CTDB 3D Viewer Editor

Richard Pater rpater@seas.upenn.edu

Matthew Hsieh hsieh2@seas.upenn.edu

Professor Barry Silverman

Abstract

Compact terrain database (CTDB) is a complex terrain database format widely used by the military in training simulations. Currently, many CTDB viewers are made specifically to render the terrain and object models 3 dimensionally in high resolution. These programs focus on editing capabilities that either change the way the terrain is presented in 3D or convert the loaded database to a different output format. Unfortunately, these programs do not provide a simple interface for adding markups, additional attributes for a terrain feature that a simulation system may wish to use, to the terrain features. In addition, these viewers tend to be overly complicated, unwieldy, and expensive.

The software we are building will be a user-friendly 3D viewer of the CTDB format. Our software will implement a simple and intuitive GUI to make 3D viewing and exploring extremely easy for a user. Beyond what CTDB viewers currently provide, our software will support editing and will interface with other related software. Editing will involve modifying existing attributes or adding new markups to objects in the environment.

Related Work

The CTDB format has been widely used by the military in virtual training exercises. This extensive database contains the physical and abstract features of terrain including buildings, roads, rivers, and terrain skin as well as attributes such as soil type or bathymetry (depth of water). The military uses this database format to realistically map and simulate buildings, roads, political boundaries, and other terrain features. CTDB viewers are programs that translate the database into full-scale, accurate representations of the environment, most commonly in 3D. There are a few commercial CTDB viewers that are on the market today, and they are usually acquired by purchasing a license. TerraTools, created by TerraSim, Inc., is a modeling program that creates a 3D environment from a CTDB or other terrain database format using OpenGL. This is similar in some ways to our software, but obtaining one license for the TerraTools package costs $23,000. A price this high makes the software prohibitively expensive for use in the types of applications for which our system will be designed. Terrain Experts, Inc. (TERREX) also has a CTDB viewer program called TerraViz which models the environment extracted from a CTDB in 3D. This program, however, has extremely limited editing capabilities and is primarily a simple viewer. They charge a more reasonable amount for this software, $1,500 per license, but it does not have the editing capabilities that our system has.

A major limitation to these commercial programs is that they do not contain programming that allows the type of editing that our system provides. Our software has the capability to add attributes or markups that can be used as descriptions, properties, or values. Importantly, markups can be used to form relationships between agents and the virtual environment in a simulation. A psychologist, JJ Gibson proposed that features in the environment (terrain, types of buildings, water, etc.) consist of affordances (possibilities for action) which provide necessary

clues toward perception. According to "Affordance Theory for Improving the Rapid Generation, Composability, and Reusability of Synthetic Agents and Objects," a virtual world based on affordances interacts with agents, leading to perception, sensation, and action. An example of this theory in use in a real system is the computer game, The Sims. The Sims allows simulated characters (sims, the agents in the theory) to perceive the virtual world and act out according to affordances in the environment. For example, a television set affords watching or playing videos. Sims have certain tanks such as energy, hunger, and entertainment as well as others. When the entertainment tank is low, a sim will seek objects in the environment that can fill this tank, such as the television set. By walking toward the television set and turning it on, a transaction takes place. The television will fill the entertainment tank but the sim's energy tank will decrease.

The markups that our system adds to the CTDB will be used in a way similar to the markups used in The Sims, but they will be used primarily by PMFServ agents, at least at first. PMFServ is a program that uses PMFs, or Performance Moderator Functions, to realistically simulate agents in an interactive environment. An agent has a large number of PMFs to regulate different aspects of behavior. Example PMFs include stress, personality, perception, social processes, and cognition. At the moment, PMFServ has no way to grab markups out of a CTDB or add markups to a CTDB that it needs.

Our project is a free alternative to existing CTDB viewers on the market. The fact that our software is freeware gives it a distinctive edge over expensive existing commercial products. Our editor is also able to add markups to the CTDB database, which can be used in different ways. They can be used to describe physical location, name, or ownership as well as to describe affordances that will be useful for interactions between simulated agents and the virtual world

described by the CTDB. They are also useful in defining legal agent movement within the world by providing information such as whether or not an agent can walk on or through a particular location. Another property of markups is that they can aid navigation by providing endpoints for decision based movements. For example, a hungry agent might walk down a road and into a restaurant because the markup of the restaurant tells the agent that he can acquire food there. Through the XML-RPC interface that can be used by PMFServ, our viewer/editor can be used to populate the simulated world in PMFServ with objects and affordances leading to more complex and accurate simulated behaviors for the agents.

Technical Approach

The system that we built works by importing a CTDB tile and using the information contained within the database to draw a 3D representation of the terrain. CTDBs store terrain features as collections of vertices given in geocentric locations. Different types of features also contain additional attributes specific to those features. Buildings, for example, contain a collection of roof elevations that correspond to the vertices of the building while trees contain information such as height and radius of foliage. Our software extracts these terrain features from the database using the CTDB querying functions and draws them, using color coding and other similar schema to differentiate different types of features.

In order to display the information within the database, we needed to use an appropriate application programming interface (API) that could draw the 3D representation quickly and efficiently. Our viewer uses the OpenGL graphics standard developed by Silicon Graphics (SGI) to display the virtual world. OpenGL is described as being a “software interface to graphics

hardware"[1]. OpenGL is much simpler to use than DirectX, a graphics standard developed by Microsoft, because OpenGL uses intuitive functions to draw primitives whereas DirectX uses the COM object model. However, we chose OpenGL over DirectX mainly because we needed to use the Linux operating system in order to access the CTDB database and DirectX is a proprietary API that only works on the Windows platform[2]. We decided to program our software using C and C++ to link between querying the database and displaying the information.

Using the OpenGL Programming Guide (The Red Book) as well as online tutorials, we were able to develop algorithms for drawing buildings and other features at locations given in the database. In order to extract local features from the database, we calculated a search space in geocentric coordinates based on the current location of the "eye", or viewpoint. CTDB querying for buildings and features is done using this search space style due to the fact that the database is optimized for storing maps. We chose to create this search space as a square with each side twice as large as the distance that the eye can see in front of it, centered on the eye. The distance that the eye can see is simply a hard coded value that we chose to be as large as possible without overwhelming the hardware. The database is then queried for all buildings and features present in this search space, and the buildings and features are stored as lists of objects in memory with relevant attributes stored with each object. The buildings and features are then drawn based on these in-memory lists so that the database does not need to be queried every time the screen is redrawn. When the eye moves close enough to the edge of the search space, a new search space is created and the database is requeried, populating the lists with new buildings and features. Once the lists of buildings and features are in memory, they are looped through and each

---

[1] Wright and Lipchack, 33

[2] http://en.wikipedia.org/wiki/Comparison_of_Direct3D_and_OpenGL

building or feature is drawn and placed in the virtual world at the proper location based upon its vertices.

A major emphasis in designing our viewer was to have a visually rich and realistic representation of the buildings in the database. Because we had a set of vertices for each side of a building, we were able to map out walls as well as calculate roofs. Also, the database contained function codes which are numbers that described the use and role of each building. Therefore, we needed a visually appealing method to differentiate between residential houses, commercial structures, as well as military establishments. Our solution was to apply textures to the planes that represented the walls of each building. A major hurdle in applying textures was actually finding suitable images that could be applied. We found a package of free photorealistic textures from a website[1] that were in JPEG (Joint Photographic Experts Group) format. We decided to convert the textures to a more usable TGA (Targa) format. Since we had an existing TGA loader, it was much easier to use this format than any other raster-type graphical format. Another texture mapping hurdle was that the walls were not equal in width and height. Therefore, if a 1 to 1 ratio was applied from a texture to the plane, the texture would not look realistic, instead the texture would look stretched or compressed. We needed to calculate the proper ratio of height over width to ensure a realistic repetitive mapping that could cover each side of the building.

One problem that could not be resolved was properly mapping a roof to each building. There is an algorithm developed by Oswin Aichholzer called straight skeleton that could be modified to map a roof to each building. The algorithm involved creating and step-wise shrinking lines from each corner until the lines intersected. Each point where the lines intersected became a roof apex which when connected, formed the spine of the roof. However, there is no known public

[1] berneyboy.planetquake.gamespy.com/textures.htm

implementation of this algorithm and the running time is O(n log2 n + r3/2 log r) which is too slow to be used[1]. Instead we used a simpler algorithm that computed the midpoint of all the roof vertices and created a triangular roof from each roof endpoint to the midpoint. However, there are disadvantages for using this algorithm, specifically it does not work for all of our buildings. It works well for four-sided and five-sided structures but complicated buildings with many walls will have a roof that overlaps the edges of the building. Unfortunately, at this time, we do not have a better algorithm to create a roof for each building.

Another major milestone was the selection of buildings in the environment in order to add markups. To implement selection, we had a few options available. First, OpenGL had a built-in selection mechanism. In OpenGL selection mode, a list is constructed containing intersected primitives within the viewing volume at the selected vertex. By properly naming each object when drawing, you are able to determine which object was selected[2]. However, in a 3d environment, one point could have many intersecting primitives. For a 3d environment, we want the object closest to the viewpoint to be selected. An alternative method of selection which had the added benefit of always selecting the top-most object was color picking. This meant choosing what object was selected by using a unique identifying color. When creating each object, a unique color id was given using the color range of RGB. Since each color value ranged from 0 to 255, we had a total of 16,581,375 unique color ids to use. Using the left mouse button as a signal for selection, we would redraw the objects using the unique color id into the back buffer, meaning the scene was not refreshed. By using an OpenGL method that could select the pixel color at the mouse selection point, we were able to match the pixel color to the object using the

---

[1] Aichholzer

[2] http://www.codeproject.com/opengl/openglmousesellection.asp

color id. This method runs slower than the built-in selection mode but the performance hit is unnoticeable[1].

Another implementation to improve realism was adding in lighting from a singular source, emulating the sun and the effects on the viewer. In order for lighting to work, we used linear algebra formulas to calculate normals for each side of a building. A major use of lighting is to realistically place shadows on the ground plane. Using a projection matrix, we could formulate the proper equation for adding a shadow to each building. Unfortunately, we ran out of time to implement this feature.

There are two projection modes in OpenGL for cameras to view the digital world, orthographic and perspective. An orthographic projection can be described as viewing a flattened 3d image projected to a 2d plane. Perspective projection is a rendering that mimics actual visual perception. To develop a degree of visual realism, the viewer uses a bird's eye camera view and perspective projection to enhance characteristics of imagery such as foreshortening. Foreshortening is a graphical effect that mimics how a camera or how our eye works when viewing a group of objects from a distance[2].

Our viewer uses GLUT (OpenGL utility toolkit) developed by Mark Kilgard, formerly of SGI, in combination with GLUI (GLUT-based C++ user interface) developed by Paul Rademacher, as an interface for GUI features such as keyboard and mouse controls as well as controls such as check boxes and buttons. The user can travel through our virtual world in the x, y, and z directions by using simple mouse translation buttons. The user can also change the direction the eye is looking by clicking on a rotation button and moving the mouse around. This

---

[1] http://gpwiki.org/index.php/OpenGL_Selection_Using_Unique_Color_IDs

[2] Red Book, 133

mouse based navigation makes it very easy for anyone who is using the program to simply and intuitively explore the virtual world.

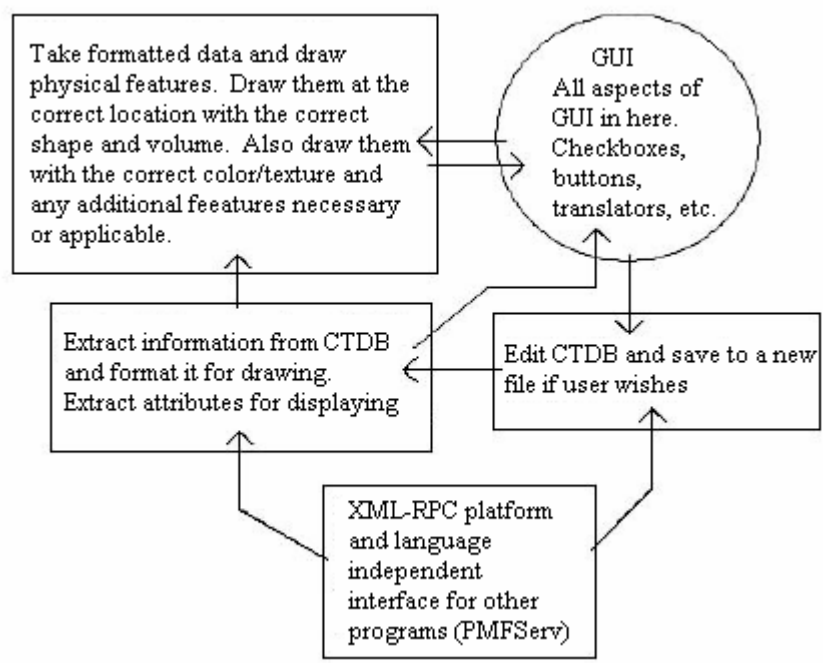
Along with navigation features, our user interface also contains additional simple controls to allow the user to easily change how things are being displayed. A user can turn textures on or off in order to see realistically textured buildings or simple color-coded buildings. Buildings, roads, and other features can also be individually turned on and off if a user wishes to have less clutter on the screen. The user interface also contains some editing and saving controls.

In addition to being a viewer of the database, our software is also an editor. Our graphical interface includes options to edit the various buildings and terrain features with the click of a button. A user can click on a building whose features they wish to view or edit and bring up an editor window. This window will display all the attributes currently set for this feature, along with the values that the attributes have. The user is then able to change any attribute or create a new attribute altogether. As long as there is room in the CTDB table for this feature, the added attributes can be written to the database if the user wishes. This allows users to permanently modify the database so that their changes will persist across loads and even applications. Once the table is full for a feature, additional attributes that are added to a feature will not be able to be written to the database and will be lost when the user exits our viewer. The functionality of adding attributes is particularly useful for adding markups to buildings and other terrain areas.

Another important part of our system is its ability to interface with other programs through the platform and programming language independent XML-RPC. We chose to use XML-RPC so that any system of any type written in nearly any language could interface with our software. We also chose it because PMFServ currently uses XML-RPC to successfully communicate over a network with other types of applications. We have set up a simple and at this point still

somewhat primitive API to allow programs to use remote procedure calls to view or edit the attributes of a building or feature based on geocentric location. Applications can also use this API to save the database to a file. Although we did not have enough time to expand this API, this could easily be done to add more functions and more sophisticated functionality. The CTDBs edited by our software are still functional for use in other systems. The inclusion of this simple interface for programmatic communication allows an application to interact with the database without going through the GUI human interface. This could be particularly useful if a person is trying to add or look up many attributes to buildings or features in a large database. The underlying XML-RPC interface could also be very useful for populating PMFServ with markups so that its agents have realistic information to make informed and accurately modeled decisions. Due to time constraints, we were unable to add graphical representations of PMFServ agents to our system.

Block Diagram of Software



Conclusion

We designed, built, and tested a 3 dimensional viewer and editor for CTDB format files. We also gave the application a simple and intuitive graphical user interface as well as a simple XML-RPC interface for programmatic interaction. Our program allows anyone to easily open a CTDB file, view and/or edit the buildings and features in that database, and save the new database to disk.

Working on this project made us appreciate the effort that goes into any kind of digital design. Working with the camera, adding textures, colors, and selection in OpenGL were all new concepts that we researched and implemented into our viewer/editor. Since we were beginners in graphical programming, every completed milestone was viewed as a big accomplishment. Getting the camera to move freely was the first major milestone and also one of the most rewarding. Another completed milestone that greatly improved the viewer was adding in realistic textures to the buildings. This step turned out to be easier than we anticipated since OpenGL had many built-in commands to easily add textures to planes. Working with the CTDB format turned out to be pretty challenging. Since the CTDB format is poorly documented, even though it is widely used in U.S. military applications, it was time consuming and difficult to extract all the features from the CTDB tile. The documentation would only lead to a file, if even that, and the code would have to be examined in order to determine its functionality. Also, the only documentation available was for a previous version of the CTDB format, so some functionality had changed or been replaced. It would have been much easier to add more features if the database code had been properly commented and the functions properly documented. Programming a GUI also turned out to a fairly hard problem as well since both authors do not have significant experience in this area. We did not even determine that GLUT's functionality

would not be enough to meet our needs until well into the project. We struggled a bit before finding GLUI, which gave us the functionality that we needed and integrated fairly easily with GLUT. Even then, our use of the open source freeglut implementation complicated our use of GLUI. In retrospect, we would have done a number of things differently. We could have spent more time on the XML-RPC interface to allow for smoother integration with other software. Also, we could have implemented GUI aspects earlier before other parts of the project in order for everything to work together a little more cleanly. Moreover, adding a roofing algorithm for the buildings would be a positive addition to the viewer.

References

Books:

Wright, Richard S. and Lipchak, Benjamin. *OpenGL Superbible*. Indianapolis: Sams Publishing, 2005.

Shreiner, Dave, Woo, Mason, Jackie, Neider and David, Tom. *OpenGL Programming Guide.* Fifth Edition. Upper Sadle River: Addison-Wesley, 2006.

Articles:

Aichholzer, O., Aurenhammer, F., Alberts, D., Gärtner, B. (1995). "A novel type of skeleton for polygons". Journal of Universal Computer Science 1 (12): 752–761.

Cornwell, J., O'Brien, K., Silverman, B. G., & Toth, J. (2003). "Affordance theory for improving the rapid generation, composability, and reusability of synthetic agents and objects." *Proceedings of the Twelfth Conference on Behavior Representations in Modeling and Simulation (BRIMS, formerly CGF)*, SISO. 12-15 May 2003.

Silverman, B. G., Johns, M., Cornwell, J., and O'Brien, K. 2006. "Human behavior models for agents in simulators and games: part I: enabling science with PMFserv." *Presence: Teleoper. Virtual Environ.* 15, 2 (Apr. 2006), 139-162.

Websites:

http://www.tec.army.mil/TD/tvd/survey/TerraTools.html

http://www.terrex.com/terraviz.htm

http://www.sedris.org/stc/2001/tu/ctdb/

http://tip.psychology.org/gibson.html

http://opensteer.sourceforget.net

http://www.cs.unc.edu/~rademach/glui/

http://www.codeproject.com/opengl/openglmousesellection.asp

http://gpwiki.org/index.php/OpenGL_Selection_Using_Unique_Color_IDs

berneyboy.planetquake.gamespy.com/textures.htm

http://en.wikipedia.org/wiki/Comparison_of_Direct3D_and_OpenGL