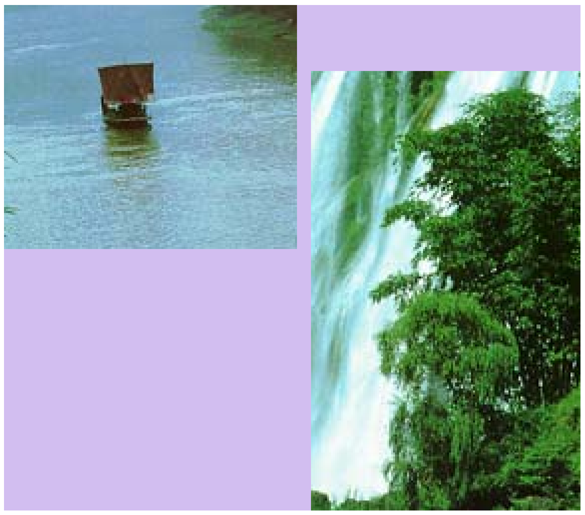


Figure 2.6: The bounds of a new bitmap object.

Once the BST is rendered, subsequent rendering of the tree does not have to render the whole BST. Since some subtrees are not changed and are rendered already, only the changed parts of the BST need to be rendered again. That is why we call the BST an image cache tree.

2.5 Editing the BST

Image Maker renders all the tiles in a BST into a combined image. If user wants to modify the new image, **Image Maker** provides the menu to edit the new image. The user can select a tile, change its location, or the user can delete it.

To modify a BST, the first thing is select a tile which the user wants to change. In Figure 2.3 , `select_tile(Point *p)` is used to search the nodes in the BST and find the tile which contains the point `p`. If such a tile is found, the node which contains that tile is returned. The parameter, `p`, of the `select_tile` function, is the mouse clicking point. The user can click anywhere on the window. If clicked on a tile, the tile is displayed. If two or more tiles contain the same point, only the top tile node is selected.

After a tile is selected, the user can change its location by using the mouse and clicking on it and dragging it to a new location or the user can delete it. All changes are written back into the BST.

Inserting or deleting a node changes the structure of the BST. Moving a selected node only modifies the tile's location, but all of these changes affect the new image. Each time the BST is modified, all the nodes on the path from the current modified node (include adding a new node) to the root are reset. This means that all the combined images in those nodes are removed. When re-rendering the BST, these are the only nodes that are rendering again.

2.6 Parallel Algorithm

Rendering a BST maps quite naturally to a parallel algorithm. Because the BeOS supports multi-threaded algorithm, we made our BST renderer multi-threaded. There are two problems which a parallel rendering algorithm must solve. The first is how to partition the rendering job and the second is how to communicate among threads[4], and these are discussed next.

2.6.1 Parallel Rendering Algorithm in BST

Rending a BST means combining all the tiles in each node into a new bitmap image. For a given node it may or may not have rendering job available now. There are three cases we must consider when dealing with nodes during rendering, they are:

- Fisrt, if a node is rendered, the node is called a rendered node. In this case, a rendering thread does not go on further down the tree because there is no rendering job below this node[1].

- The second case involves the rendered image in the node's left or right child node. But it may not be available now, in this case, the rendering thread skips the node and goes further to check the node's children. This node is called a waiting node.

- The third case involves a node which has a rendering job, but it has been taken by another thread. The current thread skips this node also. This type of node is also called a rendering node.

How does the rendering thread know whether a node has a job or not? Two variables are used to represent a node's rendering status. A variable called `Render_status` in `Rendered_Tile.h` has four values:

- 0 stands for the tile in a node that has not been combined with the combining image in either of the node's children.

- 1 indicates the tile in a node has been combined with the combined image in the node's left child node.

- 2 is similar to 1, except this node has been combined with the combined image in the node's right child node.

- 3 means that the node is rendered. There are not rendering job available below this node.

In cases 0, 1, and 2 the node can be waiting node or rendering node. But in case 3, the node is a rendered node. If a node is a leaf node, `Render_status` is assigned to 3 after the tile is selected from its source image and the combined image in the node is assigned with the tile's bitmap image.

Another variable, `Available_job`, is declared in `Tile_Node.h` with a `boolean` data type. This variable indicates whether a rendering job is available or not in a node, and is initialized to `true`. That means the rendering job available in that node. If the `Available_job` is `false`, the thread skips this node.

A variable called `Rendered` in `Image_Tree.h` indicates whether the whole BST is rendered or not. If the tree is completely rendered, the rendering threads are terminated. The problem of what to do when several rendering threads find a rendering job at a node at same time is covered in the next sub_section.

2.6.2 Multi-threaded Render

When the command to render the new image is issued, the main thread[2] spawns a drawing thread first, and then the rendering threads. The user can control the number of rendering threads via a GUI menu. After a rendering thread is spawned, it immediately checks the BST to find a rendering job by calling `render(Tile_Node *Ptr)` in `Image_Tree.h` (see Figure 2.3). Figure 2.7 gives the algorithm for this function.

When rendering with more than one thread there are some synchronization issues that must be resolved. To solve the synchronization problem a semaphore named `lock`, is used in `Image_Tree.h`. When the current rendering thread finds a rendering job in a node, it acquires lock first. After getting lock, the thread checks `Available_job` again. If it is still `true`, the node is marked. The thread then releases lock and then renders the node. If the `Available_job` has been changed into `false`, the thread releases lock and goes further on the tree to check another node. When finding a rendering job, the thread must first acquire the lock and then check `Available_job` again. In that way, the synchronization problem is solved.

The sequence of searching for rendering jobs for a thread is the same as traveling the BST in a pre-order traversed. The combining of tiles starts from the leaf nodes and moves up to the root. After the root node is rendered, rendering of the BST is done, and `Rendered` is set to `true`. The combined bitmap image in the root node is the new rendered image, all rendering threads exit, and the draw thread is woken up to draw the new rendered image. Figure 2.7 shows the renderer thread algorithm. The `Ptr` is the root of the BST or the root of a subtree in the BST that the user want to render.

```
Step 1. Getting the tile bitmap data:
    if Ptr->tile is NULL
        Ptr->tile->image = get_tile(Ptr->tile)

Step 2. Initializing the combined bitmap image in the node:
    if Ptr->render_tile is NULL
        Render_status = 0, 1, 2, or 3  // based on its children nodes
        Ptr->render_tile = Rendered_Tile(Render_status,...,Ptr->tile->image)

Step 3. Combining with the combined images in its children nodes:
    switch Ptr->render_tile->Render_status
        case 0
            if Ptr->left_child is rendered
                blend Ptr with Ptr->left_child
                Ptr->render_tile->Render_status = 1
                return false
            if Ptr->right_child is rendered
                blend Ptr with Ptr->right_child
                Ptr->render_tile->Render_status = 2
                return false
            break
        case 1
            if Ptr->right_child is rendered
                blend Ptr with Ptr->right_child
                Ptr->render_tile->Render_status = 3
                return false
            break
        case 2
            if Ptr->left_child is rendered
                blend Ptr with Ptr->left_child
                Ptr->render_tile->Render_status = 3
                return false
            break
        case 3
            return false

Step 4. Rendering its sub-tree
    if Ptr->left_child is not NULL and is not rendered
        Ptr = Ptr->left_child
        Goto Step 1
    if Ptr->right_child is not NULL and is not rendered
        Ptr = Ptr->right_child
        Goto Step 1
```

Figure 2.7: Algorithm for rendering a BST.

Chapter 3

Functionality

3.1 Overview

In this Chapter we use several figures, of different format types, to demonstrate the functionality of **Image Maker**. **Image Maker** was written in C++ and consists of approximately 3800 lines of code and was implemented on a BeBOX. The BeBOX is a machine consisting of dual 133 Megahertz 603e Motorola PowerPC Processors, with 32 Megabytes of RAM. **Image Maker** was implemented under version 1.6.1 BeOS. This OS supports different image file formats, and the formats we used to show the functionality of **Image Maker** are listed in Table 3.1.

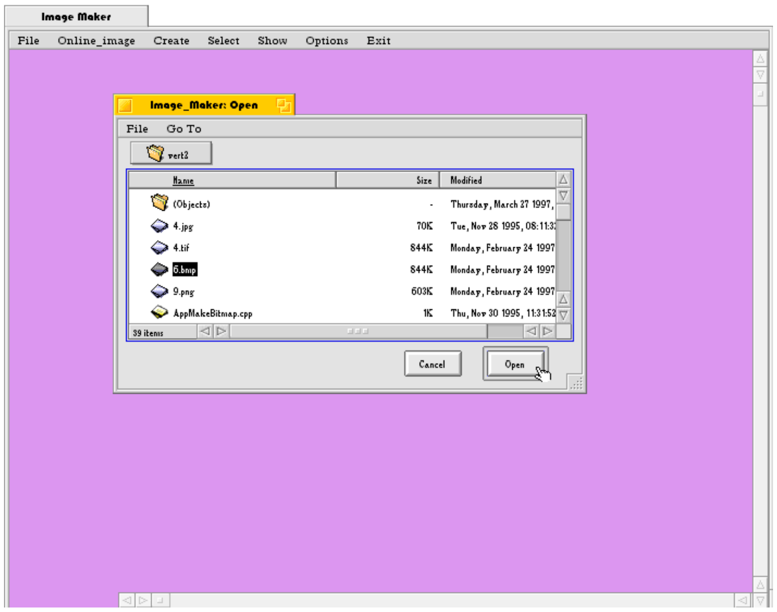
Filename	N1.tif	N2.jpg	N3.png	N4.gif	N5.bmp
File type	TIFF	JPEG	PEG	GIFF	BITMAP
Image size	630x457	640x420	630x461	780x480	630x457
Data size	864k	923k	617k	311k	864k

Table 3.1: Image files used in testing.

3.2 Options

- Loading image files.

  The user can load an image file in the current directory via **Image Maker**'s GUI. In the **Image Maker** Window, the user chooses `Open` option from the File menu.This is notated `File->Open`. Figure 3.1 shows the file panel which lists all the files in current directory pull down menu to load an image. The open folder allows the user to select files from different folders. Once you double click on an image file's name, the image is displayed in the window. Figure 3.2 shows two different format images loaded by **Image Maker**, and the fact that they are treated identically once they are loaded.

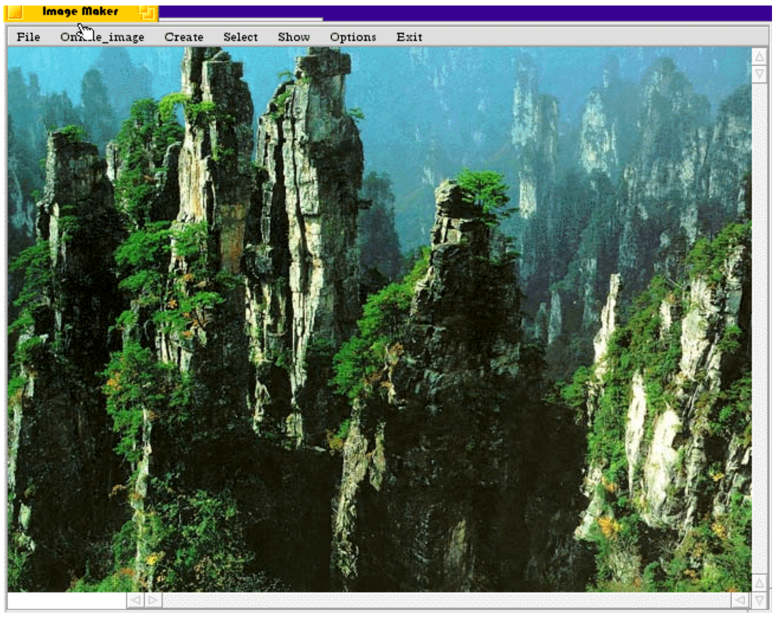


Name	Size	Modified
(Objects)	-	Thursday, March 27 1997,
4.jpg	70K	Tue, Nov 28 1995, 08:11:3
4.tif	844K	Monday, February 24 1997
6.bmp	844K	Monday, February 24 1997
9.png	603K	Monday, February 24 1997
AppMakeBitmap.cpp	1K	Thu, Nov 30 1995, 11:31:52

Figure 3.1: Open file panel.

- Selecting a tile, move it around the window, and save it.

  When an image file is loaded, the user can choose `Create->New` to select a new tile, and then drag a rectangular area using the mouse to select an interesting



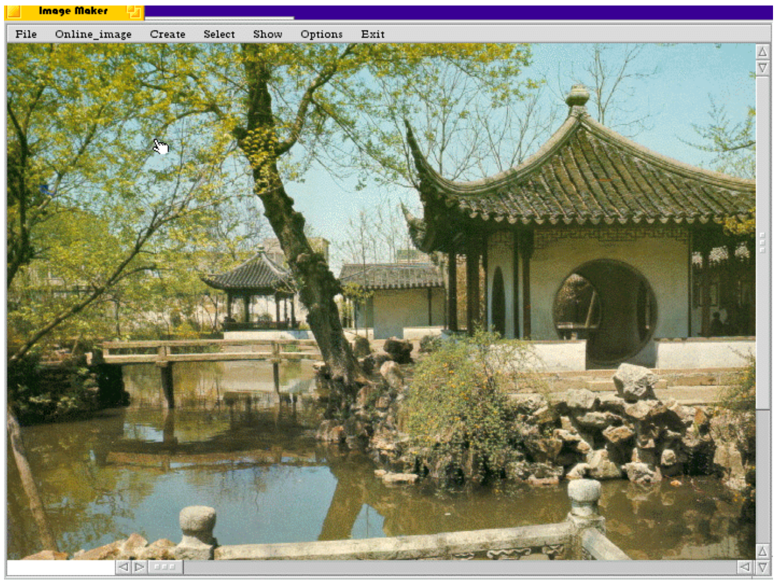


Figure 3.2: Images are loaded by Image Maker.

portion of the image. A rectangle is displayed that indicates the area which was selected. The user can drag any area in the window and can do this many times, but only the last area dragged is kept in the buffer. Figure 3.3 shows a tile that has been selected. After selecting a new tile the user can change its location. Choose the `Create->Move` menu option and then drag the tile to anywhere on the window. The new tile's location is changed when user moves it. Figure 3.4 shows the new tile which has been moved. To save the new tile, choose the `Create->Save` menu option. After the current tile is saved the user can select a new tile based upon the current image in the window.

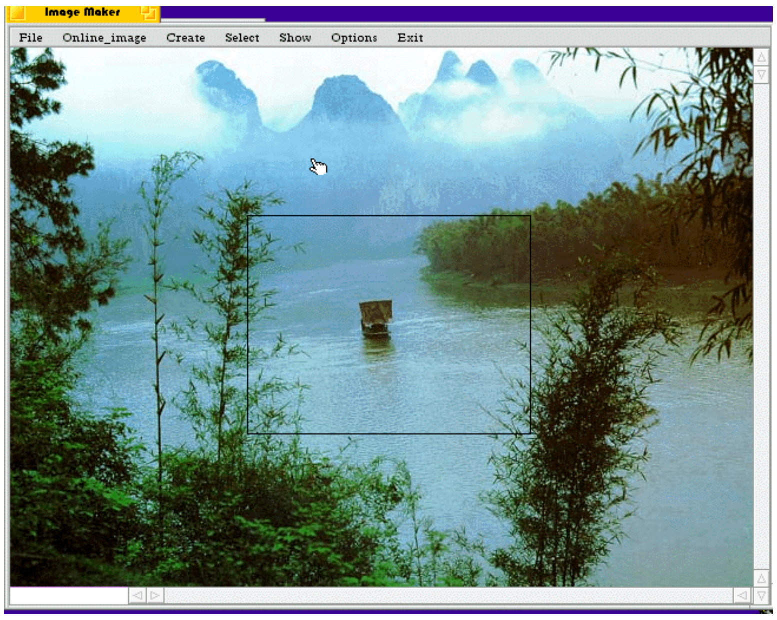


Figure 3.3: A new tile is selected.

- Displaying all the tiles which are saved in the BST.

  When a new tile is saved, a tile node is created and inserted into the BST. The user can choose the `Show->All_tiles` menu option and **Image Maker** will display each tile in the BST in the location where the tile was saved. Figure 3.5 shows the window which displays all tiles in a BST.

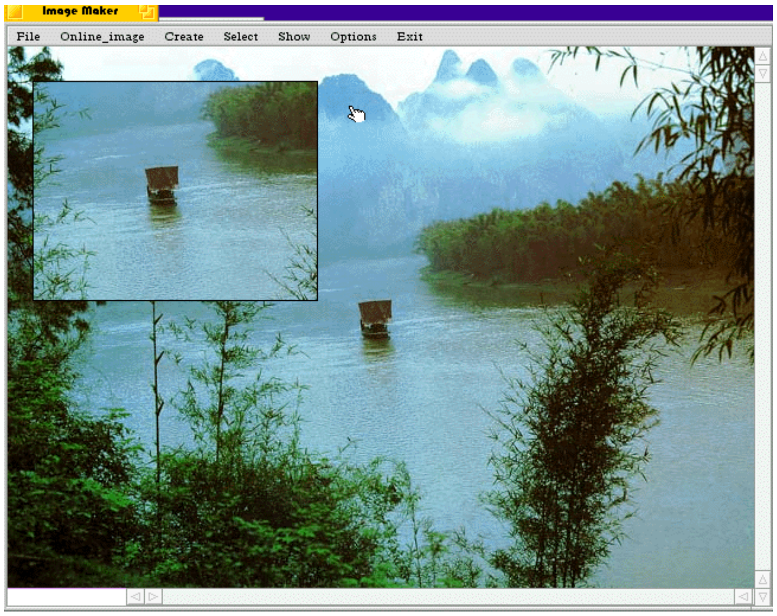


Figure 3.4: A new tile is moved.

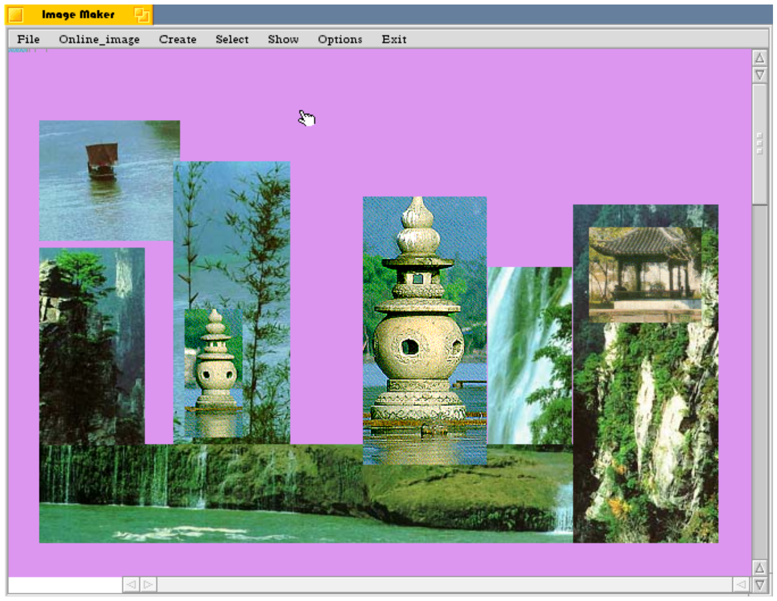


Figure 3.5: All tiles in a BST.

- Rendering the image.

  Rendering all the tiles in a BST into a new image is the main purpose of **Image Maker**. This new image is different from the images in Figure 3.5. Because all the tiles are combined into a new image and are not displayed separately. When the user select the `Show->Render->Image` menu option, **Image Maker** renders the tiles in the BST, and then displays the new combined image on the window. Figure 3.6 shows the new image created by **Image maker**.

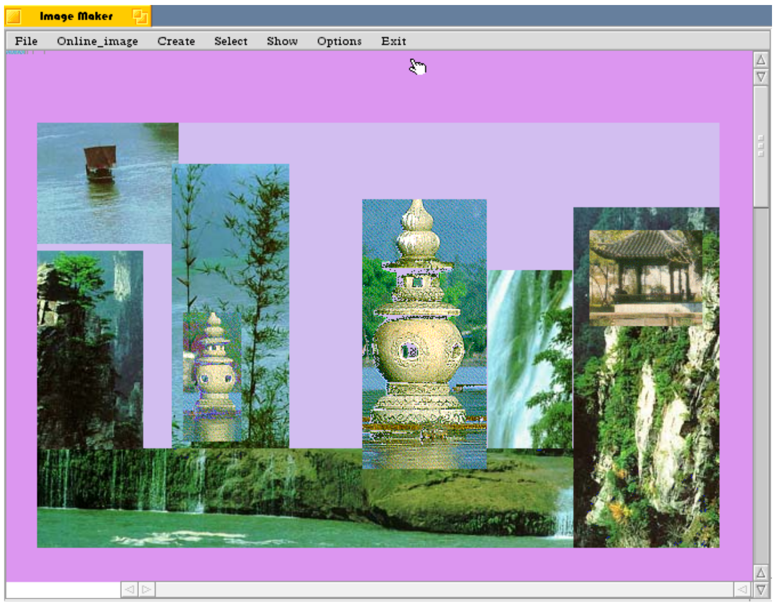


Figure 3.6: A new combined image.

- Modifying the new image.

  If the user is not satisfied with the new image, **Image Maker** allows the user to modify it. The image consists of the tiles in the BST, and editing the BST changes the image. The user can select any tile in the BST, and after a tile is selected the user can change its location or delete it. Figure 3.7 shows an image which is modified from the image in Figure 3.6.

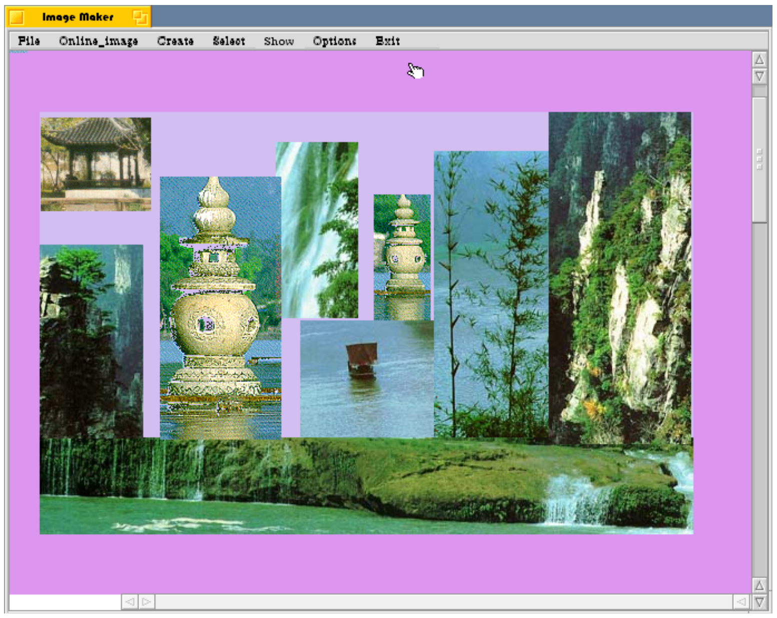


Figure 3.7: A new combined image modified.

- Render a subtree of a BST.

The default rendering of **Image Maker** is to render the whole BST. To render a subtree of a BST, the user must select a tile node. This node then becomes the root node of a subtree, and then the user can click `Show->Render->Subtree` menu option to render that subtree. Figure 3.8 displays a subtree of the image in Figure 3.7

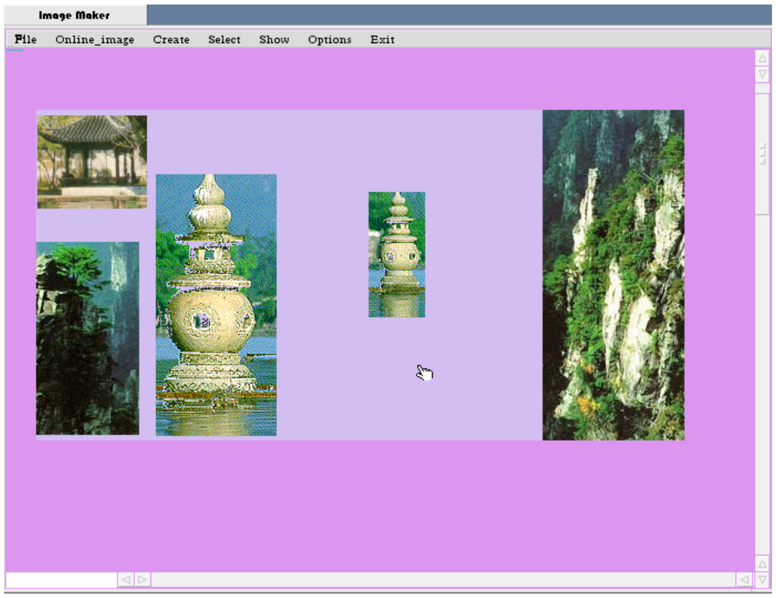


Figure 3.8: An image of a subtree.

Chapter 4

Conclusion and Future Work

4.1 Conclusions

We have developed and presented **Image Maker** which is an application that runs under the BeOS version is 1.6.1 or later. The BeOS for PC has just been released, and **Image Maker** also runs on it with no modifications. **Image Maker** allows the user to use available images, to create a new image, which consists of many tiles or small images selected from these original images. **Image Maker** is easy to use, and fast, and its GUI makes all the operations very clear and simple. **Image Maker** uses a single thread to render all tiles into a new image, but the user can also use multiple threads to render the image in parallel. We leave the number of threads as an option under the `Options` menu. A BST is used in **Image Maker** since it is very efficient for adding, searching, and deleting tiles. The user can render the whole BST or a subtree of the BST. Besides being efficient, the BST was selected because it naturally suits a parallel rendering algorithm. The operations of **Image Make** do not have a fixed order, and all the input is done by using the mouse.

4.2 Future Work

There are several things that we are still considering for future work. They are:

- Allowing the user to select any shaped area from an image instead of a rectangle. This would make the selection much more flexible.

- Allowing the size of a tile to be changed (such as enlarging or shrinking). This would be useful in the creation of much more specialized image.

- Using different blending algorithms when combining two bitmap objects. For example, mixing two bitmap objects gets an average pixel value bitmap object.

- Adding some image processing into **Image Maker** such as smoothing and sharpening .

- Modifying **Image Maker** to use it as a plug in a Game Maker or Animation Maker. This would be very beneficial for network tile-based games where only a portion of the image changes each move. Currently the whole image is transmitted over the network.

- Porting **Image Maker** to other operating systems.

- There is only one semaphore in `Image_Tree.h` for managing multi-threaded rendering. This has the potential to make multi-threaded rendering slow. We can put a semaphore in each node of a BST to speed this up, but OS's usually have limits as to the number of locks/semaphores available.

- Currently we keep the source images in a memory cache. If too many different image files are opened, that may use up the memory. We can delete the source image after we use it. This will cost a lot of time to load that image again when we reference it again.

Bibliography

[1] Robert Allen and Luigi Cinque. "A parallel algorithm for graphic matching and its maspar implementation." *IEEE Transaction on Parallel and Distributed system*, **8**(5):490–501, 1997.

[2] Aaron Cohen. *Win32 Multithreaded Programming.* O'Reilly & Associates, 1998.

[3] Michael Halvorson and Michael Young. *Running Microsoft Office97.* Microsoft Press, 1997.

[4] Vipin Kumar and Ananth Grama. *Introduction to Parallel Computing Design and Analysis of Algorithm.* The Benjamin/Cummings Publishing Company, 390 Bridge Parkway Redwood City, California 94065, 1993.

[5] Kevin Mullet. *Designing Visual Interfaces : Communication Oriented Techneques.* Prentice Hall, A Simon & Schuster Company Upper Saddle River, NJ 07458, 1995.

[6] George J. Pothering. *Introduction to Data Structures and Algorithm Analysis with C++.* West Publishing Company, 610 Opperman Drive P.O. Box 64526 St. Paul, MN55164-0526, 1995.

[7] Roger T. Stevens. *Quick Reference to Computer Graphics Terms.* Academic Press Professional, 955 Massachusetts Avenue, Cambridge, MA 02139, 1993.

[8] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, A Simon & Schuster Company Upper Saddle River, NJ 07458, 1992.

[9] The Be Development Team. *Be Developer's Guide.* O'Reilly & Associates, 1997.

[10] Elaine Weinmann. *Photoshop 4.* Peachpit Press, 2414 Sixth Street Berkeley, CA 94710, 1997.