

C Implementation of SmartForest Version 1.6.9

Matthew S. Davis
Department of Landscape Architecture
The Pennsylvania State University
msd4@psu.edu

August 7, 2002

Abstract

This document describes the architecture of the C implementation of SmartForest version 1.6.9. It also describes how to compile and run the code on an AIX 4.3 box and on Microsoft Windows. Compilation will be described first even though it is the secondary purpose of this document.

1 Introduction

SmartForest is a graphics application that renders scenes of trees on the computer screen. It reads ground elevation/slope information along with tree information from input files to render a pseudo-realistic image of the area that the user wants to see. If the user's input files are generated from programs that describe real-life places, then the image that SmartForest renders should look similar to the real-life location. SmartForest distributes trees randomly throughout the scene so the image that is rendered will not look precisely like the real-world location.

SmartForest has two implementations: one in C and one in Java. This document describes the C implementation. From this point forward, when we talk about the SmartForest implementation, we really mean the C implementation of SmartForest.

SmartForest is implemented under the X Windows system. X Windows is primarily used on computers that run the Unix operating system. It uses the Motif extension to X Windows to implement the application's windows and OpenGL to handle the drawings inside those windows. It currently compiles on AIX version 4.3. SmartForest will also compile on Microsoft Windows with the Exceed XDK library. The source code is identical when compiling to either AIX 4.3 or Microsoft Windows.

2 Compiling SmartForest

2.1 Compiling to AIX 4.3

SmartForest currently compiles with the assistance of an **Imakefile**. This compilation process has only been performed on the machine named `imlab-aix.larch.psu.edu`.

To generate the **Makefile** from the **Imakefile** on the machine named `imlab-aix.larch.psu.edu`, type:

```
lmake -l/usr/lpp/X11/lib/X11/config lmakefile
```

Once the **Makefile** has been generated, compilation is performed by typing:

```
make clean
```

```
make
```

The executable is called **SmartFOGL**. Once this file exists on the disk, the program is ready to execute.

2.2 Compiling to Microsoft Windows

Unfortunately the compilation process on Microsoft Windows is not as straightforward. The problem is that there are several additional libraries that are required when compiling to Microsoft Windows. These libraries are not installed in “standard” places. This makes describing the procedure more difficult. This procedure works on the machine with hostname 308D10.larch.psu.edu. When compiling on any other machine, the procedure may be complicated by having to determine if the libraries are installed on your local machine and, if they are, where they are located.

We compiled SmartForest using Microsoft’s Visual C++ 6.0 compiler. The project file is in C:\Program Files\Microsoft Visual Studio\MyProjects\sf. Take care to notice the include path and the library path in the project directory. These dictate where the compiler may find the libraries that SmartForest needs.

The following are the libraries that are required for SmartForest to link and run properly: mesagl.lib, mesaglw.lib, libjpeg.lib, HCLxm.lib, HcLXaw.lib, HcLXmu.lib, HcLXt.lib, xlib.lib and xlibgui.lib. The library locations for the Hummingbird Exceed XDK libraries will change when the full version of the libraries is purchased. The library directories entry in the SmartForest project will need to be updated when the evaluation XDK media is uninstalled and the full version of XDK is installed.

2.2.1 mesagl.lib

This library contains the OpenGL function calls for Microsoft Windows. It must be built by the programmer. The project file is in C:\tmp\mesa\MesaGL.

Note that this library was **modified** after it was downloaded to allow it to link properly with SmartForest. If you download a new version of Mesa, you will be required to modify the new version of Mesa as well. The linker will warn you of duplicate function definitions when you try to link SmartForest with the new version of the Mesa library. Simply comment the duplicate definitions out of the new version of the Mesa library and recompile the library. Run the linker again and the new Mesa library should link with SmartForest without complaint.

2.2.2 mesaglw.lib

This library contains the Mesa Workstation widget. It must be built by the programmer. The project file is in C:\tmp\mesa\MesaGLw.

2.2.3 libjpeg.lib

This library contains the routines that allow SmartForest to save its Forest View in a jpg file. It was supplied in the archive and does not need to be rebuilt. The library is in C:\tmp\jpeg\jpeg-6b.

2.2.4 HCLxm.lib

This library is part of the Hummingbird Exceed XDK. It is provided when Exceed XDK is installed and does not need to be rebuilt. The library is in C:\Program Files\Hummingbird Evaluation Media\7.10\Exceed XDK\Program Files\Hummingbird\Connectivity\7.10\Exceed\XDK\lib.

2.2.5 HcLXaw.lib

This library is part of the Hummingbird Exceed XDK. It is provided when Exceed XDK is installed and does not need to be rebuilt. The library is in C:\Program Files\Hummingbird Evaluation Media\7.10\Exceed XDK\Program Files\Hummingbird\Connectivity\7.10\Exceed\XDK\lib.

2.2.6 HclXmu.lib

This library is part of the Hummingbird Exceed XDK. It is provided when Exceed XDK is installed and does not need to be rebuilt. The library is in C:\Program Files\Hummingbird Evaluation Media \7.10\Exceed XDK\Program Files\Hummingbird\Connectivity\7.10\Exceed \XDK\lib.

2.2.7 HclXt.lib

This library is part of the Hummingbird Exceed XDK. It is provided when Exceed XDK is installed and does not need to be rebuilt. The library is in C:\Program Files\Hummingbird Evaluation Media \7.10\Exceed XDK\Program Files\Hummingbird\Connectivity\7.10\Exceed \XDK\lib.

2.2.8 xlib.lib

This library is part of the Hummingbird Exceed XDK. It is provided when Exceed XDK is installed and does not need to be rebuilt. The library is in C:\Program Files\Hummingbird Evaluation Media \7.10\Exceed XDK\Program Files\Hummingbird\Connectivity\7.10\Exceed \XDK\lib.

2.2.9 xlibgui.lib

This library is part of the Hummingbird Exceed XDK. It is provided when Exceed XDK is installed and does not need to be rebuilt. The library is in C:\Program Files\Hummingbird Evaluation Media \7.10\Exceed XDK\Program Files\Hummingbird\Connectivity\7.10\Exceed \XDK\lib.

3 Running SmartForest

The SmartFOGL executable has several optional parameters. The first is a parameter that affects X-Windows. Valid values for this parameter are `-installc` and `-autopref`. The remaining five parameters are file names.

The command that runs the program is:

```
SmartFOGL x_parms dem_mapname stand_name treelist_name stemlist_name trans_name
```

where

<code>x_parms</code>	May be <code>-installc</code> or <code>-autopref</code> or both.
<code>dem_mapname</code>	Contains ground height data. Has an extension of <code>.elev</code>
<code>stand_name</code>	Defines stand identifiers on a grid-by-grid basis. Has an extension of <code>.stnd</code>
<code>treelist_name</code>	Maps trees to stands. Has an extension of <code>.trl</code>
<code>stemlist_name</code>	Defines stand characteristics. Has an extension of <code>.s1f</code>
<code>trans_name</code>	Translation map from 26-char stand id to 32-bit integers. Has an extension of <code>.trans</code>

The parameters may be specified in any order. If the files exist and are formatted properly, SmartForest will display the Map View screen and the Forest View screen. The application is now considered running.

Parameters 2–5 have default file names. The user need only provide these parameters on the command-line if they want to override the default file names. The default file names are `smartforest.elev`, `smartforest.stnd`, `suppose.trl` and `suppose.s1f`. The last parameter does not have a default name and must be explicitly listed on the command-line when the user wants a translation file. If a parameter ending in `.trans` is not specified, then SmartForest assumes that all of the stand ids in the `.s1f` file will fit properly into a 32-bit integer.

SmartForest is designed to work in tandem with INFORMS. There are two environment variables that, if set, help SmartForest determine where to find the input files. The two environment variables are `INFORMS_LOCALGIS` and `INFORMS_LOWPROJID`. These two environment variables, when supplied, are concatenated together and used as the subdirectory that stores the input file names. For example, if `INFORMS_LOCALGIS = /usfs/home` and `INFORMS_LOWPROJID = /jmarston`, then SmartForest will look in `/usfs/home/jmarston` for all of the input files. If these two environment variables are not set, then SmartForest will look for the input files in the same subdirectory where the executable resides.

4 Source Code

This section focuses on the primary purpose of the document: here an attempt is made to describe the source code. The most logical place to start is with the source code files themselves.

A quick note on terminology: code that is compiled in a program but is never called is said to be *dead code*.

4.1 Files

The source files that compose the executable are described here. The files are presented in alphabetic order and not in “program-logic” order since it’s easier for the reader to find specific files when they are in alphabetic order. Another reason for this order is that the “program-logic” of the executable is examined in another section. See Section 4.2.

4.1.1 `callback.c`

This file contains the graphics callbacks for the X Windows environment. Most events in the map window and the forest window that require a callback from X Windows have their handler in this file. Examples of events would include: clicking on a new location in the map window, dragging with the left mouse button in the forest window to change the elevation, dragging with the middle mouse button in the forest window to move the current location, dragging with the right mouse button to rotate the view, etc. Note that menu selections are not handled by functions in this file even though they are considered callbacks. Menu selections are handled in `menu.c`.

According to the comment in this file, the main purpose of the functions in this file are to maintain the proper aspect ratio and cross hair location in the map window. The comment continues saying the file also tries to keep the map centered in the window. The comment is erroneous. These abilities are really implemented in `map.c`.

4.1.2 `datamap.c`

This file contains the routines that manage the datamap dialog box. This is the dialog box that appears when the user chooses File-Edit Color... from the map window. The creation of the dialog, its size, the final placement of all of the buttons, etc. are controlled in this file.

The contents of the Legend dialog and the Stand Legend dialog are also defined in this file. Note that only the **contents** of the Legend and the Stand Legend are in this file. The creation of those dialog boxes and their associated callbacks can be found in `legend.c` and `standLegend.c` respectively.

4.1.3 `debug.c`

This file only contains the `debug` method. This method is used to print debug information to the screen. It is used by the programmers and is not used in the production executable.

4.1.4 `demfile.c`

This file is responsible for reading in the `.elev` file. This happens in the `read_DEMFile` routine. This routine first calls `elevReader` in `elevReader.c`. The `elevReader` routine does the actual reading of the `.elev` file and initializes the `elev` variable with this information. When control returns to `read_DEMFile`, the `map` variable is then initialized with its values. The `map` variable’s values are set using `elev`’s values. The `map` variable is very important since it is used to create all of the pixel maps that are used by the program. The pixel maps are responsible for changing the colors on the land in the Forest View to reflect the selected stand feature.

This file, along with `standInterface.c`, also performs automatic memory management for the pixel maps. Each pixel map is approximately 12 megabytes large. Currently there are 7 different pixel maps. To help keep memory usage reasonable, these two files keep track of how many times each map is used. If the program runs

out of memory when it tries to allocate memory for a pixel map, it first releases the currently allocated pixel maps, one at a time, starting with the least used map until it has enough memory to allocate the new map. We expect that only one map will need to be released since all of the maps have the same size. The routines that coordinate this effort are located in these two files. As more pixel maps are added to the application (due to the addition of stand features) a maximum limit may need to be imposed on the number of pixel maps permitted to be active in memory.

4.1.5 `draw.c`

This file contains the routines that draw the Forest View window. Routines to draw the ground, trees and houses are here. For efficiency purposes, there are two routines to handle drawing trees. The high resolution routine draws both the tree stems and their crowns. It also draws all of the trees in each grid. The low resolution routine draws the tree stems only. The low resolution routine draws a maximum of five trees per grid. While there are two routines to draw the scene in the Forest View, the application only uses the high resolution routine. The low resolution routine exists to help the application's animation performance and was used in previous versions of the application. Performance has increased to the point where the low resolution routine is no longer necessary. The low resolution routine still exists but it is never called in the application.

4.1.6 `editProperties.c`

This file is not currently compiled with SmartForest. It contains the beginnings of an implementation of a new window to edit tree properties. After the June 2002 meeting with the US Forest Service in South Dakota, this item was tabled. The code remains for possible future completion. Note that this code will not run as written. It requires significant attention before it can be considered finished. Also note that this file is not yet in CVS.

4.1.7 `elevReader.c`

This file contains the routine that reads the `.elev` file. It utilizes the primitive scanner defined in `tokenizer.c` to help it read the file. The data read from the file is stored in the `elev` variable. The `elev` variable is defined in this file. All of the access routines to the `elev` variable are also in this file. Note that `elev` is not directly accessible outside of this file — outsiders wanting access to `elev`'s data must use the accessors.

4.1.8 `file.c`

This file contains the functions that manage the dialog boxes used for several of the options under the File menu. The options whose dialog boxes are managed here are File-Save Image As, File-Load New TreeList, File-Load Preferences and File-Save Preferences.

4.1.9 `g.c`

This is the file that initializes the application. `main` is located in this file. I have no idea why the file's name is `g.c`.

4.1.10 `infopage.c`

This file contains the functions that manage the dialog box that displays tree information when the user left clicks on a tree in the Forest View while holding the shift key. This dialog box has the title "Tree Information".

4.1.11 legend.c

The routines that manage the legend dialog box that appears when the user chooses View-Show Legend are defined here. Note that the contents of the legend (the data ranges, the colors of each range and the titles) are not defined in this file; they are defined in `datamap.c`. The routines in this file simply manage the dialog box.

4.1.12 map.c

The routines that draw the map in the Map View window are located here. The comment at the beginning of the file describes four coordinate systems that are used to maintain the map window:

1. The first is the physical coordinates of the window. The units for this coordinate system is pixels.
2. The second is the viewport coordinate system that is used by OpenGL to do the actual drawing. The viewport is normalized to $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$ where the origin is the center of the physical window.
3. The third is the coordinate system defined by the size of the elevation map. The size is given by the global (ugh) variables called `columnCount` and `rowCount`. The comment in the file says, “This is a ‘dimensionless’ grid of uniform points with equal spacing between the points.” It seems to really be a two-dimensional grid. I’m assuming the author of the comment was attempting to say that the units of the spacing on the grid are not clearly defined. Maybe. The comment isn’t clear. In any case, the dimension of the grids is meters.
4. The comment from the previous item continues, “To be drawn in the window, [the grid] needs to be scaled and positioned in the window. This will be referred to as the SCALED MAP.” The scaled map value is the map translated into the window coordinate system, in pixels, using the global (ugh again) variables called `zoomx` and `zoomy`.

This file contains the routines to draw the map and to resize it and thus maintain the proper aspect ratio. Routines that convert between coordinate systems are also in this file. The drawing of the red cross hairs is also performed here.

4.1.13 menu.c

The creation of the menu bar found in the Map View is performed in this file. The callbacks for each menu selection are set up in this file but most of the actual code for the callbacks is distributed among the other source files. A few of the callbacks are defined in this file, however — such as the `help` menu’s callbacks.

The global (yuck) variable `forest_dispmode` is maintained (but not defined!) in this file.

4.1.14 property.c

The routines that manage the dialog box for editing datamap properties are in this file. This dialog box appears when the user selects File-Edit Properties. The loading and saving of the datamap preferences file and the writing of a jpeg file (File-Save Image As) are performed here.

4.1.15 server.c

These routines compose the remote procedure call server to handle “remote SmartForest”. The interface with Aspect RPC (remote procedure call) is handled here. The only interaction from SmartForest to the server is the call to `rsf_init_server` from `main`. All other interaction is from the server to SmartForest. Unless the Forest Services chooses to not use INFORMS, this file is no longer required. INFORMS does not use RPC to interact with SmartForest. It calls the executable directly.

4.1.16 slfReader.c

The routines here are responsible for reading the `.slf` file and storing the data in the `standInfoArray` variable. Effectively `standInfoArray` acts solely as the data structure containing the raw data read from the file. The application later builds other data structures from `standInfoArray` that are actually used as the application's data abstractions. One might consider `standInfoArray` as a file interface and nothing more.

The data read from the `.slf` file is stored in `standInfoArray` and then sorted based on the stand identifier. The access routine `findStandInfo` then uses binary search to find the record corresponding to the stand identifier passed as a parameter — hence the sorting.

4.1.17 standInterface.c

This file is the cornerstone of the internal database for the application. Here all of the data that is read from the input files is built into the data structures that the application actually uses to perform its tasks (these are called the *data abstractions* of the application). All of the other “helper” data structures are simply used as an interface to the data from the files.

This file also assists `demfile.c` in automatically managing the memory used by the many large pixel maps. See Section 4.1.4 for more information. In addition to memory management of pixel maps, this file also generates all but one of the pixel maps when they are needed. The pixel maps correspond to stand feature information since they determine the color of the patches on the ground in the Forest View. Hence it is likely that for each stand feature, an additional pixel map will be required. With so many pixel maps, the need for automatic memory management of these maps is no longer a question; it's required.

4.1.18 standLegend.c

This file contains the routines that manage the stand legend dialog box. This dialog box appears when the user chooses **View-Show Stand Legend**. Like the legend dialog box, the ranges, colors and titles used in this dialog box are defined in `datamap.c` and not here.

4.1.19 standmap.c

This file contains the routines that manage the stand map dialog box. This dialog box has the title “Tree List Map”. This dialog box appears when the user chooses **View-Show Tree List** or when the user left clicks on a tree in the Forest View while holding the shift key (which displays the dialog box described in `infopage.c`) and then clicks on the Show Tree List button.

4.1.20 texture.c

This file handles everything to do with texture mapping the Forest View.

4.1.21 tokenizer.c

This file implements the scanner for the application. The scanner is responsible for reading data from a file and creating tokens out of the data that are easily recognizable by a parser. The parsers that are used in the application are all very simple. Hence the token is also very simple; it's represented by a single `struct`. The scanner is stateful and may only be used with one parser at a time. In other words only one file can be scanned at a time. There are only two parsers that utilize this scanner: the `elevReader` that reads `.elev` files and the `trlReader` that reads `.trl` files. Since the scanner is stateful, parsers that use it must make sure to call `initTokenizer` when starting to scan a new file and to call `finalizeTokenizer` when scanning of the file is complete. The latter is more of a concern as it frees up the memory that the scanner is consuming. It turns out that the scanner consumes quite a large amount of memory as it first reads the entire file into memory and then picks off tokens as the application progresses.

4.1.22 `translateStandId.c`

The routines that abstract translation of 26-character stand ids to 32-bit integers occurs here. When stand ids are read from `.slf` files, they might be larger than a 32-bit integer can hold. If the user specifies a file with an extension of `.trans` as a command-line parameter when the application is started, then that file contains a mapping from the 26-character stand ids to unique 32-bit integers. The `translateStandId` routine is responsible for taking a stand id as input and either returning the duplicate 32-bit integer that the stand id actually holds (in the case where no `.trans` file is specified), or it returns the 32-bit integer from the `.trans` file that corresponds to the stand id that was input. Thus the routines that utilize `translateStandId` have no idea whether translation is occurring or not. That's good.

4.1.23 `treeListReader.c`

The routines here are responsible for reading the `.trl` file and storing the data in the `headTreeList` variable. Effectively `headTreeList` acts solely as the data structure containing the raw data read from the file. The application later builds other data structures from `headTreeList` that are actually used as the application's data abstractions. One might consider `headTreeList` as a file interface and nothing more.

The data read from the `.trl` file is stored in `headTreeList`. The access routine `findTreeList` finds the record corresponding to the stand identifier passed as a parameter.

4.1.24 `utils.c`

This file contains miscellaneous helper functions. Some of the helper functions do graphics calculations and graphics initialization.

4.1.25 `widgets.c`

This file contains code to help the application define dialog boxes. Several callbacks are defined here.

4.1.26 `win_utils.c`

This file contains window utilities for GLX, an API that helps X Windows and OpenGL to work together.

4.1.27 `XmLib.c`

This file contains convenience utilities to assist the X Windows code.

4.2 Program Execution & Data Structures

SmartForest is an X Windows application. It is therefore inherently an *asynchronous* application. Program control does not simply start in `main()`, run sequentially and then end. Instead, `main()` “bootstraps” the program by reading the input files, preparing a few internal data structures, displaying the Map View and the Forest View windows and creating a map for the windows manager that associates certain events (such as clicking the left mouse button, pressing the ESC key, etc.) with the corresponding functions to execute in SmartForest. It then issues a call to `XtAppMainLoop`. `XtAppMainLoop` is an infinite loop that dispatches events to their handlers. The call to `XtAppMainLoop` registers the current SmartForest process with the windows manager. Metaphorically speaking, the windows manager takes control of the program from `main()` so `main()` does not need to run further. Any further actions that are performed by the user are first interpreted by the windows manager using the map that was created earlier in `main()`. The windows manager calls the appropriate SmartForest function to handle the user's action. The term *callback* is used to refer to these asynchronous functions that the windows manager calls to handle the user's actions. The term *event* is used to describe the user's actions. `main()` has nothing to do with any events that occur after control has been passed to the windows manager. This is the architecture that all X Windows programs utilize.

The remainder of this section is organized as follows: We first describe `main()` in more detail than appears above. The sequence of initialization will be discussed, however the fine details of each data structure will be delayed — details are presented in each data structure's section that follows the discussion of `main()`. A discussion of the callbacks that handle the bulk of SmartForest's capabilities ends the section.

4.2.1 `main()`

`main()` is located in the file named `g.c`. It performs the following sequence of operations:

1. The random number generator is seeded with the process identifier.
2. Process the optional command-line arguments.
3. Several X Windows arguments are prepared to create the main window. When the arguments are ready, the main X Windows application window is created. Note that the window's contents are not yet drawn.
4. The mapping between events and callbacks is established.
5. Initialize the data elements in the legend and in the stand legend.
6. The Data Preferences file is loaded. The file name is `sf.prf` where the directory is relative to the directory where SmartForest is run. The subdirectory may be overridden by setting the `SMARTFOREST_APPDIR` environment variable. Only the directory may be overridden. The file's name (`sf.prf`) may not be overridden. (This should be changed so that this file is handled consistently with the other files.)
7. Determine the size of `datamap`'s and `standDatamap`'s entries. Update each entry's `length` field with the size.
8. The `.elev` file is parsed. See Section 4.2.2.
9. The `.trl` file is parsed. See Section 4.2.3.
10. The `.slf` file is parsed. See Section 4.2.4.
11. The default image file name that is used when the user selects `File-Save Image As` is set to `./ImageFiles/default.jpg`. The directory may be overridden by setting the `SMARTFOREST_PROJDIR` environment variable. Again, only the subdirectory may be overridden. The file name, `default.jpg`, may not be overridden at initialization time. However, this file name may be overridden in the dialog box that appears when the user selects `File-Save Image As`. (This should be changed so that this file is handled consistently with the other files.)
12. The `.stnd` file is parsed. See Section 4.2.5.
13. The headers of the `.elev` and `.stnd` files should be identical for SmartForest to run properly. Verify that they are the same.
14. The `normals` data structure is initialized. According to the comment in the function `initialize_normals()`, these values are used to “compute where the person stands when they are in the middle of a dem grid. . . Initially, `map[a][b]` is the (x, y, z) position in space of the DEM sample at (a, b) .” The `map` structure is initialized when the `.elev` file is parsed, above. See Section 4.2.2. “[We] later mangle `map[a][b][0]` and `map[a][b][1]`, but leave `map[a][b][2]` alone, using it as the height of each DEM sample.
 - `normals[x][y][0]` is the increase in height from the DEM sample at (x, y) to the DEM sample at $(x + 1, y)$ times cell size.
 - `normals[x][y][1]` is the increase in height from the DEM sample at $(x + 1, y)$ to the DEM sample at $(x + 1, y + 1)$ times cell size.
 - `normals[x][y][2]` is the increase in height from the DEM sample at $(x, y + 1)$ to the DEM sample at $(x + 1, y + 1)$ times cell size.
 - `normals[x][y][3]` is the increase in height from the DEM sample at $(x, y + 1)$ to the DEM sample at $(x + 1, y + 1)$ times cell size.
 - `map[x][y][0]` gets the increase in height from the DEM sample at (x, y) to the DEM sample at $(x + 1, y)$ divided by cell size.

- `map[x][y][1]` gets the increase in height from the DEM sample at (x, y) to the DEM sample at $(x, y + 1)$ divided by cell size.”
15. The colors that are used to draw the ground, water, roads, fields, trees and tree stems are initialized. Each of these is implemented as a two-dimensional array: the first dimension varies the color based on the height of the object and the second dimension represents the RGB values at that height.
 16. If the user specified `-installc` as a command-line option, then a private color map is used for the application. Note that this command-line option is currently not permitted as our check for the number of arguments only allows for the data file names.
 17. The contents of the windows are finally drawn.
 18. Initialize the map message and the stand message. These show the current mode of the map and the stand, respectively.
 19. A remote-procedure-call server is initialized (AIX only).
 20. Finally, `XtAppMainLoop` is called and `main()` terminates.

4.2.2 The .elev File

Parsing the `.elev` file begins in the function named `read_DEMFile`. `read_DEMFile` first calls `elevReader` where the actual reading of the file occurs. After the file is read by `elevReader`, control returns to `read_DEMFile`. `read_DEMFile` continues by setting some global (yuck) variables based on the data read from the `.elev` file.

The `elevReader` function utilizes the scanner provided in `tokenizer.c`. It creates an encapsulated variable called `elev`. `elev`'s contents are accessible using the accessor functions provided in `elevReader.c`. The first six values comprise the header of the `.elev` file. `elev` is formatted as follows:

Never Used	Data Type	Field Name	Purpose
	int	nCols	The number of columns in the DEM file.
	int	nRows	The number of rows in the DEM file.
	int	xllCorner	The x-component of the south-west corner.
	int	yllCorner	The y-component of the south-west corner.
	int	cellSize	The width and height of each grid. The value is often 30. (Meters)
	int	noDataVal	The value used to represent “No Data”
	float[][]	elevInfo	The elevation information. Each datum represents the height of the sample. The units are not specified but are likely feet. The data is read from the file in row-major order.

Once `elev` is initialized, its values are used to initialize the global variables that are actually used by the application. In this light, `elev` may be viewed solely as an interface to the contents of the file while the global variables are the data abstractions used by the application. The variables are as follows:

Global Variable Name (Application Abstraction)	Initialized by elev Field (File Interface)
<code>columnCount</code>	<code>nCols</code>
<code>rowCount</code>	<code>nRows</code>
<code>x_sw</code>	<code>xllCorner</code>
<code>y_sw</code>	<code>yllCorner</code>
<code>x_ave</code>	<code>cellSize * nRows</code>
<code>y_ave</code>	<code>cellSize * nCols</code>
<code>map[col][nRows - row - 1][0]</code>	0
<code>map[col][nRows - row - 1][1]</code>	0
<code>map[col][nRows - row - 1][2]</code>	<code>elevInfo[col][row]</code>

Notice that the initialization of `map` reverses the order of the rows in `elevInfo`.

4.2.3 The .trl File

The function called `trlReader` is responsible for parsing the `.trl` file. It utilizes the scanner found in `tokenizer.c`. The `.trl` file is formatted like the output of a database query. Thus `trlReader` is equipped to detect when it sees the periodic header information that the query reports. Once detected, the periodic header is simply skipped.

The algorithm that this parser follows is given below:

1. Initialize `invalidTree` to `FALSE`
2. While there is still data in the file
 - (a) Skip the periodic header
 - (b) Read the stand identifier (`standID`)
 - (c) Determine whether a stand already exists for this stand identifier. If so, use it. Otherwise, create a new stand for the stand identifier. For some reason, the stand is represented by the `TreeList` data type. It's an awful name since the stand contains, among other things, a linked list of trees represented by the `Node` data type. Holy terrible names.
 - (d) While we haven't detected the new periodic header
 - i. We want to create a new empty `Node` to hold the current line of data. This new `Node` will be at the head of the stand's linked list of trees. There are three cases when obtaining the "empty" `Node`:
 - If we have detected an invalid tree via `invalidTree` and if the stand has a non-empty linked list of trees, then use the current head of that list as the "empty" `Node`. Note that the data currently in that `Node` will be overwritten.
 - If we have detected an invalid tree via `invalidTree` and if the stand has an empty linked list of trees, then create a new `Node`, add it as the new head of the stand's linked list and use the new `Node` to hold the current line's data.
 - If we did not detect an invalid tree via `invalidTree`, then create a new `Node`, add it as the new head of the stand's linked list and use the new `Node` to hold the current line's data.
 - ii. Parse the current line into the `Node`. The fields are described in the table below.
 - iii. If the value read for the `CURRHT` field is < 5 , then the tree is considered invalid. Otherwise it is considered valid. `invalidTree` is set accordingly.
 - iv. If we reach the end of the file here, simply fall out of the inner while loop. The outer while will subsequently detect that we are at the end of the file and end nicely.

The information that is stored for each tree in the stand's linked list is as follows:

Never Used	Data Type	Field Name	Purpose
X	int	TREENUMBER	The first four digits are the plot number. The last three digits are the tree number on the plot.
X	int	TREEINDX	Forest Vegetation Simulator's (FVS) primary key for this record.
	char[5]	SPCD	Species code. These are different for the different geographic variants of FVS.
X	int	SPNO	Species number. These are different for the different geographic variants of FVS. The treelist file is sorted by this column and secondarily by TREENUMBER.
	int	TRCL	Tree class. 1==desirable, 2==acceptable, 3==poor
X	int	SSCD	Special status code. User defined, no inherent meaning.
X	int	PNTNUM	Point (or plot) number. The point is the basic unit in the sampling design.
	float	TREESPERACRE	Trees per acre.
	float	MORTALPERACRE	Number of trees per acre that went to tree heaven during the cycle.
	float	CURRDIAM	Current diameter (inches)
	float	DIAMINCR	Diameter increment (inches). This is the change in diameter during the cycle.
	float	CURRHT	Current height (feet). Must be ≥ 5.0 .
	float	HTINCR	Height increment (feet). This is the change in height during the cycle.
	int	CR	Crown ratio—crown length as a percentage of total tree height.
	float	MAXCW	Maximum crown width. This is the diameter of the crown at the longest axis. No units given (assumed feet).
	int	MS	Hawksworth mistletoe rating code. Range 0–6. 0==uninfested, 6==completely/heavily infested.
X	float	BATILE	Basal area percentile. This is the ranking of that tree record in terms of the total basal area of the stand.
X	int	POINTBAL	Point basal area larger. This is the total basal area per acre represented by tree records from the same plot (or point) that have a larger diameter than that record.
	float	TOTCUFTVOL	Total cubic foot volume per tree.
	float	MCHCUFTVOL	Merchantable cubic foot volume per tree.
	float	MCHBDFTVOL	Merchantable board foot volume per tree.
	int	MCDF	Merchantable cubic foot defect per tree.
	int	BFDF	Merchantable board foot defect per tree.
X	int	TRCHT	Truncated height (feet). This is the height to a broken top or to topkill.

Once the data is read from the file into the `Tree` struct, it may only be accessed using the `findTreeList` accessor. `findTreeList` takes a stand identifier as a parameter and returns the corresponding linked list of trees.

The data in the `Tree` struct is used (almost)¹ solely as an interface to the `.trl` file. It is used to populate the `tree_arrays` global variable of type `TREE_TYPE` in function `F_ReadTreeList` in file `standInterface.c`. There is also a global variable called `tree_default` that is never initialized; the code that performs initialization is

¹The next paragraph explains why this qualifier is used.

currently commented out. Note that this “file interface” data is (almost) always accessed via a path from `F_LoadStandData` found in `standInterface.c`.

<code>tree_arrays->tree_types</code> Field Name (Application Abstraction)	Initialized by Tree Field (File Interface)
<code>species</code>	SPCD
<code>num_trees</code>	TREESPERACRE
<code>ave_dbh</code>	CURRDIAM
<code>ave_height</code>	CURRHT
<code>ave_cr</code>	CR
<code>db_height</code>	MAXCW
<code>age</code>	MS
<code>tree_status</code>	TRCL
<code>height_gro</code>	HTINCR
<code>diameter_gro</code>	DIAMINCR
<code>volume_gro</code>	MCHBDFTVOL
<code>total_cubic</code>	TOTCUFTVOL
<code>merch_cubic</code>	MCHCUFTVOL
<code>assortment1</code>	MCDF
<code>assortment2</code>	BDFD
<code>x_coord</code>	MORTALPERACRE

There are four fields in the `TREE_TYPE` structure that are not initialized with values that are read from the `.trl` file — these are calculated from the other fields once they are populated. The reason for the “almost” qualifier in the previous paragraph is that these four fields are not simply file interfaces but they are also the application's data abstractions. It is odd that the application's original authors would be inconsistent with the treatment of these four fields. This could be a place for improvement in the design since the `TREE_TYPE` structure consumes a large amount of memory that, potentially, could be freed once the data abstraction variables were populated. Until these four fields are treated consistently with the other fields as file interfaces only and not also as data abstractions, that is not possible. These four special fields are:

Field Name	Purpose	Initialization
<code>DBH</code>	Average diameter at breast height (meters)	<code>ave_dbh</code>
<code>crown_height</code>	crown height of the trees (meters)	If <code>ave_cr = 0</code> , it is $0.6667 * \text{ave_height}$ Otherwise it is $\text{ave_cr} * \text{ave_height} / 100.0$
<code>base_height</code>	(meters)	$\text{ave_height} - \text{crown_height}$
<code>crown_diameter</code>	(meters)	If <code>ave_height > 40</code> feet, it is 10 feet. If <code>ave_height ≤ 40</code> feet, it is $0.4 * \text{ave_height}$. Convert final result to meters.

Some unit conversions are necessary for the four computed fields defined above.

4.2.4 The `.slf` File

The `slfReader` function parses the `.slf` file and stores the data in the `standInfoArray` variable. Each set of stand information in the `.slf` file is composed of four records. Each record is designated with an A, B, C or D in the file. The program, therefore, has four different data types to hold each record type. The data types are called `StandInfo_A`, `StandInfo_B`, `StandInfo_C` and `StandInfo_D` where each holds its respective designated record. Every set of these records along with the stand's identifier is stored in a `StandInfo` structure. The `standInfoArray` is simply an array of pointers to `StandInfo` structures along with a count of how many items are stored in the array.

The format of an A-record in the `.slf` file is:

Never Used	Data Type	Field Name	Purpose
X	char *	fvsFName	The name of the file in FVS that generated the .s1f file.
X	char *	dataFlag	It's not clear what this field is. The value NoPointData is found here.
X	Node *	variants	A linked list of strings (strings' length is at most 2.)

The format of a B-record in the .s1f file is:

Never Used	Data Type	Field Name	Purpose
	int	inventoryYear	
	int	latitude	
	int	longitude	
X	char *	locationCode	
X	char *	habitatType	
	int	originYear	
	int	aspect	aspect in degrees
	int	slope	percentage
	int	elevation	elevation in hundreds of feet
	float	basalAreaFactor	a negative value is read as inverse of large-tree fixed plot size in acres
	float	inversePlotSize	inverse of fixed plot size in acres
X	float	breakPtDBH	in inches
X	int	numOfPlots	
X	int	numOfNSPlots	number of non-stockable plots
X	float	sampleWeight	stand sampling weight or size (usually acres) used to compute weighted average yield tables and other weighted averages among some set of stands.
	int	stockable	stockable percent.
X	char *	diaTransCode	diameter growth translation code as defined for FVS
X	int	diaMeasurePd	diameter growth measurement period (in years)
X	char *	htTransCode	height growth translation code as defined for FVS
X	int	htMeasurePd	height growth measurement period (in years)
X	int	mortalMeasurePd	mortality measurement period (in years)
	float	maxBasalArea	maximum basal area
	int	maxDensityIndex	maximum stand density index
	int	species	site species code
	int	index	site index
X	int	type	modes type code
	int	region	physiographic region code
	int	forestType	forest type code

The format of a C-record in the .s1f file is:

Never Used	Data Type	Field Name	Purpose
X	char **	key	
X	char **	value	
	int	count	The number of key-value pairs

The C-record is clearly a dictionary. The dictionary is never used, however.

The format of a D-record in the .s1f file is:

Never Used	Data Type	Field Name	Purpose
X	char **	addFile	
	int	count	The number of files in the array

The `StandInfo*` structures are all used solely as an interface to the `.slf` file (really only the `StandInfo_B` structure is used.) The data in these structures is used to populate the `stands` variable in `standInterface.c`. The `stands` variable is then used for the data abstractions for the application. Note that the `stands` variable is not populated until *after* the `.stnd` file is read (see Section 4.2.5). The parsing of the `.slf` file only creates the `StandInfo*` structures that are used as the file interface. With that understanding, the table below is presented here for completeness.

<code>stands []</code> (Application Abstraction)	Field Name	Initialized by <code>StandInfo_B</code> (File Interface)
	<code>inventoryYear</code>	<code>inventoryYear</code>
	<code>origin_year</code>	<code>originYear</code>
	<code>aspect</code>	<code>aspect</code>
	<code>slope</code>	<code>slope</code>
	<code>elevation</code>	<code>elevation</code>
	<code>app</code>	<code>inversePlotSize</code>
	<code>stockable</code>	<code>stockable</code>
	<code>max_basal_area</code>	<code>maxBasalArea</code>
	<code>density</code>	<code>maxDensityIndex</code>
	<code>site</code>	<code>species</code>
	<code>index</code>	<code>index</code>
	<code>physiographic_region</code>	<code>region</code>
	<code>forest_type</code>	<code>forestType</code>

4.2.5 The `.stnd` File

The `.stnd` file contains additional information for the `stands` global variable. Recall that most of the data in the `stands` variable is populated using the data read from the `.slf` file. The data from the `.slf` file does not flow to the `stands` variable until after the `.stnd` file is parsed in `F_LoadStandData` even though the `.slf` file was previously parsed. This section only describes the data that populates the `stands` variable from the `.stnd` file. For more information on how the data from the `.slf` file populates the `stands` variable, see Section 4.2.4.

The first data in the `.stnd` file populates several fields of the `stnd` variable. These fields comprise the header of the `.stnd` file:

Never Used	Data Type	Field Name	Purpose
	int	<code>nCols</code>	The number of columns in the <code>.stnd</code> file.
	int	<code>nRows</code>	The number of rows in the <code>.stnd</code> file.
	int	<code>xllCorner</code>	The x-component of the south-west corner.
	int	<code>yllCorner</code>	The y-component of the south-west corner.
	int	<code>cellSize</code>	The width and height of each grid. The value is often 30. (Meters)
	int	<code>noDataVal</code>	The value used to represent "No Data"

The remainder of the `.stnd` file contains stand identifiers in a `DEM_ROW × DEM_COL` matrix. As each stand identifier is read from the file, this parser determines if there is an entry in the `stands` array for this stand identifier already. If so, that entry's `num_grids` integer counter is incremented. If not, an entry in the `stands` array is created, the `id` for that entry is set to the stand identifier read from the file and the `num_grids` integer counter becomes 1. Thus, for each stand identifier, there is exactly one entry in the `stands` array for that stand identifier.

After the entries are created for each stand identifier and the count of the number of times each stand identifier is found in the `.stnd` file is set in the `num_grids` field, the data read from the `.slf` file then flows into the corresponding entries of the `stands` array.

At this point, the only field in the `stands` array that has not yet been created is the `tree_array_p` field. This field is set in the `gen_tree_list` function that is called when the trees are drawn to the screen, when the information dialog box is displayed (see Section 4.1.10) or when texture mapping is performed.

4.2.6 Initialization Dataflow

Thusfar the discussion has focused on a fine-grained view of data initialization. Figure 1 adopts a bird's-eye view of the dataflow.

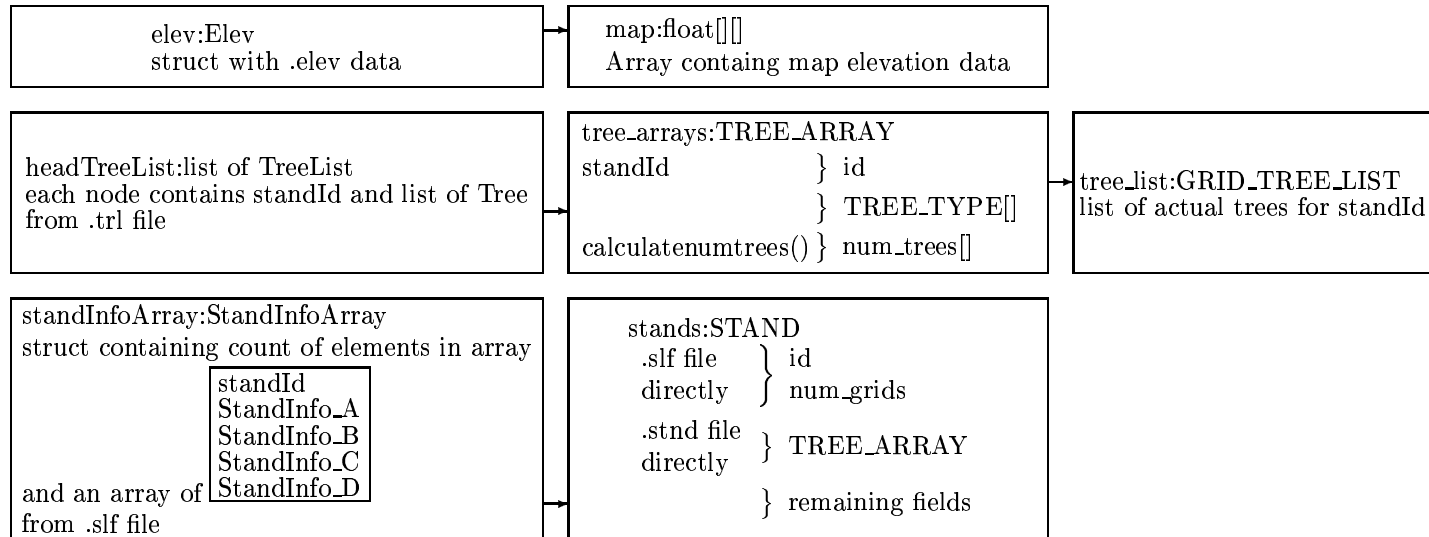


Figure 1: Initialization Dataflow

The first line in each box is the global variable's name and its data type. The lines that follow describe the structure at a very high level. The arrows show data flowing from the files through the variables. The heads of the arrows point to the fields that are the destination of that flow. Not all data abstractions that are loaded directly from the files are shown.

4.2.7 Callbacks

This section describes the events that initiate the various callbacks that SmartForest has implemented. In each section heading, the name of the callback is followed by the name of the file where it is defined. The list is sorted by the callback names.

Action in datamap.c The callback is a general-purpose callback for the features that `datamap.c` provides. The following list describes the events that cause `Action` to be called. Each of these events is relative to the `datamap` dialog box. This is the dialog box that allows users to modify the colors of trees.

- Clicking the `Accept` button.
- Clicking the `Apply` button.
- Clicking the `Default` button.
- Clicking the `Cancel` button.

- Clicking the `Help` button.
- Clicking the `SPECIES` button.
- Clicking the `CURRDIAM` button.
- Clicking the `DIAMINCR` button.
- Clicking the `CURRHT` button.
- Clicking the `HTINCR` button.
- Clicking the `CRatio` button.
- Clicking the `CWidth` button.
- Clicking the `TRCL` button.
- Clicking the `Mistletoe Rating` button.
- Clicking the `TVolume` button.
- Clicking the `MVolume` button.
- Clicking the `BVolume` button.

Action in `legend.c` The callback is called when the user clicks on the `Close` button in the Legend dialog box.

Action in `map.c` The callback is a general-purpose callback for the features that `map.c` provides. The following list describes the events that cause `Action` to be called. Each of these events are relative to the Map View window.

- Clicking the `Cancel` button.
- Clicking the `Mode` button.

Action in `property.c` The callback is a general-purpose callback for the features that `property.c` provides. The following list describes the events that cause `Action` to be called. Each of these events is relative to the property dialog box. This is the dialog box that allows users to modify the datamap properties.

- Clicking the `Accept` button.
- Clicking the `Apply` button.
- Clicking the `Cancel` button.
- Clicking the `Help` button.
- Clicking the `SPECIES` button.
- Clicking the `CURRDIAM` button.
- Clicking the `DIAMINCR` button.
- Clicking the `CURRHT` button.
- Clicking the `New` button.
- Clicking the `Delete` button.
- Clicking the `Sort` button.
- Clicking the `HTINCR` button.
- Clicking the `CRatio` button.
- Clicking the `CWidth` button.
- Clicking the `TRCL` button.

- Clicking the `Mistletoe Rating` button.
- Clicking the `TVolume` button.
- Clicking the `MVolume` button.
- Clicking the `BVolume` button.
- Clicking the `MC DF` button.
- Clicking the `BF DF` button.
- Clicking the `Motal` button.

Action in `standLegend.c` The callback is called when the user clicks on the `Close` button in the `Stand Legend` dialog box.

Action in `standmap.c` The callback is a general-purpose callback for the features that `standmap.c` provides. The following list describes the events that cause `Action` to be called. Each of these events are relative to the stand map dialog box. This is the dialog box that displays stand and tree list information.

- Clicking the `Help` button.
- Clicking the `Close` button.
- Clicking the `Update forest` button.
- Clicking the `Save to file` button.
- Clicking the `Edit >>` button.
- Clicking the `<` button.
- Clicking the `>` button.
- Clicking the `Delete` button.
- Clicking the `New` button. The `New` button only appears in the “extended” stand map dialog box after the user clicks the `Edit >>` button.
- Clicking the `Apply` button. The `Apply` button only appears in the “extended” stand map dialog box after the user clicks the `Edit >>` button.

Cancel in `file.c` The callback is called to handle the user clicking the `Cancel` button in several dialog boxes:

- The dialog box that appears when the user chooses `File-Save Image As`
- The dialog box that appears when the user chooses `File-Load Preference`
- The dialog box that appears when the user chooses `File-Save Preference`

The code also uses this callback for several items that are not currently available to the user. It appears to be intended as an option in the dialog box that would appear with several of the choices in the `File` menu bar hierarchy — specifically:

- `File-New Files-Elevation...`
- `File-New Files-Stand...`
- `File-New Files-Tree List...`
- `File-New Files-Multiple...`

However the code to add these options to the `File` menu selections is commented out thus the dialog box that would generate this callback is never created.

colorSelection in datamap.c The callback handles the event when a user selects a color name in the scrolled list of colors that appears in the datamap dialog box.

Dismiss in file.c The callback is not currently used. It used to handle clicking on the **Dismiss** button in the dialog box that used to appear when the user chose **Help-Usage...** from the menu bar in the Map View. The option is still available from the menu bar, but a different dialog box than the one containing this callback appears.

dataSelection in datamap.c There are two events that use this callback. The first event is when the datamap dialog box is first initialized. A “default” selection needs to be made in the data scrolled list. This callback makes that default selection. The callback is also used when the user manually selects something in the data scrolled list.

dataSelection in property.c The callback is used when the user selects an item from the data scrolled list in the dialog box that allows the user to edit datamap properties.

dem_toggled in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Multiple...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

draw_map in map.c The callback is used with Open GL windows. It is the callback that occurs when the application needs to repaint the map window in the Map View.

draw_window in callback.c The callback is used with Open GL windows. It is the callback that occurs when the application needs to repaint the Forest View.

Finish in callback.c The callback is called when the user clicks on the **Accept** button on the Map View.

file_help in file.c The callback is called to handle the user clicking the **Help** button in several dialog boxes:

- The dialog box that appears when the user chooses **File-Save Image As**
- The dialog box that appears when the user chooses **File-Load Preference**
- The dialog box that appears when the user chooses **File-Save Preference**

The code also uses this callback for several items that are not currently available to the user. It appears to be intended as an option in the dialog box that would appear with several of the choices in the **File** menu bar hierarchy — specifically:

- **File-New Files-Elevation...**
- **File-New Files-Stand...**
- **File-New Files-Tree List...**
- **File-New Files-Multiple...**

However the code to add these options to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

init_map in map.c The callback is used with Open GL windows. It is the “initialize the GL window” callback for the map window in the Map View.

init_window in callback.c The callback is used with Open GL windows. It is the “initialize the GL window” callback for the Forest View.

input_DEMFile in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Elevation...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

input_PrefFile in file.c The callback is used to handle the user's clicking on the OK button in the dialog box that appears when they select either **File-Load Preference** or **File-Load Preference** from the menu bar in the Map View.

input_StandFile in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Stand...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

input_TreeListFile in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Tree List...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

listSelection in standmap.c There are two events that use this callback. Both occur inside the standmap dialog box that shows both stand data and the tree list data for each stand. The first event is when the user selects a stand from the **Stand_ID** area scrolled list. The second event is when the user selects a tree from the **Tree List** scrolled list.

MenuBarCB in menu.c This callback is used when the user makes any selection from the menu bar in the Map View.

new_height in widgets.c The callback is called when the user changes the value of the **Elevation** slider on the Map View.

new_horizon in widgets.c The callback is called when the user changes the value of the **Horizon** slider on the Map View.

output_ImageFile in file.c The callback is used to handle the user's clicking on the OK button in the dialog box that appears when they select **File-Save Image As** from the menu bar in the Map View.

RGBSelection in datamap.c The datamap dialog box has three slider controls. Each control represent either the red, green or blue component of the color of the currently selected data item from the data scrolled list. The callback is used when the user either slides any one of these sliders controls or clicks to make the slider selection jump to a new position.

read_Files in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Multiple...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

read_filename in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Multiple...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

resize_map in map.c The callback is used with Open GL windows. It is the callback that occurs when the map window in the Map View is resized.

resize_window in callback.c The callback is used with Open GL windows. It is the callback that occurs when the Forest View is resized.

saveConfirmAction in standmap.c When the user clicks on the **Save to file** button in the standmap dialog box, a confirmation dialog box is brought up. If the user confirms that they want to save the file then this callback is called.

show_map in widgets.c The callback is called when the user clicks on the **Map** button in the Map View.

stand_toggled in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the **File** menu bar hierarchy — specifically under **File-New Files-Multiple...** However the code to add this option to the **File** menu selections is commented out thus the dialog box that would generate this callback is never created.

textAction in property.c In the dialog box that allows the user to edit datamap properties, there are several input boxes that allow the user to type new values directly. These boxes are labeled **Name:**, **Tree Type:**, **Min:**, **Max:**, **R:**, **G:** and **B:**. When the user types any new data in any of these boxes, this callback is called. The callback is also used when the user clicks on the check box labeled “Show this data item”.

textAction in standmap.c When the user is in the standmap dialog box and they click on the **Edit >>** button, the dialog box is extended with additional controls. Several of those controls are edit entry boxes that allow the user to type new values for the variables. The list of edit entry box labels is currently:

- (Measurement) Error (%)
- Specie
- Status
- DBH
- Trees
- Height
- CrownRatio
- DeadB_height
- AGE
- Height GRO
- Diameter GRO
- Volumn GRO
- Total Cubic
- Merch Cubic
- Assortment1

- Assortment2
- Tree Quality
- Tree Value
- X Coord
- Y Coord

The callback is called when the user modifies the data in any of these edit entry boxes.

toggleAction in standmap.c There are two events that cause this callback to execute. When the user is in the standmap dialog box and they click on the `Edit >>` button, the dialog box is extended with additional controls. One of those additional controls is the `Tree Distribution` box with two radio buttons labeled `Random` and `Uniform`. The callback is executed when the user clicks on either the `Random` radio button or on the `Uniform` radio button.

tree_toggled in file.c This callback is currently never initiated. It appears to be intended as an option in the dialog box that would appear with one of the choices in the `File` menu bar hierarchy — specifically under `File-New Files-Multiple...`. However the code to add this option to the `File` menu selections is commented out thus the dialog box that would generate this callback is never created.

viewAction in standmap.c When the user is in the standmap dialog box and they click on the `Edit >>` button, the dialog box is extended with additional controls. There is a scroll bar that appears next to all of the labeled edit entry boxes. The callback is executed when the user moves that scroll bar.

viewGinit in datamap.c The callback is used with Open GL windows. It is the “initialize the GL window” callback for the datamap dialog box.

viewRedraw in datamap.c The callback is used with Open GL windows. It is the callback that occurs when the application needs to repaint the datamap dialog box.

viewResize in datamap.c The callback is used with Open GL windows. It is the callback that occurs when the datamap dialog box is resized.

5 Known Concerns

- The original authors' reliance on global variables is very troubling. While the overall architecture of the system is sound, the use of global variables allows for unforeseen, subtle and potentially problematic interactions to occur. It would be nice to move the use of global variables to a more object-orientedish encapsulation. It's understandable that callbacks will need to use variables that are not parameters or local to the callback. However, that is not enough of a reason to violate/ignore encapsulation. We have started to slowly add more encapsulation to the program.
- In `menu.c`, X Windows determines its response to a menu choice by the physical position of the choice in the menu. The translation from menu position to action is rather ugly because it is context sensitive: If the position of a menu item moves or if an addition is made, several places need to maintain that ordering information. This problem has been addressed for the stand features menu. The change needs to occur for the other menus as well so that items can be moved around in the menu independently of their callback handlers' order.
- Some of the type names do not correspond to their actual duties. `TreeList` is an example since it's really the stand data where the stand contains a linked list of trees whose type is called `Node`. Some names could use an overhaul.

- Several of the fields in the `TREE_TYPE` struct aren't used. Of course, this relates to dead code/dead variable problems noted earlier in the document.
- There is dead code throughout the source files. It would be useful to trim the dead code from the source files. One tool that would be extremely helpful in identifying dead code is built by Rational. Alas it is quite expensive.
- The width of viewing area is slightly lower than it should be. The problem that this introduces is that, periodically, sections of the very edges of the Forest View will not paint. There will be "jagged edges" and areas that are not drawn on the screen. These areas are very small but this problem still occurs.
- Collision detection with the ground is not implemented. Thus you can end up under the ground.
- The num lock, caps lock and scroll lock keys must be off. If any are on, the mouse clicks will not register since X cares about things like shift key, ctrl key, num lock, etc. when reading input. It would be nice if the application made sure that these three are off.

Acknowledgements The SmartForest project is headed by its founder, Brian Orland. SmartForest is currently housed at Penn State University. The project originated under Prof. Orland at the University of Illinois at Urbana-Champaign. The original software developers are Jay Obermark, Daniel Pape, Paul Radja, Kittipong Mungnirum, Kenneth C. Schalk, Kaiyu Pan, Kun Liu, Christopher Currie and Xia Lu at UIUC [Orl97]. Lan Wu-Cavener at Penn State also contributed to this implementation and is currently working on the Java version of SmartForest.

Appendix A Input File Generation

Prof. Orland suggested that the document would be more complete if an appendix was included that described how each of the input files to SmartForest is generated.

Appendix A.1 Generating an `.elev` File

TODO : Add a paragraph explaining how this file is created.

Appendix A.2 Generating a `.trl` File

TODO : Add a paragraph explaining how this file is created.

Appendix A.3 Generating a `.slf` File

TODO : Add a paragraph explaining how this file is created.

Appendix A.4 Generating a `.stnd` File

TODO : Add a paragraph explaining how this file is created.

References

- [Orl97] Brian Orland. SmartForest-II: Forest visual modeling for forest pest management and planning. Available on the web at <http://www.imlab.psu.edu/papers/sfjul97/index.html>, 1997.
- [WF95] Sharon W. Woudenberg and Thomas O. Farrenkopf. The westwide forest inventory data base: User's manual. General Technical Report INT-GTR-317, U.S. Department of Agriculture, Forest Service, Intermountain Research Station, Ogden, UT, 1995. Available on the web at <http://www.srsfia.usfs.msstate.edu/woman.htm>.