AUTOMATIC EXTRUSION OF THREE-DIMENSIONAL BUILDINGS FROM OVERHEAD TERRAIN PHOTOGRAPHY

by

Geoffrey M. Draper

Submitted to Brigham Young University in partial fulfillment

of graduation requirements for University Honors

Department of Computer Science

June 2000

Advisor: Parris K. Egbert

Signature:________________________

Honors Dean: Steven E. Benzley

Signature:________________________

Abstract

Previous research at the Advanced 3D Graphics Lab has produced a six-gigabyte geographic database of the Wasatch front. However, due to limitations in the elevation data, all the buildings appear flat, not rising up at all from the ground. This artifact is not noticeable when performing high-altitude fly-bys, but is unsuitable for close-range walk-throughs. We have devised an automated system to extrude buildings of any height from the otherwise flat terrain. Users can interactively select areas on the terrain where they wish to add buildings, and then our system generates a new version of the landscape, complete with full-size three-dimensional buildings at the chosen locations. This program was designed for the Virtual Olympics project, but is general enough to be used by any application desiring this functionality.

Table of Contents

I. Introduction 1

1.1 Virtual Olympics 1

1.2 Potential Uses for Virtual Olympics 2

II. Related Work 3

2.1 The Evolution of the Virtual Olympics Project 3

2.2 The Need for Further Work 5

2.3 Thesis Statement 6

III. Method of Building Extrusion 7

3.1 Preliminary Steps 7

3.2 Building Selection 8

3.3 VRML Code Creation 10

IV. Implementation Details 14

4.1 Placement of Buildings 14

4.2 Painting the Roof 15

4.3 Specific Challenges 16

4.3.1 N-Sided Polygons 16

4.3.2 The Indexed FaceSet Node 17

V. Future Work 18

5.1 N-sided polygons 18

5.2 Terrains Determined at Run-time 19

5.3 Using *Jview* More Than Once for the Same Scene 20

5.4 Automatic Edge Detection 20

VI. Conclusions 21

References 22

Table of Figures

Figure 1 8

Figure 2 9

Figure 3 11

Figure 4 12

Figure 5 13

Figure 6 16

I. Introduction

Although the field of computer graphics is relatively young, it has already sparked considerable interest in both industry and academe. Originally just an obscure topic of university research, rapid progress in both hardware and software over the past twenty years has catapulted computer graphics into the mainstream. One exciting application of computer graphics is the notion of *virtual reality*. Virtual Reality aims to simulate real or imaginary environments through a computer. It has been said that the "end goal of virtual reality is to create an interactive experience for the user that 'suspends disbelief.' This means that the simulated environment must be convincing enough so that the user feels 'immersed' in the scene." [2] This thesis builds on previous research conducted in the Advanced 3D Graphics Lab, providing a software tool to enhance the realism of our current Virtual Reality framework.

1.1 Virtual Olympics

The Advanced 3D Graphics Lab is currently implementing a unique virtual reality system known as "Virtual Olympics."[1] The 2002 Winter Olympics in Salt Lake City has generated intense interest in Utah tourism, and will continue to do so. Virtual Olympics is an attempt to "bring the Olympics to the people," in a sense. In brief, it is an interactive utility that allows users to virtually fly through a computerized 3D model of the Wasatch front in northern Utah. In addition to mere terrain, it also contains accurate models of several

[1] Although the Salt Lake Organizing Committee (SLOC) and other commercial parties have expressed interest in this research, the word "Olympics" as used herein is purely an in-house term, and does not necessarily reflect endorsement by the International Olympic Committee.

prominent buildings in downtown Salt Lake City — thus giving users a virtual tour of the city. Most excitingly, Virtual Olympics will permit users to visit the Olympic venues that will be used in the 2002 Olympic Games, and virtually ski, sled down, or just visit the venues of their choice. To ensure maximum availability, Virtual Olympics will be accessible via the World Wide Web. Thus, anyone with a home computer and an Internet connection will be able to get a taste of the Olympic games — no matter where they live.

1.2 Potential Uses for Virtual Olympics

As mentioned earlier, the Virtual Olympics project will be very beneficial for bringing the Olympics closer to those who normally would not be able to participate due to financial and geographic considerations. Newscasters could integrate virtual ski hills into their presentations; for example, an anchorman could describe the twists and turns of a ski run as the simulation chugs along on a large screen behind him, guiding his viewers down the slope before their very eyes. Schoolteachers could use the virtual environment to stimulate classroom discussion about the evolution of Olympic traditions. Since the virtual skiing environment will be available through the increasingly ubiquitous Internet, people who may have never been to (or even heard of) Utah will get to “visit” some of this state’s prime ski slopes – possibly leading to increased interest in Utah tourism. Moreover, an increase in Utah tourism almost invariably leads to an increase in curiosity about the LDS Church; so in a way, the Virtual Olympics project may have an indirect missionary benefit as well.

While the scope of Virtual Olympics is very large, this paper deals with one aspect in particular: the problem of quickly adding simple buildings to otherwise flat terrain. More details about this problem are discussed in the next section.

II. Related Work

2.1 The Evolution of the Virtual Olympics Project

The Virtual Olympic project traces its roots to previous research conducted in the Advanced 3D Graphics Lab. Kirk Duffin [5] built a 3D model of the Wasatch front from data obtained from the United States Geological Survey (USGS). The elevation data was taken from digital elevation models (DEMs), while the overhead photography came in the form of digital orthophoto quadrangle (DOQ) images. Duffin implemented, among other things, a novel system for automatically "stitching" together the individual DOQ's to flow together as a single terrain map. Duffin's work is often referred to as the "terrain database," a term which will be used frequently throughout this paper.

While Duffin's work was impressive, the amount of computation and storage required to manage all six gigabytes of the texture and elevation data in realtime is prohibitive, even on specialized graphics workstations. David Cline [1] devised a unique texture-caching system that treats the terrain textures as a bandwidth-limited resource. Instead of loading all the texture into memory at once, Cline's application, *Tview,* loads only those textures into memory that are needed for the user's current viewpoint. When the user's viewpoint is at a high elevation, fast-loading low-resolution images are used. As the user zooms in to view a portion of the terrain at close range, the low-resolution images are swapped out of memory in favor of more detailed images.

In addition to texture caching, Cline also created [3] a new algorithm for terrain decimation. Terrain decimation is the process of simplifying the geometry of the terrain to use fewer polygons, resulting in faster rendering and hence, a more interactive experience for the user. Cline's algorithm, called Quadtree morphing or simply "Q-morphing," utilized the quadtree structure of the terrain database to reduce the number of triangle strips in the terrain at run-time, dependant upon the user's position in the environment.

Cline's application was excellent at displaying terrains at interactive speeds, but adding additional objects, such as buildings, to the scene was problematic. Additionally, Cline used a proprietary graphics API, Iris GL, that only runs on IRIX-based Silicon Graphics workstations. To make 3D scenes that would run on multiple computer platforms, a more flexible approach was needed. A modeling language such as VRML (Virtual Reality Modeling Language) is very good at creating complex 3D environments, but it does not support Cline's advanced texture caching scheme. To overcome this challenge, Casey Songer created a new VRML browser from scratch, called *Cview,* that supports Cline's texture caching system in addition to supporting many standard VRML constructs as well. Furthermore, Songer's implementation replaced Cline's Iris GL code with OpenGL, [12] a platform-independent graphics API.

In an effort to speed up things even more, Brian Zaugg [13] extended *Cview* to include an improved occlusion culling algorithm. *Culling* is the process of determining which polygons are not visible for a particular frame, and simply not rendering them. Cline's original backface culling algorithm simply culled those polygons whose normals pointed away from the user's viewpoint. Zaugg improved this technique by culling all polygons which would be hidden by other polygons closer to the user's viewpoint. This technique dramatically reduced the time required to render each frame.

2.2 The Need for Further Work

The work described above makes up the bulk of the Virtual Olympics application as it currently stands. As impressive as it is, there is room for improvement. For example, the overhead photography obtained from the DOQ's is accurate to approximately one meter per pixel. However, the elevation data from the DEM's is much less precise, their samples having been taken approximately 30 meters apart. As a result, the mountains are the only landmarks that rise above ground level. Other features such as buildings and trees are visible only in the texture mapping. Even then, since the DOQ's consist only of *overhead* photography, only the tops of the trees and the tops of the buildings are visible. The geometries of the buildings and the textures on the sides of the buildings are completely missing from the terrain database.

This shortcoming is not noticeable in high-elevation fly-bys, since the user is more likely to pay attention to the mountains anyway, and since the buildings seem so tiny and insignificant down below. However, this illusion breaks down once the user zooms in closer to the buildings. As the user's viewpoint approaches ground level, it becomes obvious that what had appeared to be buildings are nothing more than images painted onto a flat surface.

To address this issue, 3D models of some of the more prominent buildings such as the Delta Center and the Salt Lake Temple were created to enhance the realism of the scene.

Adding these buildings is a tedious process that requires several hours of work per building. As helpful as these models are, they alone are not sufficient to give the user a feeling that he or she is actually inside the scene. The user needs to be surrounded by many buildings, not just a handful of the more important ones, in order to feel immersed in the virtual environment. Only then will the simulation suspend the user's disbelief.

2.3 Thesis Statement

This thesis proposes a software-based utility, *Jview,*[2] to assist student researchers in quickly extruding many buildings from the flat terrain with a minimum of work, and in much less time than it takes to model just one complete building. *Jview* does not attempt to model complicated buildings, since commercial applications like *Canoma* [8] and *3D Studio MAX* [10] already do this admirably well. This system gives its users the ability to designate the size and location of simple buildings to "fill out" the scene. This utility is to be an in-house tool for research use; hence its focus will be much more on function than on fancy "bells and whistles." Furthermore, it is not designed to do all of the work for the student, but it is designed to make his or her job *easier*. Its strength comes from letting the human do what humans can do easily but what a computer cannot do easily [4], i.e. selecting the corners of buildings with a mouse. The computer, then, does what it can do easily but what a human cannot do easily, i.e. generating large amounts of bug-free VRML code containing potentially

[2]This name is in harmony with the tradition, begun by *Tview* and *Cview*, of naming our internal applications with a letter followed by the word "view." The name *Jview* was originally coined by Casey Songer in an attempt to pay me homage by naming my program after me. He evidently forgot that my first name, Geoffrey, begins with a 'G' rather than a 'J.' I did not have the heart to change the file name after that, out of respect for Casey.

hundreds of floating-point vertex coordinates.

III. Method of Building Extrusion

This section describes a high-level overview of how the *Jview* application works from a user's point of view. Lower-level details concerning the implementation of the application are covered in the next section.

3.1 Preliminary Steps

In order to add buildings to the terrain, the user must first determine the general area wherein he or she wishes to put them; for example, BYU campus, downtown Salt Lake City, or any other location within the Wasatch terrain database. When the user starts the *Jview* program, he or she passes in an image file in PPM format as a command-line parameter. This image is to be a slice of overhead photography, obtained from the terrain database, representing the area to which the user wishes to add buildings. If the user does not have an image representing the desired portion of the terrain, one can be created automatically by using a small program entitled *tiletoppm* (written by Kirk Duffin) which creates a PPM image from any given part of the terrain database. We chose to use PPM images since they are extremely easy to read, write, and edit. The drawback is that they are uncompressed, and may take a bit of time to load.

The first thing the program does once it begins is to prompt the user for the bottom-left (x,y) coordinates for the image previously indicated at the command line. The database, interestingly enough, has its own unique coordinate system in which the Talmage building on BYU campus is at (0, 0). Everything else in the entire world is situated relative to it. For example, in database coordinates, Temple Square in Salt Lake City is located at (-80, 226) and BYU campus is at (-5, -5). In both of these cases, the coordinates represent the southwest corner of the areas spoken of. It is important that *Jview* have access to the coordinates of the region to which buildings are to be added; otherwise the program would not know where to place the buildings relative to the terrain in the VRML file it will create later on.

3.2 Building Selection

After the user inputs the proper coordinates for the image, *Jview* loads the image into memory and displays it on the screen, as shown in Figure 1. At this point, the user may begin placing buildings into the scene. To create a building, the user clicks the left mouse button on the corners of the building's roof within the image. The outline of the building, as indicated by the user, is echoed on the screen. Once the user is finished outlining the

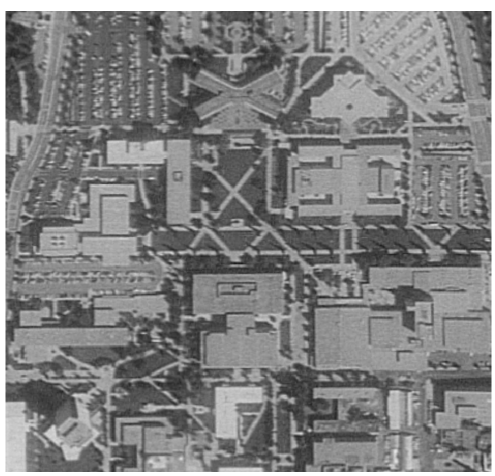

Figure 1. *Overhead view of BYU campus in* Jview.

building's roof, he or she clicks the middle mouse button anywhere within the image to access a pop-up menu used for specifying the building's desired height.

The user may continue clicking on rooftops until he or she has outlined all the buildings that he or she wishes to extrude. If the user makes a mistake and wants to undo a previous selection, clicking the right mouse button anywhere on the image brings up a menu that — among other things — gives the user the option of undoing either the most recently outlined building, or undoing all of them (Figure 2).

As the user goes along, it is a good idea for him or her to draw a map on a piece of paper of all the buildings being outlined, along with what order they are created. This information will be invaluable if the user decides later on to hand-edit the VRML code

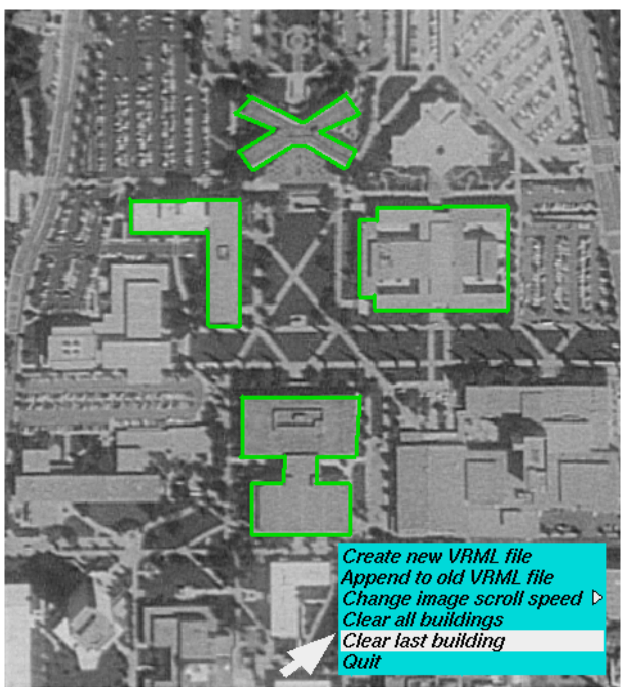

Figure 2. Jview *allows users to undo mistakenly selected buildings.*

produced by *Jview*, which contains comments indicating a sequential number for each building.

3.3 VRML Code Creation

After the user has outlined all the buildings he or she wishes to for a given section of terrain, the user may invoke *Jview*'s most powerful feature, the "VRML Generator." This is accessed by clicking the right mouse button to access the pop-up menu and selecting the option, "Create New VRML File." (Figure 3.) At this point, the program prompts the user to type in a filename, and then *Jview* writes a file to disk in VRML format. This file defines each side of each building as a separate node called an *IndexedFaceSet*, [11] which is the VRML way of defining an arbitrarily-shaped polygon. The sides of the buildings are covered with a generic texture map which the user can optionally replace by hand-editing the *textureURL* fields in the file. (*Jview* inserts comments into the VRML code that explain how this is done.) The roof of each building is textured with its appropriate roof texture as extracted from the overhead photography. In order to include Cline's texture-cached terrain model into the scene, we use a non-standard VRML node, called *HeightGrid*, which is only recognized by our in-house VRML browser, *Cview*. The *HeightGrid* node unfortunately has not yet been adopted as part of the international VRML standard, and hence is not recognized by commercial VRML browsers such as *CosmoPlayer*.

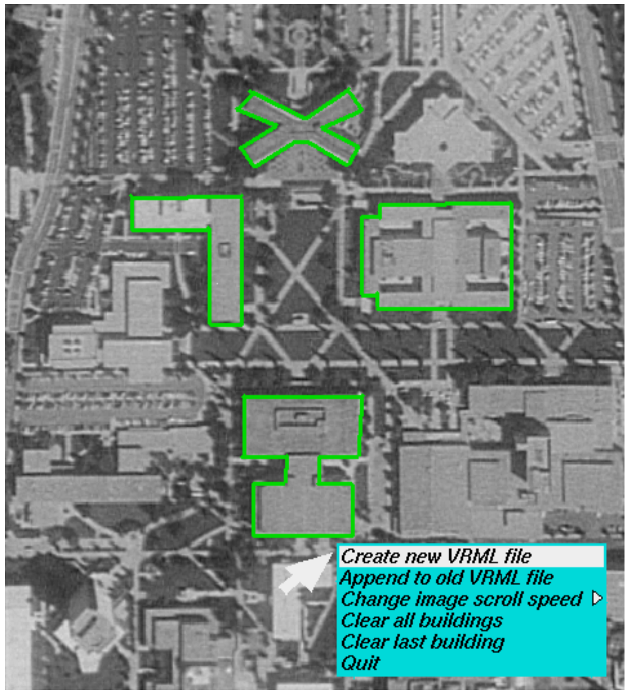

Figure 3. Jview *automatically creates VRML code to model the terrain and buildings.*

After having created a VRML file describing the scene, if the user wishes to add more buildings to that part of the terrain, he or she can simply load the overhead image into *Jview* and add the additional buildings in the regular manner. The only difference is that upon finishing, the user should select the "Append to Old VRML File" option from the right mouse button's pop-up menu. *Jview* will add the additional buildings to the file specified by the user, without overwriting its original data.

Figures 4 and 5 on the following pages show a "before and after" example of how a scene from the terrain database can be enhanced by addition of *Jview*'s automatically created buildings.

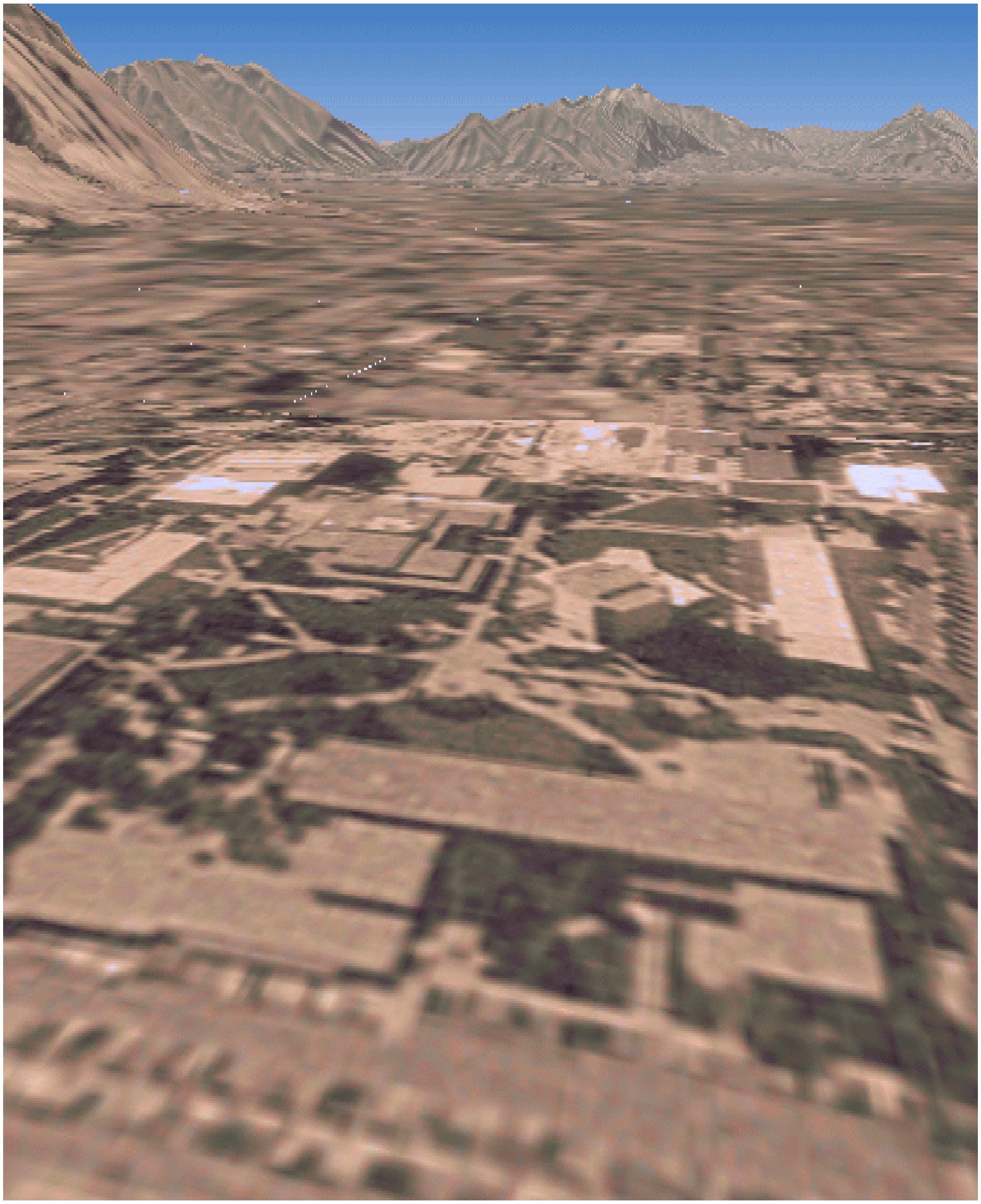

Figure 4. *Snapshot of a 3D fly-by through BYU campus in* Cview. *Notice how there are no polygonally modeled buildings; their presence is only suggested by the textures.*

Figure 5. *The same scene as before, with buildings automatically created by* Jview. *Texture maps for the sides of the buildings were added manually.*

IV. Implementation Details

4.1 Placement of Buildings

When *Jview* outputs the VRML description of any given scene, it uses a preprogrammed VRML template to which it adds the necessary nodes and coordinates. Making this template general enough to accurately define any region of buildings in the database was the biggest challenge of this project. In its first inception, the template did nothing but create a collection of untextured floating polygonal structures representing the relative layout of the buildings defined by the user. Later on, the template was expanded to include the actual terrain underneath the buildings. It then became necessary to scale the buildings appropriately to fit exactly on their proper locations in the terrain. To do this, the x and z coordinates of every building must be scaled by 8/255 (divided by 31.875). The value 8/255 is a result of certain particularities in Cline and Duffin's [2, 5] implementation of the terrain database. The 8 refers to the number of individual data grids in a single unit of the terrain map, and 255 refers to the number of pixels in a data grid. In order to situate the buildings exactly on top of the terrain (not below the terrain, nor too far above it), *Jview* scans through the height data of the terrain database, finds the lowest elevation at any of the building's corners, and sets the base of the building at that elevation. That way, if there are elevation differences in the terrain beneath a building (as on a hill, for example), the base of the building is guaranteed never to "poke out" of the ground.

4.2 Painting the Roof

In computer graphics, images are generated using perfect rectangles. Objects in the real world, however, are rarely so geometrically clean. Hence, mapping a rectangular texture to a polygonal model of a real-world object (a building, in our case) is no simple task. Whenever the user outlines a building in *Jview*, the program calculates the minimum and maximum x and y values of the roof and thereby creates a bounding box for it. *Jview* then finds the corresponding area in the overhead image and makes a small "cut-out" image of it. The program automatically assigns this new image a unique filename and copies it to disk.

Since the roofs are rarely perfectly rectangular, these bounding-box images cannot simply be pasted on top of their respective buildings without some modification. When the time comes to create the VRML file, *Jview* must tell each image how it is supposed to fit onto its building. Thankfully, VRML provides a node, called *TextureCoordinate*, [11] which is designed specifically for the task of mapping images onto irregularly shaped polygons. All *Jview* needs to do is the math.

Here is how it works. In computer graphics, images use a special (s,t) coordinate system analogous to the (x,y) system used in traditional geometry. Values for s and t range from 0 to 1, such that the lower-left corner of the image is at (s,t) = (0,0), the top-left corner is at (0,1), and the top-right corner is at (1,1) as shown in Figure 6.

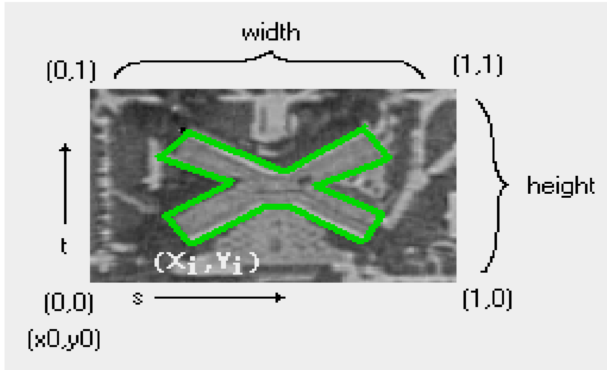

Figure 6. *The administration building on BYU campus, outlined in* Jview.

In the above figure, (x_0,y_0) correspond to (s,t) = (0,0) and represent the lower-left (x,y) coordinates of the roof's bounding box relative to the rest of the image. Then, every (x,y) vertex of the roof is assigned an (s,t) value according to the equation

$$(s_i, t_i) = \left(\frac{x_i - x_0}{width}, \frac{y_i - y_0}{height} \right)$$

where *width* and *height* are the width and height of the roof's bounding box. In this manner, every (x,y) coordinate specifying a roof's vertex has a corresponding (s,t) coordinate specifying how the texture fits onto it.

4.3 Specific Challenges

4.3.1 N-sided Polygons

The *Jview* specification as outlined in section 3 of this paper describes how the user should be able to define buildings of arbitrarily many sides. This idea worked poorly in practice, however. To achieve reliable results under the current implementation, it is best to stick to four-sided buildings with convex shaped roofs.

The original version of *Jview* was written with the implicit assumption that buildings could only have four sides. In general, this is not a significant limitation. To define more complicated buildings, the user only needs to compose the building out of multiple quadrilaterals. This restriction had a side effect that I did not realize at the time: all of my roofs were convex polygons. Of course, not all quadrilaterals are necessarily convex; however, in architecture they generally are.

I later modified *Jview* to accept buildings of any number of sides. This had the side effect that many of my roofs were no longer convex. At this point I discovered a quirk of OpenGL, the graphics API used in *Cview*. OpenGL can apply textures to convex polygons without difficulty; however, it requires that concave polygons be *tessellated*, or subdivided, into simpler polygons prior to rendering. The commercial VRML browser *CosmoPlayer* takes care of this problem automatically; the same cannot be said for *Cview*. It was designed more as a method for integrating Cline's texture-caching algorithm into a more standard rendering engine, rather than as a full-featured VRML browser. To this day, *Cview* is still a work in progress. Nevertheless, because of its limitations regarding concave texture mapping, I archived my enhanced version of *Jview* as a historical footnote and returned to my original "four-sided" code base.

4.3.2 The IndexedFaceSet Node

As mentioned previously, the buildings are implemented in VRML as IndexedFaceSets. For simplicity's sake, *Jview*'s original implementation created two IndexedFaceSet nodes for each building: one for the roof, and one for the sides. This was done because each IndexedFaceSet can have no more than one texture applied to it — and the roofs need to have very specific textures based on their location, while the sides are initially given a generic texture map.

With the basic system in place, I used *Jview* to create a model of most of the buildings on BYU campus. This was done in preparation for a visit from some of our sponsors in April. In order to make my model of BYU more impressive, my advisor Dr. Egbert and I took some digital photographs of the sides of these buildings. In order to apply these photographs as texture maps to the buildings, we had to hand-edit the VRML file that *Jview* had produced. At this point I realized that my scheme of defining all the sides of the building as one IndexedFaceSet was a poor design choice. In order to apply different textures to the different sides of each building, we had to manually subdivide the coordinates of each IndexedFaceSet into separate nodes for each side. It was not necessarily difficult, but it was tedious and time-consuming. It would have been trivially simple to have *Jview* do this work for us from the outset; it simply did not occur to me when I began. After the demos were completed, I updated *Jview*'s source code to create separate IndexedFaceSet nodes for every side of every building.

V. Future Work

5.1 N-sided Polygons

While *Jview* is effective at what it claims to do, there are many possibilities for embellishment that could make it both more powerful and more user-friendly. An obvious example that has yet to be implemented is the power to define n-sided buildings with concave polygons as roofs, as described in section 4.3.1 above.

5.2 Terrains Determined at Run-time

Although the user can tell *Jview* at run time (via a command-line parameter) which image to display on the screen for use in picking buildings to extrude, the actual portion of terrain upon which the buildings will sit in the VRML file is determined at compile time. This is because the terrain models are enormous; if the user wishes to extrude buildings in downtown Salt Lake City, he or she is not likely to want to load in an overhead image of the entire Salt Lake valley and scroll through it to find the area containing the desired buildings. Rather, the user will load in a relatively small image (about the size of the user's computer screen) representing the immediate area from which buildings are to be extruded. *Jview* does not know where this image fits into the terrain database until the user inputs the lower left (x,y) coordinates of the image in relation to the terrain, as described in section 3.1 of this paper.

Finally, in addition to knowing the offsets of the particular image the user has loaded for the purpose of defining buildings, *Jview* needs to know the portion of the terrain database upon which the buildings will physically sit. The Wasatch terrain database is enormous; it is rare that someone would want to show all of it at once. Rather, users typically want to show off a particular region of the database, such as Salt Lake City, Provo, or Park City.

Instead of forcing the user to input all three pieces of information (the image used for defining buildings, the location of that image relative to the larger terrain, and which terrain that will be), it was simply a design choice on my part to make choice of terrain — in my opinion, the part that is least likely to change from session to session — a compile-time switch instead of a run-time option. Doubtlessly there could be a more user-friendly way to do this, but since *Jview* was created as a research tool for a limited audience, I felt it was better to focus on the actual research portion of the project rather than on a pretty interface.

5.3 Using *Jview* More Than Once for the Same Scene

In section 3.3, I mention how *Jview* can be used to add buildings to a scene that already contains some extruded buildings. In order to use this option, the user must include an additional command line parameter, '-p', after the image filename when the program is launched. This has to do with a peculiarity in which *Jview* selects filenames for the roof textures. There are perhaps more convenient ways to do this without making the user enter an additional parameter; however, the current implementation has been sufficient for our needs at this time.

5.4 Automatic Edge Detection

As *Jview* is currently implemented, the user must click on the corners of the buildings' roofs in order to define that building's shape. The user's mouseclicks are directly translated into VRML coordinates when the time comes. A future feature of *Jview* might be to employ some form of image-based edge detection on the buildings so that the user has some help in picking the vertices. Two possibilities have been discussed: either have the user drag rubberbanded lines that "snap to" the edges of the buildings, or have the user click somewhere on the building and allow *Jview* to detect all of its edges automatically. Generalized algorithms for both of these approaches [6,7, and 9] have been proposed; their inclusion into *Jview* would be a helpful convenience to its users.

VI. Conclusions

This thesis describes a method of extracting three-dimensional buildings from flat overhead photography. It uses both user input and automatic creation of VRML code to accomplish this. It is designed to be used in tandem with the Wasatch terrain database and texture-caching technology currently in use at the Advanced 3D Graphics Lab at BYU. Although it was created specifically for the Virtual Olympics project, its techniques are general enough to be applied to any VRML-based application where three-dimensional models must be implied from flat images.

References

[1] David Cline and Parris K. Egbert. "Interactive Display of Very Large Textures." *IEEE Visualization 1998*, pp. 343-350.

[2] David Cline. *Interactive Display of Very Large Textures.* Master's Thesis 1998. Department of Computer Science, Brigham Young University.

[3] David Cline and Parris K. Egbert. "Terrain Decimation through Quadtree Morphing." Submitted for publication in *Transactions on Visualization and Computer Graphics.*

[4] Paul E. Debevec, Camillo J. Taylor, Jitendra Malik. "Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach." *SIGGRAPH '96,* pp. 11-20.

[5] Kirk L. Duffin. *Image-Based Modeling Techniques for Virtual Environments*. Doctoral Dissertation 1999. Department of Computer Science, Brigham Young University.

[6] Bryan S. Morse, Parris K. Egbert, and Jaymes Wilks. "User-Assisted Rectangle Selection Using the Hough Transform." *Proceedings of the IASTED International Conference of Signal and Image Processing 1999,* pp. 26-29.

[7] Lowell Jackson Reese. *Intelligent Paint: Region-Based Interactive Image Segmentation.* Master's Thesis 1999. Department of Computer Science, Brigham Young University.

[8] Richard Schrand. *Canoma Visual Insight.* The Coriolis Group, 2000.

[9] Eric Sokolowsky, Casey Songer, Bryan S. Morse, Parris K. Egbert. "User-Assisted line-segment extraction using the Hough transform." *Computer Vision, Pattern Recognition, and Image Processing 1998.*

[10] *3D Studio MAX R2 User's Guide*. Autodesk, Inc. 1997.

[11] "VRML97 Specification ISO/IEC 14772-1:1997." On-line posting. http://www.vrml.org/techicalinfo/specifications/vrml97/index.htm 29 March 1999.

[12] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide, 2nd edition*. Addison Wesley Developer's Press. Reading, Massachusetts. 1997.

[13] Brian Zaugg and Parris K. Egbert. "Visibility Culling for Large Terrain Data Sets." Submitted for publication in *IEEE Visualization 2000*.