REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	14 Jul 2004	Final 11 Mar 2002 – 30 Sep 2003

4. TITLE AND SUBTITLE

BST Graphics Platform

5. FUNDING NUMBERS

6. AUTHORS

Stephen J. Dow

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Alabama in Huntsville
Huntsville, AL 35899

8. PERFORMING ORGANIZATION REPORT NUMBER

5-22005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Aviation & Missile Command
Redstone Arsenal, AL 35898

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The SED Basic Skills Trainers (BST), which display targets moving over a background terrain scene, use a layer of graphics software we call the graphics platform to deal with the low-level construction of each video frame. The graphics platform used in the trainers fielded to date was mostly developed in the late 1990s and has limited ability to handle dynamic effects such as atmospheric obscurants and cannot perform continuous zoom. To overcome these limitations the graphics platform has been considerably restructured.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Graphics architecture	10
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

BST Graphics Platform

Stephen J. Dow
Department of Mathematical Sciences
The University of Alabama in Huntsville

July 14, 2004

Final Report

F/DOD/ARMY/AMCOM/Training Device Software Research
DAAH01-97-D-R005 D.O. 44

Period of Performance: 3/11/02 to 9/30/03

1. Introduction

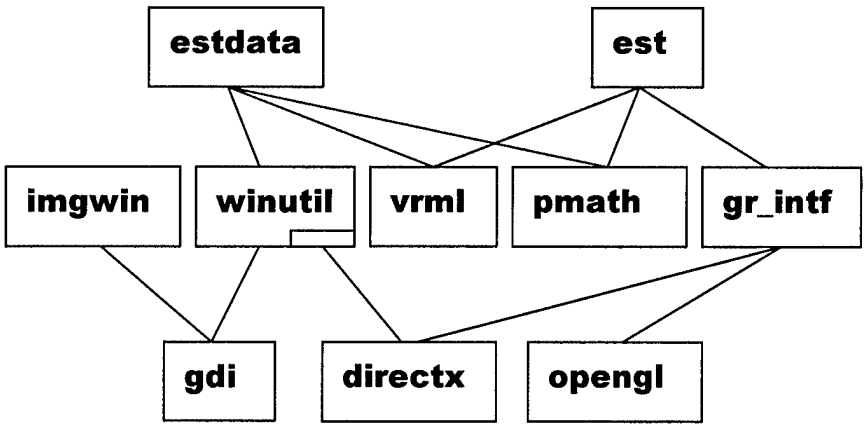
This report concerns software developed for a series of Basic Skills Trainers (BST) which display targets moving over a terrain scene, and in particular the portion of that software dealing with the low-level construction of each video frame to be displayed; we call this portion of our software the *graphics platform*. The graphics platform used in the trainers fielded to date was mostly developed in the late 1990s as a portion of the Javelin EPBST software, and is described in a 2001 report [1] on that effort and a 2001 SPIE conference paper [2]. Methods used in that graphics platform limit its ability to handle dynamic effects such as atmospheric obscurants and do not allow for continuous zoom. To overcome these limitations a considerable restructuring of the graphics architecture was needed. This report documents the restructured graphics platform in its present state.

2. Overview

We begin with a brief overview of the differences between the new platform and old. As described in the papers cited above, the old platform used a method of constructing each new video frame by copying and shifting the previous frame, and then constructing only portions of the new frame where changes occurred. Only in certain circumstances, such as when the user changed the brightness setting, would a "full update" be needed, in which case the previous frame would be ignored in constructing the new frame. The new platform dispenses with this scheme and constructs each frame independently of the previous frame (although some

components used in constructing the frames continue to be reused from frame to frame). The other major difference in the structure of the new platform is that it makes more use of graphics card acceleration, and allows for the use of either DirectX or OpenGL, rather than just DirectX, to access the graphics card functionality. Since we are already using the term "graphics platform" for something else, here we will refer to this underlying graphics layer (DirectX or OpenGL) as the *graphics API.* In both the old and new platforms, the graphics API is used to render targets into offscreen bitmaps. Beyond that, the old platform used the graphics API only in the final transfer of the rendered scene to the frame buffer, whereas the new platform uses the graphics API more extensively. Specifically, in the old platform the function EstDrawSceneRect combines bitmaps of rendered targets, explosion effects, and terrain into a scene bitmap by means of our own pixel-by-pixel operations with no calls to the graphics API. The combined bitmap is then transferred to the frame buffer by a DirectX call. In contrast, the new platform uses graphics API calls to store the component bitmaps (rendered targets, explosion effects, terrain) in video memory as textures, and then combines them on the graphics card via further graphics API calls.

We now discuss the organization of the code we are calling the graphics platform. The diagram below illustrates the dependencies among some of our "common" libraries and three Windows graphics APIs.



Each of the seven boxes in the top two rows above corresponds to a subdirectory of WinCode\common in the source code directory tree, containing a number of C source code files, which when compiled produce a library having the indicated name; for example, estdata.lib. Each entry also corresponds to a header file of the same name found in

WinCode\common\include; for example, estdata.h. The two libraries at the top of the diagram (estdata, est) contain high level functions with the prefix Est, such as EstAddTarget. One of these (estdata) is the main component of the "old platform" and the other (est) is the main component of the "new platform." The other part of the old platform is the portion of the winutil library which interfaces to DirectX; it consists of three source files containing the functions with prefix Dx, and is indicated in the diagram by the subrectangle in the lower right of the winutil box. The other part of the new platform is the new gr_intf library, discussed in some detail below.

Other parts of the common code are independent of whether the old or new platform is being used. As indicated in the diagram, both estdata and est depend on functions in vrml (for reading target model files) and pmath (for reading ground files, doing matrix computations, etc). The Exercise Editor built on the new platform continues to use parts of winutil other than the Dx functions.

The imgwin library deals with displaying imagery in a window, such as the "preview" window of the Javelin EPBST. It was used in the old platform version of the Exercise Editor, but the new platform version does not need it, using instead functions from gr_intf to take care of the image display there. However, it is possible to use imgwin in a new platform application if needed, and this is currently being done for the DEM window in Ground Editor.

3. The gr_intf library

We now turn to the details of the first of the two libraries making up the new platform, namely gr_intf, which stands for "graphics interface." The following table lists the source files making up the library:

gr_intf.h	Header file defining interface to the library
gr_intf.c	Interface functions with prefix GRi providing access to the functions in gl_intf.c and dx_intf.c.
gl_intf.c	Implementation in OpenGL, functions are prefixed GLi.
dx_intf.c	Implementation in DirectX, functions are prefixed DXi.
gr_font.c	Functions for creating fonts and drawing text with them. They are implemented by calls to other GRi functions.

gr_tile.c Functions for managing display of a large image by breaking it up into tiles and managing which tiles are stored in texture memory at a given time. They are implemented by calls to other GRi functions.

scrl_win.c Functions for handling a window with scrollbars. These are completely independent of the rest of the gr_intf library, but can be used for scrollable windows using GRi drawing (to replace similar functionality in imgwin).

The library isolates dependencies on OpenGL to a single file gl_intf.c, and dependencies on DirectX graphics (Direct3D) to a single file dx_intf.c. Those two files contain a parallel set of functions having the same names, functionality and interfaces, except the first is prefixed GLi and the second DXi. For example, GLiSetColor and DXiSetColor each set the active color used in subsequent drawing commands. A third file gr_intf.c contains interface functions (prefix GRi) which intermediate between the two APIs; in most cases simply choosing which of the corresponding functions (GLi or DXi) to call. For example, the code for GRiSetColor is simply:

```
if (gi.opengl) GLiSetColor(r, g, b);
else           DXiSetColor(r, g, b);
```

An initialization function GRiInit function specifies which of the two APIs to use. The intention of this code organization is to allow our other source code, such as that in the est library, to be written independently of which API being used.

The following program skeleton illustrates the basic structure of a Windows program using the GRi functions:

```
hwnd = CreateWindow("GraphicsClass", "", WS_VISIBLE, ...)
GRiInit(hwnd, 0, 0, 0, TRUE);
while (1) {
   if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
      if (msg.message == WM_QUIT)
         break;
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   } else {
      GRiSetViewport(0, 0, wxsize, wysize);
      GRiSetTransform(x0, y0, sx, sy, zmin, zmax);
      GRiClear();
```

```
            GRiEnable(GRI_DEPTH);
            GRiSetActiveTexture(ctex, ntex);
            GRiRender(GRI_TRIANGLES, xyz_list, 0, uv_list, num_points);
            GRiDisable(GRI_DEPTH);
            GRiSetColor(255, 0, 0);
            GRiRender2i(GRI_LINES, icoord, 0, 0, 4);
            GRiFlip();
        }
    }
    GRiExit();
```

In this example a window is created and the resulting window handle (hwnd) is passed to GRiInit, which initializes the graphics system and tells it that drawing will be done into that window. (The function GRiSetWindow can be used to switch to drawing into another window if needed.) The main Windows messaging loop is structured the same way it has been in all our BST application programs which draw moving targets; i.e. if PeekMessage indicates a Windows message has been queued (a key has been hit, the mouse moved, etc), then the message is dispatched to the appropriate Windows procedure, otherwise we have some time to draw the next frame. Generally that "else" code is longer than the few lines shown above, and is split out into a function we have been calling ProcessFrame. The new frame is drawn on the "back buffer" and then GRiFlip swaps the front and back buffer so that the newly drawn frame becomes visible. GRiSetViewport and GRiSetTransform set up the portion of the window to be drawn and the geometry (scaling, etc) to use. GRiClear fills the back buffer with the background color. The drawing calls illustrate enabling depth occlusion and drawing some textured triangles, and then disabling depth occlusion and drawing a couple 2d lines on top of the triangles. Of course all the various parameters and coordinates passed to the functions would need to have been set up with sensible values in code not shown here.

4. The est library

The est library is the high-level library that deals with the data structures needed to display the training simulation; i.e. terrain, targets, explosions, symbology, etc. In the old platform, the construction of a "scene patch" consisting of terrain with overlaid targets and explosions was handled within the library (estdata.lib), but any further symbology and the final

transfer to video memory were handled in application code. In the new platform, more of the graphics pipeline is handled within the est library. Here we give a high level overview of the process.

As in the old platform, the function ExReadDataFile reads information from the "data file" (.dat) into a top-level data structure called ExData which stores the list of available terrains, target models, film clips, and exercises. Typically this step occurs at initialization, and then an exercise from the exercise list is selected in some manner. If film clips (for explosions, etc) are to be used they are typically loaded at initialization as well (EstLoadFilmClip). The following example code omits various details but shows the main remaining steps needed to load and display an exercise using the new platform:

```
Exercise *ex;
EstData *ed;

hwnd = CreateWindow("GraphicsClass", "", WS_VISIBLE, ...)
GRiInit(hwnd, 0, 0, 0, TRUE);
EstLoadTerrain(ed, ex->terrain_num);
if (using_ir)
   EstLoadIR(ed);
for (n = 0; n < ex->num_targets; n++) {
   EstAddTarget(ed, ex->target[n].tgt_vis, ex->target[n].tgt_ir,
                ex->target[n].path_id);
   EstReadPathFile(ed, n);
}
EstLoadTargets(ed);
while (1) {
   if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
      if (msg.message == WM_QUIT)
         break;
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   } else {
      ed->current_time = ...
      EstSetLOS(ed, x, y);
      EstDrawFrame(ed, 0, EST_UPDATE_SCENE | EST_DRAW_SYMBOLS);
   }
```

```
}
EstUnloadTerrain(ed);
GRiExit();
```

Some required initializations have been omitted in the code above. The ex and ed pointers need to point to valid data structures, and some fields within the ed structure need to be initialized such as the fields specifying directories where files are to be found:

- terrain_dir, the directory containing the terrain files discussed below
- target_dir, the directory containing target files
- path_dir, the directory containing target path files
- clip_dir, the directory containing film clip files
- icon_dir, the directory containing icon bitmap files

The terrain files expected by the new platform for a terrain with id "cp1" are

- cp1.jpg or cp1.bmp, the visible terrain image
- cp1.gnd, the ground model file
- cp1_rng.mng, the above-ground objects file
- and, if IR is being used, cp1IR.jpg or cp1IR.bmp, the IR terrain image

If the visible terrain image above is not found in terrain_dir, the software looks instead for a file named cp1.cim. The cim extension stands for "Composite IMage". This is a simple ascii text format which we created to specify terrains put together using multiple images and/or multiple placements of an image within a larger image. As a simple example, assume that cp8.jpg is an ordinary 4000 x 700 jpeg image of a terrain scene, and that grass.jpg is a 1000 x 300 image of a grass texture. The following cim file creates an 8000 x 1000 composite image containing a copy of the terrain image in the upper left, a mirrored copy of the same image to its right, and four copies of the grass running across the bottom:

```
size = 8000 1000
file = cp8.jpg
placement = 0 0
placement = 4000 0 m
file = grass.jpg
placement = 0 700
placement = 1000 700
placement = 2000 700
```

```
placement = 3000 700
```

When using a cim file, the software still uses only a single ground file (.gnd); however, each image found within the cim file has its own above-ground objects file (.mng). The above-ground objects file contains individual binary (one-bit-per-pixel) images of above-ground objects such as trees or buildings, with the names of the individual objects within the file indicating their ranges in meters; for example, tree_1245.png indicating a tree at 1245 meters. If this mng file is not present, the software just takes that to mean there are no above-ground objects for the corresponding image.

If IR terrain imagery is to be used, an IR terrain image should be present alongside the visible terrain. It can be based on a cim file as well. The visible and IR terrain images must have the same angular coverage, but need not be at the same resolution. If the two are at different resolutions, the software adjusts the scale internally so that the same field-of-view appears in IR as in visible mode when switching between the two. The function EstSetScale is used to set the display scale for visible mode; for example calling EstSetScale(ed, 2.5f) sets the display scale to 2.5X, so that each terrain image pixel is displayed across 2.5 screen pixels. Note that the handling of scale is quite different from the old platform, in which narrow and wide versions of the imagery were stored in memory and a flag was used to indicate which version to display for the current frame.

Returning to the code listing above, note that the main "while" loop is very similar to the loop in the gr_intf example, the difference being in the else clause, where high-level Est function calls replace the GRi function calls to draw primitives such as lines and triangles. First the current time is updated; its computation is omitted; when the simulation is running it would normally involve a call to the system clock. Following that there is a call to EstSetLOS to specify the current line-of-sight (the terrain image location to appear at the center of the field-of-view). Then the line

```
EstDrawFrame(ed, 0, EST_UPDATE_SCENE | EST_DRAW_SYMBOLS);
```

performs all the actual drawing, using GRi functions internally to draw the graphics primitives. The last parameter is a combination of optional flags defined in est.h; the two flags shown here are EST_UPDATE_SCENE, which indicates that targets and film clips should be updated based on the current time, and EST_DRAW_SYMBOLS, which indicates that overlay symbols should be drawn. The system for placing symbology into the frame is new to the new platform;

functions have been added for creating and placing text, bitmap, and other symbol types.

We now discuss the graphics processing that occurs within EstDrawFrame. As in the old platform, one of the first steps is to render each of the targets which are currently in view into offscreen bitmaps. The target's position and orientation are computed based on the current time and the target's path, and a corresponding transformation matrix is computed. The frame buffer is cleared to a special background color, the matrix is applied to target vertices, the target model is rendered, and then the frame buffer read back into a bitmap in system memory. The bitmap is then antialiased and simultaneously converted a bitmap using alpha transparency to define the target/nontarget pixels. This step has been documented in a previous report [3], and is unchanged from the old platform. The new bitmap is loaded as a texture to be used in the final scene construction. At this point (during ESTiUpdateScene) the active frames of film clips which appear in the current frame are also identified and loaded as textures. Each of these scene objects (targets, film clip frames) is treated as lying at a single range; the objects are placed in a display list sorted by range from back to front.

The target and film clip display list having been created, the scene is now constructed in the frame buffer starting with the terrain imagery. The terrain imagery is handled as a "tiled image," in which the overall terrain is broken up into square tiles, a portion of which are in video memory as textures at any given time. The function GRiDrawTiles is used to transfer the appropriate imagery from the tiles into the frame buffer. Also if a portion of the field-of-view is above the top edge of the terrain, this portion is filled in with the "sky color." The video card now contains the background terrain scene, but no associated range values. To fill the ranges into the Z-buffer, the ground model and the above-ground objects are now rendered as depth only (i.e. leaving the color values in the frame buffer unchanged). Next the scene objects in the previously constructed display list are rendered at their respective ranges; they are occluded by terrain and each other as dictated by the terrain and object ranges. Note that the occlusion would work regardless of whether the objects were sorted by range or not; the sorting is needed because they also use alpha for partial transparency.

At this point the est code also allows for drawing additional 3d objects into the scene; that capability has been used a vehicle body in the foreground. Then if requested the ground triangulation and/or target paths are drawn. Finally, depth occlusion is turned off, and any additional symbology which has been enabled is drawn.

There are several mechanisms to allow the application code to insert additional drawing commands of its own along the way or at the end of the graphics process described above. To perform further drawing at the end of the process, the application may specify EST_NO_FLIP in the last parameter to EstDrawFrame. This turns off the call to GRiFlip, which as mentioned earlier causes the newly drawn frame to become visible. The application would then make GRi drawing calls of its own, ending in its own call to GRiFlip. The application may also insert drawing and other processing following "scene construction" (terrain, targets, film clips) and prior to the symbology drawing by assigning an application function to ed->FrameCallback. Another callback mechanism is provided by EstAddGenericSymbol, which takes an application callback function as one of its parameters; these callbacks occur as the list of symbols is traversed. Finally, a separate function named EstDrawScene may be used to draw the scene into an application bitmap rather than displaying it. This allows for further image processing of the scene, such as is needed to create a simulated seeker image, before display. In this case the application would call EstDrawScene to fill in the bitmap image, process it, and then pass the processed bitmap on to EstDrawFrame, which skips the scene drawing steps when a bitmap is passed as the second parameter.

Discussion of both focus processing and atmospheric obscurants has been omitted from this report, because those aspects of the new platform have not yet been fully developed.

References

1. Stephen J. Dow, *Javelin EPBST Software Development*, Final Report, Contract DAAH01-97-D-R005 D.O. 13, Oct. 2001.

2. Stephen J. Dow and Kevin Shelledy, A simulation methodology using photographic terrain and rendered objects, Proc. SPIE Vol 4368, p 74-81, Apr. 2001.

3. Stephen J. Dow, Basic Skills Trainer Simulation Improvements, Final Report, Contract DAAH01-97-D-R005 D.O. 29, May 2002.