TOADS: A Two-Dimensional Open-Ended Architectural Database System

by

Samuel R. Madden

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

© 1999 Samuel R. Madden. All Rights Reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Signature of Author: ____________________________________________

Department of Electrical Engineering and Computer Science
May 13, 1999

Certified By: ____________________________________________________

Nathaniel Durlach
Senior Research Scientist, MIT Research Laboratory of Electronics
Thesis Supervisor

Accepted By: ___________________________________________________

Arthur C. Smith
Chairman, Department Committee on Graduate Students
TOADS: A Two-Dimensional Open-Ended Architectural Database System

by

Samuel R. Madden

Submitted to the Department of Electrical Engineering and Computer Science on May 13th, 1999, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

The TOADS system is presented as an innovative tool for building interior-space virtual environments in two-dimensions. Existing virtual environment design tools typically operate in three-dimensions, which makes it difficult to manipulate objects on the inherently two-dimensional computer screen. TOADS allows nearly the same functionality as those three-dimensional systems in an easy-to-use two-dimensional environment. Users edit and enhance DXF floorplans with height and texture information. The software includes an inference engine which automatically identifies doors in the floorplan and generates openable polygons in the final environment. It also includes a sophisticated mechanism for embedding complex textures, such as transparent windows, at arbitrary heights in wall polygons. The entire interface is integrated with software that drives a custom texture-acquisition device. This device consists of a rack-mounted camera which captures narrow bands of textures and tiles them together to form long, continuous swaths of texture. This thesis summarizes these tools and their implementation, along with a discussion the basic software structure and a presentation of algorithms for properly tiling, enhancing, and applying textures. A tool which generates VRML environments from the augmented DXF files is described, and examples of generated environments are given.

Thesis Supervisor: Nathaniel I. Durlach
Title: Senior Research Scientist, M.I.T. Research Laboratory of Electronics.
Acknowledgements

This work was motivated based on discussions during the Spring of 1998 with Tom Weigand, Nat Durlach, Kari Anne Høier Kjølaas, and Andrew “Zoz” Brooks. All deserve credit for contributions to the program concept. Throughout the development process, Tom offered consistent and worthwhile feedback as well as invaluable hardware expertise while building the texture rack. Zoz assisted with early versions of the ToVRML tool and provided excellent feedback on program interface and design. As an advisor and willing provider of all manner of assistance, Nat was faultless. Josh Silberman deserves a mention for being willing to think about, discuss, and offer useful advice as to how to approach a number of software and hardware problems.

To family, friends, and others whose path I may have crossed during twenty-two years of life, thanks for letting me get this far, for putting me into this world with a good head and sound footing, for giving me something to look forward to beyond TOADS and M.I.T.
## Contents

TABLE OF FIGURES ......................................................................................................................... 7

1. INTRODUCTION ............................................................................................................................. 8
   1.1 THE STATE OF THE ART .................................................................................................................. 8
   1.2 MORE ON TOADS .......................................................................................................................... 9

2. EXISTING TOOLS ............................................................................................................................. 11
   2.1 DXF .................................................................................................................................................. 11
   2.2 VRML .............................................................................................................................................. 11
   2.3 3D MODELING ENVIRONMENTS ............................................................................................... 12
   2.4 ARCHITECTURAL WALKTHROUGH PROGRAMS ...................................................................... 12
   2.5 THE BERKELEY BMG SYSTEM .................................................................................................. 13
   2.6 GAMES AND GAME EDITORS .................................................................................................. 13

3. DESIGN CONSIDERATIONS .............................................................................................................. 14
   3.1 TWO-DIMENSIONS IS INTUITIVE ................................................................................................. 14
   3.2 PHOTOREALITIC VS. TRULY THREE DIMENSIONAL TEXTURE ACQUISITION ......................... 14
   3.3 TEXTURE ACQUISITION .............................................................................................................. 15
   3.4 HABITRAILS ............................................................................................................................... 15
   3.5 DATA INFERENCES ..................................................................................................................... 16
   3.6 DEFAULTS AND OBJECT PROPERTY INHERITANCE .............................................................. 16
   3.7 INTERMEDIATE 3D FILES ........................................................................................................ 17

4. IMPLEMENTATION ............................................................................................................................. 17
   4.1 MAIN TOADS PROGRAM .............................................................................................................. 17
   4.2 DXF PARSER .................................................................................................................................. 19
      4.2.1 DXF Format Extensions ........................................................................................................ 20
      4.2.2 Parser Limitations ................................................................................................................ 21
   4.3 MAIN UI ....................................................................................................................................... 21
      4.3.1 Main UI Implementation ....................................................................................................... 23
      4.3.2 Exception Handling .............................................................................................................. 24
      4.3.3 Memory Management ......................................................................................................... 25
      4.3.4 Performance ....................................................................................................................... 25
   4.4 TOOLS ......................................................................................................................................... 26
      4.4.1 Basic Object Manipulation .................................................................................................. 27
      4.4.2 Lights .................................................................................................................................... 27
      4.4.3 Origin Placement ................................................................................................................ 28
   4.5 OBJECT SETTINGS ...................................................................................................................... 28
      4.5.1 Object Length Dialog .......................................................................................................... 28
      4.5.2 Object Info Dialog .............................................................................................................. 28
      4.5.3 Object Types ....................................................................................................................... 29
   4.6 DEFINING POLYGONAL OBJECTS ............................................................................................... 29
   4.7 ROOMS ....................................................................................................................................... 30
Table Of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Module dependency diagram for main Toads program.</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Module dependency diagram for the DXF parser.</td>
</tr>
<tr>
<td>Figure 3</td>
<td>A screenshot of the Toads UI with a single DXF file open.</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Module dependency diagram for exceptions in the Toads program.</td>
</tr>
<tr>
<td>Figure 5</td>
<td>The tool palette.</td>
</tr>
<tr>
<td>Figure 6</td>
<td>The object length dialog.</td>
</tr>
<tr>
<td>Figure 7</td>
<td>The object info dialog.</td>
</tr>
<tr>
<td>Figure 8</td>
<td>The polyline info dialog includes a base altitude and top-texture.</td>
</tr>
<tr>
<td>Figure 9</td>
<td>A texture wrapped around a four-sided polygon.</td>
</tr>
<tr>
<td>Figure 10</td>
<td>The room settings dialog.</td>
</tr>
<tr>
<td>Figure 11</td>
<td>The preferences dialog.</td>
</tr>
<tr>
<td>Figure 12</td>
<td>The texture dialog.</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Three views of a repeated texture on the same floor.</td>
</tr>
<tr>
<td>Figure 14</td>
<td>The texture design dialog.</td>
</tr>
<tr>
<td>Figure 15</td>
<td>A designed texture showing a transparent window in a wall.</td>
</tr>
<tr>
<td>Figure 16</td>
<td>The difference_rects algorithm.</td>
</tr>
<tr>
<td>Figure 17</td>
<td>The compute_background algorithm.</td>
</tr>
<tr>
<td>Figure 18</td>
<td>The grabber rectangle.</td>
</tr>
<tr>
<td>Figure 19</td>
<td>The texture acquisition dialog</td>
</tr>
<tr>
<td>Figure 20</td>
<td>The robot settings dialog</td>
</tr>
<tr>
<td>Figure 21</td>
<td>The source, image, and compression settings panels.</td>
</tr>
<tr>
<td>Figure 22</td>
<td>The frame list window and the frame info window.</td>
</tr>
<tr>
<td>Figure 23</td>
<td>An improperly adjusted panorama window.</td>
</tr>
<tr>
<td>Figure 24</td>
<td>The compute frame size dialog</td>
</tr>
<tr>
<td>Figure 25</td>
<td>The X, δ, and Δ parameters and their physical significance.</td>
</tr>
<tr>
<td>Figure 26</td>
<td>A properly tiled panorama.</td>
</tr>
<tr>
<td>Figure 27</td>
<td>The vertical-compression calibration image.</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Graph showing the linearity of image compression.</td>
</tr>
<tr>
<td>Figure 29</td>
<td>A door texture, compressed and uncompressed.</td>
</tr>
<tr>
<td>Figure 30</td>
<td>An image before and after noise correction.</td>
</tr>
<tr>
<td>Figure 31</td>
<td>A sample image from the rack camera.</td>
</tr>
<tr>
<td>Figure 32</td>
<td>The texture rack.</td>
</tr>
<tr>
<td>Figure 33</td>
<td>Four lasers allow texture-rack to be aligned.</td>
</tr>
<tr>
<td>Figure 34</td>
<td>Photographs of the texture rack.</td>
</tr>
<tr>
<td>Figure 35</td>
<td>Module dependency diagram for 3D export classes</td>
</tr>
<tr>
<td>Figure 36</td>
<td>Texture and ceiling height inheritance diagrams.</td>
</tr>
<tr>
<td>Figure 37</td>
<td>A model and its convex hull.</td>
</tr>
<tr>
<td>Figure 38</td>
<td>Screen shot of the lobby of the 7th floor of building 36.</td>
</tr>
<tr>
<td>Figure 39</td>
<td>The building 36 7th floor model.</td>
</tr>
<tr>
<td>Figure 40</td>
<td>The fridge and microwave.</td>
</tr>
<tr>
<td>Figure 41</td>
<td>View past mailboxes and offices.</td>
</tr>
<tr>
<td>Figure 42</td>
<td>Secretary's desk and printer.</td>
</tr>
<tr>
<td>Figure 43</td>
<td>Coordinate axes for intermediate 3D file format.</td>
</tr>
</tbody>
</table>
1. Introduction

The Three dimensional Open-ended Architectural Database System (TOADS) is a general purpose software tool for cataloging and organizing data in three-dimensional environments. Its main goal is to manage the wealth of data which is available in architectural environments; this includes, but certainly is not limited to, floorplans, wall, ceiling, and floor textures, photographs, historical documents, and locations of movable objects such as furniture, and lighting. With TOADS, users can manage these data in a coherent, easy-to-use environment, and then select relevant subsets of the data and export them to less manageable contexts, such as three-dimensional virtual environments (VRML, Inventor, etc.) or HTML-style documents containing text and pictures connected by hypertext links.

The software tools which have been created to date consist of a prototype suite which is designed to demonstrate the feasibility and usefulness of such a project. In its current incarnation, the TOADS interface consists of a 2D editor for DXF-format blueprint files, combined with several support programs that facilitate texture-capture and application. In addition to textures, these blueprints can be augmented with further information, which, in the current TOADS release, consists primarily of ceiling heights and lighting. These models can then be exported to VRML files and explored using a standard web-browser. Additional features, such as support for other input and output file formats and textual information linked to the blueprint, are not included in the initial implementation, but the software is designed with these features in mind.

1.1 State of The Art

Before launching into a detailed explanation of the TOADS system, it’s important to understand how it fits into the context of existing 3D tools and why it is an important addition to the set of tools already available.

Virtual reality has become one of the most media-hyped terms in computer science today. A rash of computer generated movies, beginning with “Toy Story” and including the more recent “Antz” and “A Bug’s Life” have popularized the notion that computers are capable of generating immersive virtual universes. This is furthered by the huge number of first-person, three-dimensional video-games, among them “Doom”, “Descent”, and “Unreal”. These applications of 3D graphics technology have made most of society aware of the possibilities that virtual reality holds. Trends in academia have drawn from popular ideas of science fiction and begun to envision an electronic world in which vast numbers of Internet users co-exist in a three-dimensional virtual community. To this end, recent literature has focused on managing complexity, bandwidth, and integration issues involved in building and rendering virtual worlds.

Researchers have taken two major approaches to this problem: the first has to do with keeping track of which objects in a world are visible in an efficient manner and spending compute time on rendering visible or probably visible objects [5, 15, 16]. The second has to do with limiting the size of models by linking them together via hyperlink-style visual portals in the environments; smaller models mean faster performance, though issues of discontinuity arise when users move across the boundaries of models [17].

3D movies are generated by throwing an amazing amount of computing power towards rendering fixed scenes which will later be played back to the viewing audience. Computer games, conversely, are rendered in real-time and give the player control over the action – which scenes are explored and where the character moves. For this reason, they’re considerably less realistic looking than the pre-rendered scenes in Hollywood movies. The engines which power these computer games are typically commercially designed pieces of software, which are not available for other developers (or consumers) to use to construct their
own virtual environments. Furthermore, these engines are highly specialized towards achieving reasonable performance on consumer class machines in a game environment.

Academics who are interested in creating real-time virtual environments of reasonable quality have thus had a limited set of tools available to them. As a de facto standard, SGI's OpenGL, OpenInventor, and Performer languages have been adopted [3,13]. OpenGL forms the low-level interface between the hardware and the programmer; OpenInventor adds support for scenes, clusters of objects which can interact and be animated; Performer builds on Inventor by adding support for management of complex environments and input hardware such as headtrackers and joysticks. OpenGL is used to describe three-dimensional scenes which consist of simple geometric primitives such as polygons, spheres, cones, and curved surfaces. Each of these objects is given surface characteristics, such as reflectivity, transparency, and texture. Texture refers to a two-dimensional array of colored pixels which is stretched across each of the underlying polygons. To create the most realistic looking 3D environments, textures are made from photographs – such photo-texture environments are referred to as “photorealistic”. The SGI tools are an extremely important piece of the 3D software tools set because they take care of the details of interfacing between hardware and software; programmers simply specify a description of the three-dimensional world in terms of basic primitives and the location of the viewer and OpenGL (the lowest-level tool) renders the scene on the users computer. OpenGL toolkits are available for all computer platforms. SGI, in association with a number of commercial and academic institutions, recently developed a formal specification for 3D environments called Virtual Reality Modeling Language (VRML) [3]. Typically implemented on top of OpenGL, VRML is an OpenInventor-like language designed for making 3D environments accessible to users of desktop machines via the Internet. It’s important because it provides an accepted, platform-independent, high-performance rendering engine which is extremely easy to use. As with HTML, VRML allows users with little programming experience to create dynamic, high-quality worlds which can be explored in real time. Cosmo Software, a spinoff of SGI, currently distributes CosmoPlayer, a VRML viewer for Netscape Navigator which allows Internet users to download and interact with virtual reality scenes from within their web browser.

VRML, Inventor, and other proprietary languages for describing virtual environments are not, by themselves, adequate tools for most users who wish to create virtual scenes without learning a complex programming-style language [14]. Most end-users will not invest the time required to learn VRML, and visions of a world wide virtual environment will never be realized without user-friendly development tools. This is where TOADS fits in – as a tool to facilitate the rapid generation of high-quality, photorealistic virtual environments. Thus, TOADS, as a VRML authoring tool, is to VE design as HTML authoring tools are to web page design – it provides a non-programming based environment where most architectural VE’s can be built.

1.2 More On TOADS

Previous work in the MIT RLE Sensory Communication Group suggests that manually creating photorealistic three-dimensional environments is painstaking and tedious [2]. This is due primarily to two concerns: first, manually describing each polygon as a sequence of numeric points that form walls, doors, windows, and ceilings is extremely slow. Once the model has been built, making changes to it is relatively difficult because of the density of the 3D data and the fundamentally 2D nature of the computer screen – even with a graphical editor, locating, selecting, and accurately moving or resizing a particular polygon can take several minutes.
The second major problem with such environments has to do with acquiring, cropping, and applying textures. Typically, textures are obtained via a digital camera, then cropped in a digital photography program to be the correct size and orientation for the polygon they're to be applied to. Then, the user is required to manually associate each polygon with one of the textures, with no way of logically applying default textures to all polygons of a certain class—e.g., making all of the doors in a building have the same appearance.

TOADS provides a solution to both of these problems. It is designed to work with existing architectural data formats (such as DXF) in which blueprints are commonly stored. This provides the user with a two-dimensional representation of the building being modeled. Furniture (and other movable objects) can be specified as two-dimensional polygonal outlines and then textured with photographs to create a realistic appearance. All objects can be readily moved and resized.

The single largest benefit of TOADS, however, lies in its ability to facilitate texture acquisition and application. In conjunction with the development of the TOADS software system, an effort is underway to build hardware devices that acquire large strips of texture by moving a camera along a section of wall (by placing the camera on either a motorized track or a semi-autonomous robot.) TOADS includes a hardware interface which allows it to control these devices, and part of the software system is a texture-tiling application which precisely lines up frames from the digital camera to create wall-sized textures which do not require any additional user-editing. Furthermore, because the system is intelligent about the floorplans it's working with, it can automatically detect some kinds of architectural objects (for example, doors are nearly universally represented as arcs in architectural plans) and apply intelligent texture defaults to those objects. Users can associate objects with particular categories (such as “rooms” or “windows”) and apply textures to all objects in a particular grouping.

A third advantage of TOADS lies in its flexibility: it allows any geometric data with a two-dimensional-representation to be edited and exported into arbitrary three-dimensional file formats. Although the system is initially directed towards manipulating blueprints, adapting it to work with geographic data (e.g. open-space or undersea environments) is as simple as writing a parser for those data. Two and three-dimensional USGS data is available for the entire world—TOADS could import and enhance such data to generate complete three-dimensional virtual environments: land could be textured, lighting effects added, and polygonal objects such as houses and trees could be drawn in.

Yet another importance of TOADS lies in its ability to handle dynamic environments. Because objects can be moved, rotated, and resized, it’s easy to change the appearance of an environment without having to manually tweak polygons in three dimensions. Because TOADS saves files in a compact form, and can open and save quickly, it’s possible to keep many versions of an environment around, each with slightly a different arrangement of objects, textures, and heights.

What is the importance of being able to quickly generate 3D environments? Our work is immediately heading towards experiments on the training of spatial behavior, in which human subjects are trained in virtual environments of real buildings and then taken to the actual building and asked to perform tasks that depend on the spatial knowledge acquired by the subject in the virtual environment. Among the spaces we’re considering modeling is a very cluttered warehouse; making a model of such a building by hand would be tedious and hugely time consuming, but should be considerably easier with TOADS.
On a larger scale, the system is significant in any study in which a virtual environment is used. Not only does it greatly reduce the time to build the environment, it provides a simple interface to make and catalog changes as experiments evolve.

2. Existing Tools

3D graphics tools have existed for many years, and there are such a wide variety that it is at times hard to separate one from the other. In order to properly place TOADS amongst other tools, it’s important to understand how those tools work and what they do. Some mention has already been made of Inventor and VRML, but a bit more background is important to understand why some of the design decisions in TOADS were made.

2.1 DXF

DXF is the data format (front-end) that initial versions of TOADS support. The DXF specification is available from freely AutoDesk [6]. DXF was selected for several reasons:

- **Nearly universal support**: DXF support is available for many different kinds computers and platforms.
- **Large existing set of data available**: DXF-format blueprint files are commonly available many different buildings. MIT provides DXF files for all of its buildings.
- **Extensible**: DXF provides a comment mechanism that can be used to embed TOADS specific data while still allowing other DXF parsers to view the files.
- **Easy to parse**: DXF is a text-based format that is very easy to parse using a simple top-down, finite-state based parser.

DXF is a two and three dimensional drawing format. Objects are separated into named layers, and object-primitives can be clustered into named blocks. The principal object primitives are lines, circles, arcs, polygons (two and three dimensional), and text. It provides a variety of drawing options which are designed with CAD and architectural needs in mind.

2.2 VRML

VRML is currently the VE file-format of choice for TOADS files. After a model has been created and enhanced with appropriate textures and ceiling heights, it is exported to the TOADS intermediate 3D format. This intermediate format can be converted to VRML using a supplied conversion tool, or to any other 3D file format using a user-provided tool and the TOADS intermediate file interfaces.

As mentioned previously, VRML was selected because it’s a nearly universally supported standard for virtual environment creation. Increasingly, it’s the format in which commercial VE’s are designed. Browsers are available for all platforms, and active research is being done to improve its performance and capabilities [17,18,19].

The VRML format offers a number of useful features. Firstly, the format is easy to write, consisting of text-based statements to define and transform objects. It’s powerful, supporting fully textured environments (with transparency) with complex lighting, collision detection (in Version 2.0), and interactivity.

Interactivity is primarily accomplished via embedded JavaScripts (Netscape Corporations Java-like language originally developed to allow web-page authors to quickly insert scripts into their pages.) Objects
can be linked to JavaScripts which are activated when the user approaches or clicks on them. Scripts can have a variety of effects, from initiating a simple animation to teleporting the user to a new location in the environment.

Interactivity is important to the TOADS system to allow openable objects like doors to be supported. Since Javascripts are easily embedded into VRML, TOADS can build environments with openable doors without requiring the user to compile source code.

VRML and DXF are important because they allow TOADS to operate within the domain of existing standards; users can continue to use the file formats and tools they are familiar with and still gain the benefits of an advanced, UI driven VE system like TOADS.

2.3 3D Modeling Environments
These are tools whose primary purpose is to generate three-dimensional objects or renderings of three-dimensional scenes. They often include tools to manipulate objects in three dimensions, apply lighting and textures, and generate high-quality ray-traced renderings. They differ significantly from TOADS in that, rather than concealing three dimensional details from the user, they work hard to display those details in their full glory. These are the most prevalent of three-dimensional design tools; they’ve existed for a number of years on all desktop platforms.

2.4 Architectural Walkthrough Programs
There are several programs designed to create 3D walkthroughs for quick prototyping of architectural environments. One of most powerful of these is Virtus Walkthrough, which presents the user with a two-dimensional overhead view of the model they are creating with a side-by-side 3D view of the environment. It allows textures to be applied to walls, lights to be specified, and a variety of polygonal objects to be defined. It can also import DXF files, and can export to VRML (version 1.0). In this respect, it is quite similar to TOADS and serves as a minimum for what TOADS should do. Despite the initial similarities, however, it is lacking in a number of areas:

- No facilities for texture management and acquisition. Walkthrough assumes that textures exist a priori and has no support for cropping and tiling textures within the program, much less an interface to hardware specifically designed to acquire textures. It does come with a large selection of pre-defined textures; unfortunately, such files are of little use when trying to design a photo-realistic environment.
- Limited expandability. Although the initial TOADS implementation may only support DXF import and VRML export, its design is such that plugging in new front and back ends is straightforward for anyone with moderate programming experience. Walkthrough is targeted at the commercial sector, and thus provides no room for programming-literate users to expand it.
- No Intelligent data management. Walkthrough provides no facilities to automatically or manually group objects beyond the creation of simple layers. Objects can’t have classes (e.g. door/wall/window) and thus can’t be textured or lit by object type. Even layers don’t offer default textures or heights. There is no facility whereby objects can be subdivided into rooms, and the program does no intelligent processing to eliminate user-tedium (such as automatically classifying all quarter-circles as doors and automatically generating doors which can be clicked to open or close in the 3D environment.)
- Poor Performance. Walkthrough took more than 30 minutes to import a 2000 polygon DXF floorplan on a Pentium 90. TOADS opened the same file in under a minute on a comparable-speed PowerMac 7200 (PowerPC 601 at 75 Mhz). Though the Virtus software did present a 3D view of the file immediately, it is
so slow at managing a large number of polygons as to be almost unusable except on the highest performance PC's. TOADS runs well on all Macintosh computers built within the last five years. The fully-textured models it exports require a speedy machine, but most of the building and design can be done on much lower performance machines.

*Walkthrough* does offer a 3D view of the model as it is being built. There are no plans to implement such a feature in TOADS, primarily because VRML and Inventor browsers are available for all desktop machines and can be used to prototype a model when necessary.

### 2.5 The Berkeley BMG System

The Building Model Generator (BMG) is a tool to automatically convert 2D floorplans into 3D environments [4]. It does some of what TOADS does, in that its input is a floorplan and its output a 3D environment. It includes some sophisticated two-dimensional analysis which attempts to determine which areas of a model constitute rooms and corridors, and draws some of the same inferences about what kinds of lines make up windows and doors as TOADS. BMG doesn't include a user interface which allows the same sort of detailed manipulation of 2D models, and it is lacking any of the tools TOADS includes to import, manipulate, and rapidly apply 3D textures.

The BMG system is extremely interesting in that a large amount of work has been put into generating the three-dimensional models once the two-dimensional floorplan is fixed. It includes algorithms to automatically detect rooms and corridors (which TOADS will not initially include), as well as code to eliminate inconsistencies such as overlapping polygons and non-joined corners (which TOADS will only include in a rudimentary way). If possible, incorporating some of this work into TOADS would be beneficial.

Because the BMG system exists and heavily focuses on the 3D export engine while ignoring the 2D user-interface that TOADS focuses on, the two systems complement each other nicely.

### 2.6 Games and Game Editors

The fast 3D graphics cards that come with most of today's desktop PC's have made remarkably realistic 3D virtual environments available in the form of first-person shoot-'em-up arcade games. *Quake II, Tomb Raider II, Unreal, Duke Nuke-'Em,* and a multitude of other aggressive titles are available. Some of these games have been enhanced by clever players around the world with level-editors that do some of what TOADS will do, although they are not built with the same extensibility and texture-acquisition options in mind, but are instead packages with features very specific to the game editor for which they were designed. There are quite a number of such packages, ranging from commercial applications that cost as much as the games themselves to lightweight freeware programs which offer only a minimum of features for quick level editing.

One such program is Quiver, a Macintosh based editor for Quake maps. It lies somewhere between a 3D modeler and Virtus *Walkthrough*; it's remarkable as a piece of shareware software. Users draw polygons in a 2 or 3D window, apply textures to the polygons, join them together to make rooms and passageways, and then export their models into Quake. It lacks the import and extensible export features of TOADS and has a tendency to create very complicated-looking pictures which are hard to manipulate quickly when models get large. Like *Walkthrough*, it includes real-time 3D previewing features.
Because modern 3D games are driven by very powerful graphic engines, they provide a potential viewing environment for TOADS models – writing an export tool for Quake would be straightforward, and would open the door for a variety of spatial learning experiments ranging from basic studies of learning in videogames to military readiness exercises.

3. Design Considerations
Before looking closely at the implementation of TOADS, it’s useful to examine some of the higher level concepts which motivated the implementation.

3.1 Two-Dimensions is Intuitive
The main interface for the TOADS system is a two-dimensional, overhead view of the floorplan of the building. This is different from the conventional view of a virtual environment – most modelers present a three dimensional view. The problem with a three-dimensional interface is that the computer screen is inherently two-dimensional; thus, most interfaces are clunky and difficult to use, requiring the user to select separate tools for panning (moving in the xy plane at the current depth), zooming (changing the depth), and rotating (changing the viewing angle). This limitation makes it hard to locate specific points and place objects accurately in three-dimensions; for a complex architectural model, simply placing all the walls would require many hours. Conversely, a two dimensional interface is extremely easy to manipulate using familiar computer metaphors – the standard mouse pointer is sufficient to locate and place any line within a flat model.

Of course, there are some situations in which a three-dimensional interface is necessary; if objects exist at many different depths or have widely varying heights, there is no reasonable two-dimensional interface for model-viewing. Fortunately, in the domain of architectural spaces, most objects lie on the plane of the floor, and most of the walls have the same height. A two dimensional interface makes it extremely simple to move and place objects and walls, and navigating the model is intuitive and familiar to users of graphical user-interfaces. By allowing users to specify heights for walls and objects, with defaults that let most of the walls have the same height, little is lost in terms of functionality or realism when working in two-dimensions with architectural models.

There are some environments where it’s necessary to create objects which don’t lie on the floor. If these objects don’t require user interaction and aren’t particularly deep, they can be accurately approximated by photorealistic textures (see section 3.2 below.) There are a few conspicuous cases where these conditions can’t be met; doors need to be opened and may have small thresholds such that they aren’t flush with the floor; windows need to be transparent and usually don’t stretch all the way from the floor to ceiling. To solve this problem, TOADS provides an interface which allows the position of transparent and openable objects to specified within the context of a larger wall – the term “designer textures” is used to refer to such multi-piece textures. This interface is summarized in Section 4.9 – “Texture Application and Acquisition.”

3.2 Photorealitic vs. Truly Three Dimensional
A significant amount of time has been spent by serious computer scientists on the problem of how to manage the immense complexity which arises in large virtual environments, particularly when the entire model is represented as polygons [15, 16, 17]. If every chair, computer, bookshelf and book in an environment is drawn as an individual polygon, a single floor of a small office building would require tens of thousands of different polygonal objects. This complexity is painful in two ways: first, generating such
In TOADS, we work around this problem by replacing many polygons with a single photographic-quality texture of those polygons. For example, a bookshelf with books on it is represented by a single picture rather than separate geometric objects for each book and the shelf. In many cases, it’s safe to carry this process even further, so an entire wall, with bookshelves, pictures and boxes along it is represented as a single, long swath of texture. This is a standard technique for improving performance in complex virtual environments [1].

Of course, there are some drawbacks to this representation. Most significantly, there is no sense of depth in textures, so as a viewer gets close to a wall, all objects on it will appear to be flat; there is no motion parallax and all shadows and reflections are static. Also, there can be no interactivity in such a world – objects are fixed in their location and cannot be moved or rotated.

TOADS attempts to rectify some of these problems by allowing users to specify independent polygonal objects in the model. Of particular interest are objects with which the user interacts: doors need to be opened and closed and are thus represented as separate polygons which can be clicked on; windows are transparent and should allow the user to see into the rooms which are behind them. There may also be cases where an object has too much depth or is too far from a wall to be included in a flat texture. In these cases, users can specify polygons with their own textures at arbitrary locations within a model.

3.3 Texture Acquisition

Because TOADS is designed to use photo-realistic textures, a good interface for obtaining and applying the textures is necessary. Rather than requiring the user to manually take photographs, crop them in a photo-editing program, and then associate polygons with photographs, an interface to custom-designed texture acquisition hardware is built into TOADS. The hardware currently consists of a track-mounted camera which moves along a portion of wall, capturing narrow bands of texture as it moves. TOADS tiles these bands of texture into a larger texture file; because the camera is moving at a constant speed and its distance from the wall is known, it’s possible to determine exactly how wide to make each band so that a seamless texture results. The scanning process is simple: the user clicks on the polygon he wishes to acquire a texture for, moves the hardware to the location he is scanning, and clicks a button to start the scanning process. TOADS automatically tiles the texture and applies it to the section of wall being scanned. The user doesn’t have to worry about cropping, tiling, or scaling the images to produce photo-realistic textures.

To further generalize the texture acquisition process, the software which obtains photographs from the environment is implemented as a separate application from the main TOADS program. This allows other texture acquisition hardware to be used without modification to TOADS by developing a small interface program and using a standardized software interface.

3.4 Habitrails

A solution to the problem of flat texture panels which approximate walls with some depth is to introduce paths that constrain the users movement and prevent him from approaching close enough to a complex scene to perceive that it does not actually contain depth. These paths are referred to as habitrails (a term from animal psychology reflecting the paths which many animals naturally carve through their living-
spaces) to indicate the areas of the model which the user is actually allowed to explore. The TOADS system allows habitrails to be defined in a model; they are automatically set up to be transparent walls which limit the user's movement.

### 3.5 Data Inference

The inference engine is what makes TOADS easy to use. It eliminates several very repetitive tasks which no person wants to do.

The TOADS data inference engine is responsible for making decisions about how to auto-type objects and how to generate intermediate polygons which enhance the environment. Currently, it has four major components:

- **Door detection and animation:** Models contain doors which have a standardized appearance (a 90° arc with an edge connecting to the center of the circle the arc lies on.) TOADS automatically types such objects as doors. Furthermore, when the 3D model is created, doors are exported as flat polygons (not arcs) which have two-states: open and closed. 3D modelers which support interactivity (e.g. VRML) can link mouse click events to transitions between the door states, creating an environment in which the user can open and close doors.

- **Window detection:** As with doors, windows are often drawn with a standardized appearance in blueprints. In the case of the MIT DXF models, windows are drawn as two parallel lines on the outside of the model. Windows are automatically assigned a transparent texture.

- **Layer to object type mappings:** An interface to automatically type objects based on their layer in the DXF file is also supported. Because DXF models often separate objects into layers based on object types, it's straightforward to map all objects within a particular layer into a comparable object type.

- **Automatic generation of ceiling and floor polygons:** Polygonal objects, rooms, and models need to be closed objects, with tops and bottoms. Drawing these tops and bottoms by hand is slow, especially in complex models. TOADS automatically generates tight top and bottom polygons to represent the floors and ceilings of rooms and models as well as the top and bottom polygons of closed moveable objects.

There are a number of potential extensions to this system; the most important is automatic room detection, a feature present in the BMG system which may be transferable to TOADS. Rooms provide an important way of logically dividing up models and applying defaults: just as building floors have walls with similar textures, rooms are even more likely to have walls with the same texture.

### 3.6 Defaults and Object Property Inheritance

Modern architecture has the fortunate property that, within a particular building, almost every wall, floor, ceiling, door, and window looks the same. There are some differences due to human occupation – pictures on walls, notes taped to bookshelves, etc.; but in the absence of personal touches, most architectural interiors don’t include a large variety of surfaces or textures. For this reason, being able to specify model-wide defaults – default settings for every floor or wall within a model – can greatly reduce the number of individual polygons which have to be textured. For this reason, TOADS sets up texture-defaults for many common types of objects as well as a height default for the entire model. Each object in the blueprint is given a type, such as window, door, or wall, which is used to generate its texture if a specific texture is not otherwise specified.
3.7 Intermediate 3D Files

The intermediate 3D format is generated from the DXF objects which comprise the 2D model. We choose to use an intermediate format to eliminate any knowledge of a specific 3D environment from the individual two-dimensional objects and from the main part of the TOADS system. As users need to support additional virtual environment formats, they can use provided TOADS classes to parse the intermediate file and generate the appropriate 3D file. The intermediate format is a concise text-based description of polygons and textures which is easy to generate and parse. A tool to generate VRML-based environments from the intermediate format is provided.

4. Implementation

TOADS is a sizeable piece of software engineering, currently about 20,000 lines of C++ code in length. The design of the software comprised a significant part of the thesis project, and so a significant portion of this document is dedicated a discussion of the program structure. A summary of the major modules is followed by a detailed discussion of each module, complete with algorithms, specifications for IO formats, and screen shots and interface descriptions of associated interface items. Limitations and bugs are also indicated – a complete listing of known bugs and planned enhancements is summarized in Appendix B. Though this approach is not the most convenient for someone interested in a simple user's manual, nor the most useful for someone interested solely in source code reference, it provides the most coherent layout for readers with a general interest in the software.

The TOADS Tool Suite consists of the main TOADS program plus a number of smaller tools which facilitate texture acquisition and VRML file creation.

The TOADS program is responsible for opening DXF files and allowing the user to interact with them by adding textures, rooms, and other objects. It interfaces to the other three tools, which are:

*TripodGrabber*, a tool designed to interface to the rack-mounted texture acquisition engine described in section 4.9.3. This program generates QuickTime movies which consist of the sequential frames generated as the camera moves along the wall. This tool is described in section 4.9.1.

*QuickTime Viewer*, a tool designed to take the movies generated by the TripodGrabber program and tile them into PICT files which represent the actual textures. See section 4.9.2.

*ToVRML*, which inputs TOADS intermediate 3D files and generates VRML environments, including openable doors and transparent windows. It is described in Section 4.12.2.

4.1 Main TOADS Program

Throughout this document, module dependency diagrams are used to show the structure of pieces of the program. These diagrams are used in MIT software engineering classes and provide a convenient way to visualize relationships between the major modules of the program; readers who are confused by them should refer to [10].

The top level module dependency diagram for TOADS is shown in Figure 1. These are abstract modules, none of which exist as actual C++ classes but which reflect the overall structure of the code.
The UI Controller is responsible for managing the basic user interface and calling into all of the other modules. It maintains all of the open documents, handles updates and user-interface events, and stores the top level data-structures for the program.

The UI Tools appear in the tool-panel of the program and are the principal interface through which users interact with DXF documents.

The DXFParser is responsible for reading and drawing DXF files. It is the largest and structurally most complex of the modules, and is used by nearly all of the other modules.

The Object Management module provides the user interface for manipulating object settings.
The Texture Acquisition module is responsible for providing the user interface for acquiring textures and managing the interface between the main TOADS program and the external hardware interface program. The Texture Management module stores information about standard and designer textures and includes interfaces for writing and reading textures to and from disk.

The 3D File Export module is responsible for generating intermediate 3D files from DXF files; it provides the user interface for exporting objects as well as a number of utility routines to create 3D textured environments.

The Preferences module maintains settings for each DXF file; these include default textures and ceiling heights, rules about converting PICT-based textures to other formats, and conversion factors between internal document units and meters.

4.2 DXF Parser

Figure 2 shows the main modules and dependencies of the DXF parser. It’s a large piece of code – nearly half of the program is devoted to parsing and managing DXF of one form or another. For reasons of simplicity, some modules have been omitted from this diagram; they consists mainly of utility functions used by many of the routines and a simple IO interface for reading and writing lines of a DXF file.

Much of the structure shown here derives immediately from the DXF format itself. A DXF-format drawing (represented by the high level DXFPicture class) first contains a DXFHeader, which specifies top-level settings such as the drawing scale, default orientation, and color map. Then, there are a number of DXFTables, which specify information about line styles, viewpoints, and layers; we only concern ourselves with DXFLayers (see 4.2.2, “Parser Limitations” below.) A DXFLayer is a named grouping of co-planar DXFEntity objects. Each DXFEntity represents one DXFObject, which may be a DXFLine, DXFPolyLine (polygon), DXFSolid (4-vertex filled object), DXFCircle, DXFArc, DXFComment, or DXFText class. The additional classes in the right hand side of the diagram (DXFLight and DXFDoor) are derived from one of these base classes and instantiated through a DXFComment (Lights – see Section 4.2.1, “DXF Format Extensions”) or generated dynamically from the picture (Doors – See Section 4.11).

Each line of a DXF file consists of an integer-tag which identifies the type of data which follows, and some arbitrary amount of following data. The boundaries between specific DXF objects are marked by a unique integer identifier, which makes it easy to tell when an object has been completely parsed. Parsing a DXF file consists of reading a tag and the following data, and then using the tag to modify the internal representation of the drawing and transition to the next expected input. If a particular tag is unexpected at a particular point, this is a parse error. In this manner, it’s possible to express the parsing of a DXF file as a finite-state machine, where each of the possible valid inputs from the current state are transitions to another state. An FSM-based parser is straightforward to write: standard compiler books like [11] provide relevant technical details. A distributed FSM is used for parsing DXF files in TOADS; the top-level DXFPicture object is instantiated with a pointer to a candidate-DXF file to be parsed. It’s Parse function is called, which reads the first tag; if the tag is expected (e.g. it identifies a header, table, or entity), then an appropriate object is instantiated and that new object’s Parse routine is called. If an unexpected field is encountered, an exception is thrown. This process is repeated, with classes calling Parse for objects they contain, until the entire file is read or an error is encountered.
4.2.1 DXF Format Extensions

One of the major benefits of using DXF in TOADS is that it provides a way for TOADS specific data to be saved without introducing a new file format. Because TOADS is not a DXF editor *per se* (although it does have some editing capabilities), users may want to use an editor in conjunction with TOADS tools. There are many free and commercial tools which can read and manipulate DXF, so obtaining an editor is not an issue. This leaves the problem of encoding TOADS specific data so that files don't appear to be garbled when these editors read them. The solution is the comment field which DXF provides.

TOADS embeds application specific information in the body of comments and then extracts that data as it reads the file. Since other DXF editors will ignore these comments, they can still view files augmented in this way. Well-behaved editors (such as AutoCad [6] ) leave the comments in place so that TOADS can still read them. TOADS comments are parsed exactly as any other object (by the Parse routine) and are encapsulated in a DXFComment object. They have the following format:

999
V*EKF, version, type, data
Where 999 denotes the start of a comment, VEDXF indicates the comments belongs to the TOADS system, version is the software revision that wrote out the file (to allow newer parsers to recognize and properly parse files which may have out-of-date comments), type is the class to which the comment belongs, and data is optional additional information for that type.

Comments are used for a variety of functions; Appendix A at the end of this document summarizes them by type. These comments store data structures which are described in more detail later in this document. Where possible, the appendix includes section-number references.

4.2.2 Parser Limitations
The TOADS DXF parser is not full featured. The DXF file-format is extremely rich, and writing a complete parser and renderer would require significantly more time than the few months of engineering available for a Master’s thesis. In practice, none of the blueprint files found so far have used any of the constructs TOADS doesn’t understand, although TOADS has been tested with these constructs to verify that it doesn’t break.

There are three major limitations: as mentioned above, some of the tables and header fields, such as line-patterns and viewpoints are simply ignored; blocks, which provide support for complex objects comprised of the basic DXF object types are not parsed and will not be drawn; 3D points, though parsed, simply have their third dimension omitted when drawn (so 3D DXF files will appear very flat.)

4.3 Main UI
The main TOADS UI consists of a number of menus and a floating tool palette, as shown in Figure 3. The tool palette provides a number of operations which can be performed on open documents; see Section 4.4, “Tools.” The program opens DXF files and places each into its own window. Users select tools and menu items to manipulate DXF files in preparation for VE generation. Menu options are summarized in the following table:
<table>
<thead>
<tr>
<th><strong>File</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open…</td>
<td>Open a DXF file in a new window.</td>
</tr>
<tr>
<td>Save</td>
<td>Write changes to the front-most window to disk.</td>
</tr>
<tr>
<td>Save As…</td>
<td>Write the front-most window to a new file.</td>
</tr>
<tr>
<td>Write 3D Data…</td>
<td>Write out an intermediate 3D data file for the front-most window. (4.12)</td>
</tr>
<tr>
<td>Close</td>
<td>Close the front-most window.</td>
</tr>
<tr>
<td>Page Setup…</td>
<td>Show the Page Setup dialog for the front-most window.</td>
</tr>
<tr>
<td>Print…</td>
<td>Print the front-most window.</td>
</tr>
<tr>
<td>Preferences…</td>
<td>Show the preferences dialog for the front-most window. (4.8)</td>
</tr>
<tr>
<td>Quit</td>
<td>Exit TOADS.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Edit</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo</td>
<td>Standard Edit option. Not supported in TOADS.</td>
</tr>
<tr>
<td>Cut</td>
<td>Remove the selected DXF objects and place them on the clipboard.</td>
</tr>
<tr>
<td>Copy</td>
<td>Copy the select DXF objects to the clipboard.</td>
</tr>
<tr>
<td>Paste</td>
<td>Paste DXF objects on the clipboard into the front-most DXF file.</td>
</tr>
<tr>
<td>Clear</td>
<td>Standard Edit option. Not supported in TOADS.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Layers</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Named Layers</td>
<td>Checkmarked layers are visible. Selecting a layer toggles its visibility.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Room</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Item(s) To Room…</td>
<td>Add selected DXF objects to the specified room.</td>
</tr>
<tr>
<td>Select Room</td>
<td>Select all DXF objects in specified room.</td>
</tr>
<tr>
<td>Room Settings…</td>
<td>Setup textures and ceiling-height for specified room.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>DXFObject</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Object Length…</td>
<td>Establish DXF unit-to-meters mapping by specifying the length, in meters, of the currently selected DXFLine object. (4.5)</td>
</tr>
<tr>
<td>Get Object Info…</td>
<td>Texture and height settings for the currently selected object. (4.5)</td>
</tr>
<tr>
<td>Collision Detect</td>
<td>Toggle collision detection for currently selected objects. (4.5)</td>
</tr>
<tr>
<td>Set Type</td>
<td>Set the object type (e.g. wall, door) for currently selected objects (4.5)</td>
</tr>
<tr>
<td>Grab Texture…</td>
<td>Acquire and apply a texture to currently selected objects. (4.9)</td>
</tr>
</tbody>
</table>

Where appropriate, references to the sections of this document which describe the functioning of particular menu options are indicated. The other menu-options behave as they should in any Macintosh application: opening, printing, and saving are common operations which bear no further description.
Each DXF window presents a view of a single DXF file. Documents are drawn much as they would appear in any DXF viewer, although there are some TOADS specific drawing features, limitations, and additional visual cues:

- Walls to which textures (either designed or otherwise) are indicated in blue. This is to allow users to quickly tell which portions of a document they have texture without examining each object. This feature will make documents which use a lot of blue confusing; fortunately, the blueprints provided by MIT use almost no blue.

- DXF files generally specify their colors by number; the exact meaning of these colors is up to the viewer, although AutoCad does specify default meanings. The TOADS class DXFColorMap is responsible for mapping color numbers to actual colors – it uses the standard AutoCad definition. Note that, for purposes of 3D VE generation, color has no significance.

- DXF documents specify coordinates in internal units. The DXF header field establishes the size and ratio of width to height via the Minimum Extents and Maximum Extents fields. These specify (x,y) coordinates for the upper-left and lower-right corners, respectively. TOADS parses these fields and sets up the size of the drawing to have the proper width-to-height ratio.

DXF files are usually separated into layers. These layers may correspond to any logical division of objects the file creator saw fit. TOADS properly parses layers and allows them to be turned on and off via the Layers menu – selecting a layer toggles its visibility. By hiding one or more layers, it may be easier to manipulate objects in the file. Invisible layers are not exported to the 3D virtual environment.

### 4.3.1 Main UI Implementation
The Main UI consists of two classes: TOADSMain and DXFWindow, which together handle almost all of the UI maintenance. TOADSMain is responsible for initializing the program, creating the menus and the tool window, and responding and forwarding UI events. It handles menu-selections which create and close windows, and forwards most other commands to the front-most DXFWindow class.

DXFWindow consists of a hodge-podge of routines which handle many different types of basic UI operations. It contains the DXF representation of the picture it is drawing, and maintains scroll-bars and buffers to allow fast redraws of the image. It manages all of the menu-commands directed at a specific DXF window, and also handles most of the mouse and selection tracking for the various editing tools.

These two files comprise a large amount of code, none of which is particularly complicated. Further detail as to the code structure would be tedious; curious readers should refer to the source code.

4.3.2 Exception Handling

TOADS incorporates a reasonably sophisticated exception handler which handles errors in the program and passes them up to the top-level UI so that informative dialogs can be shown. TOADS exceptions are used for propagating global, unrecoverable failures and not for message passing or flow control, as in some C++ programs [10]; therefore, any exception results in the closing of the front-most window and/or program termination. Figure 4 shows the basic exceptions which TOADS handles. Exceptions are handled by the TOADSMain and the DXFWindow class, which give different user feedback based on the type of exception.

![Module dependency diagram for exceptions in the TOADS program.](image)

DXFWindow handles ParseErrors, which result when an invalid, incomplete, or unintelligible DXF file is opened. These errors only occur when a file is opened; the user is prompted with an informative message, such as “The selected DXF file was invalid” and the existing DXFWindow structure is de-allocated.
TOADSMain handles errors elsewhere in the program, which generally result when a MacOS ToolBox call fails or runs out of memory. When these exceptions fire, it means that some caller was unable to proceed; therefore, the DXF file which generated them must be closed. If no DXF window is open, we assume the program must quit. Though this may not be a strictly valid interpretation of the status of the program (e.g., the error may only prevent some piece of the program from working, such as a dialog-box), gracefully closing the frontmost window and giving the user an error message is considered better than crashing.

4.3.3 Memory Management

One of the most difficult issues in designing large C++ programs today is memory management. Each document in the TOADS system contains thousands of objects, each of which is represented by its own DXFObject structure, which may contain pointers to textures, rooms, and other objects. Keeping track of these objects so as to minimize the required amount of storage is difficult. Knowing when objects are no longer in use and can safely be disposed of is still harder.

Limiting memory usage turned out to be not so difficult; the real issue has to do with making sure that DXFObjects (which use most of the memory in the system) don't grow too large; this is done by keeping NULL pointers instead of initialized, empty fields for rooms and textures in DXFObjects which aren't yet textured or affiliated with rooms.

The more difficult task was disposing of those DXFObjects at the appropriate times. The problem is that references to objects are kept in many locations; for instance, an object which appears in a drawing may belong to several rooms, be a member of the current selection, or be a vertex or line in a larger enclosing polygon. Because it's inefficient (for both storage and performance reasons) to keep separate copies of the object in each of these locations, a reference is stored instead. But then, since there are many pointers to the same object floating around, it's hard to tell when the object is no longer needed. If the user makes a new selection in the front window, the object won't be a part of the current selection anymore, but it still belongs to the main drawing and thus can't be disposed. The solution to this problem is reference counting. DXFObjects are stored in DXFObjectList structures, which have methods to add and remove objects from them. Each object has a counter as to the number of lists it belongs to; when the object is added to a list its counter is incremented. When an object is removed, its counter is decremented. When an object's counter reaches 0, no one is referring to it anymore and it's safe to deallocate it. This is a standard software engineering approach to memory management (in the absence of garbage collection) [10].

4.3.4 Performance

Part of the reason for selecting C++ to implement the TOADS system was that a sleek, high performance engine for viewing and navigating DXF files was desired. A fair amount of engineering went into providing a high speed, low-memory footprint system. It was deemed more important to provide a speedy interface than one which used fancy graphics or offered an immediate 3D preview. Several key optimizations were made:

- Caching: Drawings are cached to provide quick redraws when scrolling and navigating.
- Minimal draw routines and efficient coordinate transforms. Each object's Draw method is short, just a few lines, and doesn't rely on any information on disk (e.g. stored pictures). Screen-to-DXF transforms are efficient because scaling factors are precomputed so that each transformation takes only two multiplies.
As a brief comparison, the following table summarizes the performance of TOADS at opening, redrawing, and saving a DXF file versus several graphics packages on the Macintosh. Canvas (version 5.0.2) is a commerical drawing package not targeted specifically at CAD but with support for DXF. It is considered a very-powerful, versatile vector-based image editor. CADintosh (version 3.0.1) is a fully-featured shareware CAD program with native DXF support. It is much like a modernized Macintosh version of AutoCad (which hasn’t been released in several years), though it isn’t as quality a program as Canvas.

All tests were performed on a 233 Mhz G3 Macintosh with an ATI-RAGE graphics card, a SCSI II interface, and 72 Mb of RAM. Virtual memory was disabled. All applications were allocated 30Mb of application heap. The document was a 1.6 Mb DXF file with 13950 objects in it, most of them lines.

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Used</th>
<th>Open Document</th>
<th>Zoom</th>
<th>Duplicate All Objects</th>
<th>Time To Save</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOADS</td>
<td>2.2 Mb</td>
<td>7.7 s</td>
<td>1.9 s</td>
<td>9.4 s</td>
<td>12.21 s</td>
</tr>
<tr>
<td>Canvas</td>
<td>2.2 Mb</td>
<td>11.3 s</td>
<td>4.5 s</td>
<td>30.2 s</td>
<td>3.2 s</td>
</tr>
<tr>
<td>CADintosh</td>
<td>1.8 Mb</td>
<td>57 s</td>
<td>2.8 s</td>
<td>-</td>
<td>16.28 s</td>
</tr>
</tbody>
</table>

Canvas and CADintosh are both full featured drawing programs with considerably more editing features than TOADS. The above chart is not meant as a head-to-head competition between the programs but as an indication that the TOADS DXF rendering engine is quick and capable.

4.4 Tools

Figure 5 shows the Tool Palette. The following table summarizes each of the tools:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>➡ Arrow</td>
<td>Used to select and move objects; holding down the shift key allows multiple selections.</td>
</tr>
<tr>
<td>✱ Magnifying glass</td>
<td>Used to zoom into the document. Holding down the option key zooms out.</td>
</tr>
<tr>
<td>🖼 Selection Box</td>
<td>Used to select multiple objects.</td>
</tr>
<tr>
<td>🔍 Texture Grabber</td>
<td>Used to acquire textures (See section 4.10).</td>
</tr>
<tr>
<td>🐐 Junk (Movable Object) Definition</td>
<td>Used to define moveable polygonal objects (aka “junk”) in the model. (See Section 4.7).</td>
</tr>
<tr>
<td>🏥 Room Definition</td>
<td>Used to define rooms and room boundaries (See Section 4.8).</td>
</tr>
<tr>
<td>🌟 Light Placement</td>
<td>Used to create light sources. (See Section 4.4.2).</td>
</tr>
<tr>
<td>🌀 Path Definition</td>
<td>Used to define paths (aka habitrails, see Section 4.12)</td>
</tr>
<tr>
<td>⚪ Origin Placement</td>
<td>Used to define the starting point for the viewer in the 3D environment. (See section 4.4.3 and 4.13).</td>
</tr>
</tbody>
</table>
As with other drawing programs, tools are selected and then used to manipulate the drawing by clicking within it. The rest of this section details the tools which aren’t described as a part of another major section.

### 4.4.1 Basic Object Manipulation

The arrow tool allows rotation when manipulating PolyLine objects (closed-polygons). Because these objects are used primarily to represent temporary objects in the model, it was important to allow them to rotate as well as move: hold down the option-key and drag on a polygon to cause it to rotate.

The arrow, magnifying glass, and selection box work similarly to their counterparts in standard drawing programs. The arrow ( ) can be used to select objects, and also to drag objects to new locations on the screen. Multiple objects may be selected by holding down the shift-key.

The arrow tool allows rotation when manipulating PolyLine objects (closed-polygons). Because these objects are used primarily to represent temporary objects in the model, it was important to allow them to rotate as well as move; hold down the option-key and drag on a polygon to cause it to rotate.

The magnifying glass ( ) zooms in and out (if the option-key is held down) out of the drawing. Clicking on a point in the drawing zooms while keeping that point at the center of the window.

The selection-box tool ( ) selects multiple-objects. Holding down the shift key allows objects to be added to the current selection. Click and hold in the drawing, then drag to define the selection-area. The selection box only selects objects whose upper-left and lower-right corners are completely enclosed by the dragged-area.

Objects can also added or removed via the Cut, Copy, and Paste commands in the Edit Menu. Pasted objects are replaced into the exact location where they were copied or cut from; the new objects are automatically selected and can be moved to a new location if desired. Note that the Edit commands allow objects to be moved from one drawing to another, which may be useful if complex polygonal objects which are needed in many drawings have been defined.

Hitting the delete key while objects are selected removes them from the drawing.

### 4.4.2 Lights

The Light tool ( ) creates a new light source at a specified point in the drawing. It is indicated by a circle at the specified point. The current implementation of lights is limited – TOADS only supports diffuse lights with a fixed radiance. There is no user-interface which allows more complex or interesting light sources to be defined, even though VRML and most other 3D viewers support such constructs. Time permitting, this feature will be implemented at a later date. Notice that the Light comment and intermediate file format do provide storage for additional lighting information, such as light-type and radiance.
4.4.3 Origin Placement

The origin tool ( ) allows specification of the start-point of the viewer in the 3D environment. This feature is important because environments are large and closed: users could be initially placed in an area which was closed off from the rest of the model, which would be unacceptable. Currently, only and (x,y) orientation is specified; it would be useful if a viewing angle could also be provided.

In addition to adding a viewing angle, future enhancements also include allowing the specification of multiple “Points of Interest” which could be jumped to quickly in the environment; VRML supports this feature, but TOADS currently contains no provisions for it (although the intermediate 3D file format does.)

4.5 Object Settings

The DXFOBJECT menu includes several menu-items which set basic object parameters.

4.5.1 Object Length Dialog

The “Set Object Length...” item allows a correspondence to be established between DXF coordinates (as arbitrarily defined in the DXF file) and real world distances (as measured in the real building). Initially 1 DXF unit is assumed to be equal to 1 meter in length. This menu item is enabled when a DXFLINE object is selected in the front most window (it’s hard to define "length" for non-line objects). Figure 6 shows the dialog – the length of the line is specified in meters:

![Image of Object Length Dialog]

Figure 6: The Object Length Dialog allows a mapping to be established between DXF and real-world coordinate spaces.

This setting is important anytime real-world values are specified: for instance, the size of the DXF grabber rectangle and the heights of ceilings and objects are specified in meters, but the DXF file stores other coordinates in internal units. When the 3D model is built, all coordinates are converted to meters.

The value entered into the Length field is used to initialize the metersPerDXFUnit field of the DXFPrefs record stored in the DXFPicture for the front-most window. The value of this field is set to be length (in meters) / length (in DXF units), where the number of meters comes from this dialog and the number of DXF units is computed from the vertices of the currently selected DXFLINE object.

4.5.2 Object Info Dialog

Selecting “Get Object Info...” brings up the Object Info Dialog for the first-selected item, as shown in Figure 7. The Object Info Dialog allows a PICT file-based texture to be associated with an object, allows an object to be given a specific height, and allows collision-detection to be turned on or off for the object.
Object names are included to simplify data management and serve no purpose except as an organizational tool for the user. The “Object Height” field specifies the height of the object – if it is blank, then the object is assumed to have the same height as the ceiling of the room or model it is contained within. The “Collision Detection” check-box determines if the object can be collided with – that is, in the resulting VE, will the user be able to pass through the object or be blocked by it. The “Change” button allows a new texture to be specified for the object; the “Design” button allows a new texture which consists of several tiles of simple texture to be designed. See section 4.9 for a discussion of these dialogs.

![Object Info Dialog](image)

**Figure 7:** The Object Info Dialog allows modification of the name, texture, height, and collision detection flag for a selected object.

### 4.5.3 Object Types

The “Set Object Type” item within the DXFObject menu allows objects to be assigned to a general class of objects. Choosing an item from this submenu changes all of the objects in the current selection to the specified type. Object types are used to associate default textures with objects – for example, to make windows appear transparent and walls appear to be brick in the absence of individual settings for the object. The textures applied to each object type are set through the preferences dialog (see Section 4.8).

### 4.6 Defining Polygonal Objects

TOADS includes the Junk Tool ($) to allow movable, transitory objects to be created within environments. The term “Junk” is meant to suggest that these objects are small, impermanent features of the environment. The intention of this tool is that it will be used to represent objects such as boxes, chairs, and other small objects which change location frequently.

Defining polygonal objects is accomplished by selecting the tool and then clicking out the points which form the vertices of the polygon. Double-clicking stops the definition process. Once a polygon has been created, it can be moved, rotated (by option-clicking with the Arrow tool), and textured, just like other objects.

The Object Info Dialog differs slightly for polygonal objects, as shown in Figure 8; in addition to the settings discussed in section 4.5 above, polygons also have a top-texture and a base-height associated with them. The top-texture is used to specify the texture which is drawn on the lid of the polygon when it is exported – if no texture is specified, the regular object texture is used instead. The base-height allows polygon objects which are off the ground to be defined – this is important because models often contain objects which sit on other objects (such as books on a bookshelf).
Textures are applied to PolyLines in one of two ways: either, a single “Side Texture” is wrapped around all the faces of the polygon, or a separate texture is applied to each face of the PolyLine. The former mode is used if the “Separate Textures For Faces” checkbox is unchecked, the latter if it is checked. When the checkbox is checked, sides of PolyLines can be clicked on and have textures applied to the via the Object Info dialog (Figure 7, above.)

An example of how a texture is wrapped around a polygon when the “Separate Textures” checkbox is unchecked is shown below:

4.7 Rooms
Rooms are one of the key organizational features of TOADS. Users can define polygons which are the boundaries of rooms. Objects can be added and removed from the room as needed. The most important function of this grouping is divorce the user from the single polygon – for most of the detail within models,
once objects have been grouped into rooms, the user will never have to select and manipulate individual lines or objects and can instead simply specify the appearance of the room. Rooms have a number of settings associated with them: most importantly, ceiling height and textures for floors, walls, and ceilings.

The room tool ( ) works much like the Junk tool – the polygonal border of rooms is clicked out in the main window. Once a room’s border has been defined, the Room Settings dialog, as shown in Figure 10 appears. This dialog can be used to specify a name, color, ceiling height, as well as wall, ceiling, and floor textures for a room. Note that, although textures are specified and stored for ceilings and floors, ceiling and floor polygons are not generated for each room in the current TOADS implementation.

Figure 10: The Room Settings Dialog allows the user to specify defaults for objects in rooms.

The Room menu allows options to be changed and objects to be added to rooms. The “Add Object(s)” hierarchical menu adds currently selected objects to specified room. The “Select Objects” hierarchical menu adds items in the selected room to the current selection. The “Room Settings...” shows Figure 9 for the specified room.

Objects inherit height and texture information from their parent rooms: when an object doesn’t specify it’s own height or texture, it will use the height and texture of the first room it belongs to. Object may belong to several rooms – though this seems unintuitive, it’s possible for a wall to form the boundary between two or more rooms.

4.8 Preferences

The “Preferences...” option in the file menu is used to specify document-wide settings for the current environment, including default textures and ceiling heights as well as texture-export settings. Figure 11 shows the dialog.

The default texture settings specify textures to be applied to different object-types within the model (see Section 4.5.3). These textures are used only when objects don’t specify their own texture type and don’t belong to rooms which specify textures. The “Change...” button brings up the Texture Settings dialog (See Section 4.9) for the current object-type.
The “Default Ceiling Height” field specifies the height of the ceiling of the model in meters. When an object belongs to a room with a height or specifies its own height, the default height is not used.

The DXF Unit mapping is used to set up the scaling factor between DXF units and meters; generally, it’s easier to use the “Set Object Length...” option in the DXFObject menu (see Section 4.5.1).

The “Grabber Size” option specifies the dimensions of the texture-acquisition device. This is important to determine how many bands of texture are applied to each polygon in the model. See section 4.9.2.

The “Auto-convert textures” option enables conversion of PICT based texture files to other file formats, such as JPEG. When this option is enabled during 3D file export (see section 4.12), QuickTime will be invoked to write out a new version of each texture file. The new files will be created with the same name as the original with the addition of an appropriate file-type suffix. If the “Save In Export Directory” box is checked, the new files will be saved in the same directory as the exported 3D environment; otherwise, they will be saved in the same directory as the original texture file. By automatically converting, renaming, and moving textures, the user is removed from the tedious process of keeping track of the location, name, and type of each texture file and manually converting them using an image editing program.

4.9 Texture Application and Acquisition

The Texture dialog, accessed through the “Change” button of the Object Info and Preferences dialog allows a Macintosh PICT file to be selected as a texture for the object or object-type currently being edited. Figure 12 shows the dialog.

The “Select File...” option brings up a Standard File Dialog through which a PICT file to be used as a texture can be selected. If the “Mac OS Easy Open” and “QuickTime” extensions are installed, other picture formats, such as GIF and JPG may also be selected.

The “Transparency” option sets the degree to which light is allowed to pass through the object. A transparency of 0 indicates the object is opaque, 100 completely transparent.
The "Tile Horizontally" and "Tile Vertically" check boxes indicate whether the texture is repeated over the surface of the object or stretched to cover the object. A tiled texture is often used for floors or ceilings which consist of many identical panels – the number of tiles specifies the number of times the texture is repeated. Figure 13 shows an example.

![Texture Settings](image)

**Figure 12:** The Texture Dialog allows a Macintosh PICT file to be selected as a texture for an object. It allows texture and tiling options to be specified.

![Figure 13](image)

**Figure 13:** Three views of the same floor, with (left-to-right) a texture not repeated, repeated 10 times vertically and horizontally, and repeated 100 times horizontally and vertically.

The "Openable" check box is only enabled for door objects. In the exported VE, doors can be clicked on to cause them to rotate 90° along the axis of the door arc in the DXF file.

The Texture Design Dialog, accessed through the “Design” button of the Object Info Dialog, is used to create more complex textures which consist of several standard texture panels (as built in the Texture Dialog) combined to form one larger texture. It’s most useful for doors and windows which may have central transparent or moving regions surrounded by opaque, immobile borders. Transparent windows can be made to appear half-way up a wall, and doors lips can likewise be created. The dialog consists of a pane with a background texture on top of which other rectangular texture regions are layered. These texture regions may be transparent or openable, which will show through the background texture. The basic dialog is shown in Figure 14.
Figure 14: The Texture Design dialog allows complex textures to be designed for doors or windows.

The size of the design rectangle, in meters, is shown on its sides. These sizes are determined by the length (from the DXF file) and height (from the Object Info Dialog) of the object.

The “New Rect” button allows a new rectangle to be specified. Clicking on it and then dragging a rectangle in the design area specifies the size and location of a new texture panel. Once a panel’s area has been defined, the Texture dialog appears so that the appearance of the texture can be specified. Texture panels can be selected by clicking on them, dragged to new locations, and resized using the resize handles that appear when the panel is selected. Double-clicking on a texture panel brings up the Texture dialog, so that texture settings can be changed.¹

The “Background” button specifies the background of the texture file. This is the texture which is drawn behind all of the texture panels.

The “Down” and “Up” buttons allow texture panels to be moved on top of or behind each other. Clicking on one of these buttons causes the front-most panel to move up or down in the stack of textures.

It may be desirable to save and reuse designed textures in several objects, particularly because there are likely to be many doors and windows with the same appearance in a single building. The “Save” and “Load” buttons allow a designed texture to be written to or loaded from disk. They bring up Standard File dialogs which enable this. Designed texture are stored in the model as references to a file on disk, so when the dialog is closed, via the “OK” or “Cancel” button, a dialog asking to save the texture will be brought up.

4.9.1 Designer Texture Implementation

It’s important to understand exactly how a designed texture works in an exported environment. Texture panels are subtracted from the background texture; thus, transparent or openable objects will not show the background texture and the rest of the environment will be visible through them. Figure 15 shows an example of a designed transparent window in a virtual environment.

¹ Currently, the Texture Design dialog does not reflect transparency or tiling settings, although they are properly supported in the exported VE.
The algorithm for computing the rectangles which compose the background after the top texture panels have been subtracted works as follows:

\[
\text{compute\_background} \left( \text{br}, \ panels, n \right) \ ; \ \text{compute \ hr} - \ \text{all \ panels \ in \ (0..n-1)}
\]

\[
\text{let \ brs} = \text{difference\_rects} \left( \text{br}, \ panels[0] \right)
\]

\[
\text{for \ all \ panels \ p \ from \ 1 \ to \ n-1}
\]

\[
\text{let \ new\_brs} = \text{empty \ rectangle \ list}
\]

\[
\text{for \ all \ rectangles \ r \ in \ brs}
\]

\[
\text{add \ difference\_rects} \left( r, \ p \right) \ \text{to \ new\_brs}
\]

\[
\text{let \ bbrs} = \text{new\_brs}
\]

\[
\text{return \ bbrs}
\]

\[
\text{difference\_rects} \left( r1, \ r2 \right) \ ; \ \text{compute \ r1 - r2}
\]

\[
\text{let \ ri} = \text{intersection \ of \ r1 \ and \ r2}
\]

\[
\text{if \ ri \ is \ empty, \ return \ r1}
\]

\[
\text{let \ diff} = \text{empty \ rectangle \ list}
\]

\[
\text{if \ (ri.r < r1.r) ; \ overlap \ on \ right \ edge}
\]

\[
\text{add \ rect \ with} \ t = ri.t, \ l = ri.l, \ b = r1.b, \ r = ri.r \ \text{to} \ \text{diff}
\]

\[
\text{if \ (ri.l > r1.l) ; \ overlap \ on \ left \ edge}
\]

\[
\text{add \ rect \ with} \ t = ri.t, \ l = ri.l, \ b = r1.b, \ r = ri.l \ \text{to} \ \text{diff}
\]

\[
\text{if \ (ri.b < r1.b) ; \ overlap \ on \ bottom \ edge}
\]

\[
\text{add \ rect \ with} \ t = ri.t, \ l = ri.l, \ b = r1.b, \ r = ri.x \ \text{to} \ \text{diff}
\]

\[
\text{if \ (ri.t > r1.t) ; \ overlap \ on \ top \ edge}
\]

\[
\text{add \ rect \ with} \ t = ri.t, \ l = ri.l, \ b = ri.t, \ r = ri.l \ \text{to} \ \text{diff}
\]

\[
\text{return \ diff}
\]

The difference rectangle algorithm is simple – it computes the rectangles which represent r1 after r2 has been removed from it; a basic example is shown in Figure 16. Figure 17 shows the background computation for a simple 2 rectangle designed texture.

---

**Figure 15:** A designed texture showing a transparent window in a wall.
Example of difference_rects \((r_1, r_2)\)

\[
\begin{array}{c}
\text{\(r_1\) - \(r_2\) yields 4 new rectangles:} \\
\text{\(r_{1,1}, r_{1,2}, r_{2,1}, r_{2,2}\)} \\
\text{\(r_{1,1}, r_{1,2}, r_{2,1}, r_{2,2}\)} \\
\text{\(r_{2,1}, r_{2,2}, r_{1,1}, r_{1,2}\)} \\
\text{\(r_{2,1}, r_{2,2}, r_{1,1}, r_{1,2}\)} \\
\end{array}
\]

Figure 16: The difference_rects algorithm computes the intersection of two rectangles.

\[
\begin{array}{c}
\text{compute_background}(bg, \{p_0, p_1\}, \text{true}) \\
\text{let \(f_{\text{new}} = \text{difference_rects}(bg, \text{panels}[0])\)} \\
\text{for all panels \(p\) from 1 to \(n\)} \\
\text{let \(f_{\text{new}} = \text{empty rectangle list}\)} \\
\text{for all rectangles \(r\) in \(f_{\text{new}}\)} \\
\text{add \(\text{difference_rects}(r, p)\) to \(f_{\text{new}}\)} \\
\text{let \(f_{\text{new}} = \text{new \_brs}\)} \\
\text{return \(f_{\text{new}}\)} \\
\end{array}
\]

let \(f_{\text{new}} = \text{difference_rects}(bg, \text{panels}[0])\)

Figure 17: The compute_background algorithm running on a simple 2 panel example.
4.9.2 Texture Acquisition

Textures are applied and acquired using the grabber tool (\textsuperscript{\textcopyright}) and the grabber dialog, which is accessed by selecting “Grab Texture…” from the DXFObject menu. The grabber tool is used to indicate the position of the texture acquisition hardware in the model and to automatically associate acquired textures with the model. The grabber appears as a square polygon in the model; it can be dragged about, and will snap to the nearest line or PolyLine it is brought near. Figure 18 shows the grabber rectangle being dragged along a wall. Its size is set via the “Grabber Size” field of the preferences dialog (see Section 4.8). The grabber rectangle can be moved one length up or down the wall it’s currently snapped to by using the left and right arrow keys. Once the grabber has been positioned near the wall for which a texture is to be acquired, select “Grab Texture” to bring up the texture grabber dialog (Figure 19). This dialog implements the TOADS side interface to the texture acquisition and editing programs.

![Figure 18](image)

**Figure 18:** The grabber rectangle, circled. Notice it is being dragged along the wall.

![Figure 19](image)

**Figure 19:** The Texture Acquisition Dialog

The left scrolling list show QuickTime movies which have been scanned using the Grabber Program (see Section 4.9.3 below) or added from an external source. Click the “Add” button to bring up a standard file dialog which can be used to select any QuickTime movie. The “Scan” button launches the Grabber
program (if it isn’t already running) and commands it to acquire a texture-movie. In either case, a new item is added to the left scrolling list.

The right scrolling list contains PICT files which represent swaths of texture. They are obtained by tiling frames of captured texture movies into a single file. Click on a movie in the left list and press the “Copy >>” button to launch the QuickTime Viewer application (See section 4.9.4 below) and open the selected movie. QuickTime Viewer processes the movie to create texture swaths, which are sent back to the dialog and added to the “Processed Textures” list. Use the “Up” and “Down” buttons to arrange pictures so that textures corresponding to adjacent wall segments are next to each other in the list. Once textures are properly arranged, click the “Make” button to concatenate them into a single texture file and automatically apply that texture to the object the grabber is currently adjacent to.

4.9.2.1 Implementation

The texture acquisition portion of the TOADS code is largely a user-interface implementation. One important software engineering issue, however, is the implementation of the inter-application interface which allows external applications to provide acquired textures and movies to the TOADS application.

There are two applications which TOADS interacts with: the first is the Grabber application, which encapsulates the interface to the texture acquisition hardware and returns QuickTime movies which, after some post-processing, can be converted into textures. The second is the QuickTime Viewer, which takes in the QuickTime movies, tiles them, and produces useable textures as PICT files.

Since the current texture acquisition technique (a rack-mounted camera) is probably not the ideal texture-capture device, users of the TOADS system may wish to deploy more sophisticated texture acquisition hardware in the future. By using external applications to generate and tile movies, TOADS makes this possible. The interface between these programs and TOADS is implemented via AppleEvents, which are Apple’s native inter-application communication device.

The format of these events is summarized here, and the reader is referred to [20] or the source code for a further information as to how to implement AppleEvents. Most of the constants referred to here are defined in the TOADS header file GrabberAEs.h. The TOADS application sends two events, one to the Grabber and one to the Viewer. The Grabber event is of class kGrabberEventClass with ID kGrabMovieAE. It has one parameter, which identifies the ProcessSerialNumber (unique ID assigned to the process by the MacOS) of the TOADS application. This parameter is of keyword kSenderPSN with type typeProcessSerialNumber. The Viewer event is also of class kGrabberEventClass, but with ID kEditMovieAE. It has two parameters, the first of which is the same as the serial number parameter of the kGrabMovieAE event. The second is a reference to a movie file to be edited; it has a kFSSpecKey keyword and is of type typeFSS. The contents of the parameter are an FSSpec to the movie file.

The Grabber application is sent a grab movie event when the user clicks the “Scan” button. The user will have the texture acquisition device in place and ready to acquire, so the Grabber should begin scanning upon receipt. It is made the front-most application before receiving the event, so it can safely present UI items and interact with the user. After scanning a texture-movie, it should send an event of class kGrabberEvent with ID kReturnMovieAE back to the TOADS application (the PSN of TOADS can be determined by extracting the kSenderPSN parameter from the kGrabMovieAE the grabber received.) The
kReturnMovieAE event should contain one parameter, with keyword kFSSpecKey and type typeFSS which identifies the movie on disk.

The Viewer application receives an edit movie event when the user clicks the “>>” button. The viewer should open the movie and allow the user to combine its frames in some way to produce a texture band. Once the user saves the texture, it should send an event of class kGrabberEvent with ID kReturnPictAE back to the TOADS application. This event has one parameter, with keyword kFSSpecKey and type typeFSS which identifies the picture on disk.

TOADS locates the Grabber application by searching for the first application on disk with creator code ‘Grbr’ and the QuickTime Viewer application by searching for creator code ‘Edtr’. These programs are automatically launched when the TOADS program runs. Users implementing their own texture-acquisition programs should make sure their programs have the appropriate creator codes and that no other applications with those creator codes are on disk.

This interface should be sufficient to allow external texture acquisition hardware to be developed and easily plugged into the TOADS application.

4.9.3 TripodGrabber

The TripodGrabber program is a texture-grabbing application designed for use with a rack-mounted texture acquisition device (see section 4.9.5, “Texture Rack”). This rack consists of a serial interface and a small CCD camera; the TripodGrabber application controls the movement of the camera along the rack via the serial interface and uses a 3D video capture card attached to a Macintosh computer to grab frames from the video camera.

The application includes a number of settings which allow the user to specify how fast and far the camera moves, how big the captured video area is, and how the captured video data should be formatted when written out to disk. When the program is launched, it automatically searches for attached video-input devices and brings up a window showing live-video from any device found. If no devices are attached, the program will quit. Any QuickTime compatible video-capture hardware can be used; for the purposes of our system, an iRez Zoomed Video card was used with a Macintosh PowerBook G3 Series computer.

4.9.3.1 Macintosh-to-Texture Hardware Settings

Before beginning texture acquisition, settings for the Macintosh-to-Texture Hardware serial interface need to be specified. These are specified through the “Settings...” item in the Robot menu; Figure 20 shows the dialog which appears.
The “Steps per picture” field specifies how far the motor should move between each frame that is captured. The size of a step is dependent on the connected stepper motor; in the case of the current hardware, the motor has 96 steps per inch. The half-steps option decreases the size of each step by half; selecting this option will slow the motor down, but may make its motion smoother.

The “Delay before capture” field specifies how long the application should wait (in 60ths of a second) before capturing a frame but after moving the motor. This option allows some time for any vibrations in the rack to dissipate.

The serial-port menu specifies which serial port the texture-hardware is connected to.

### 4.9.3.2 Manual Camera Control

Now that these settings have been properly made, the motor can be moved. Pressing the left or right arrow keys causes the motor to take one step to the left or right, respectively. Pressing the up-arrow key moves the camera to the far left edge of the rack, and pressing the down-arrow key moves it to the far right edge.

The “Control Robot” item in the Robot menu moves the camera from left-to-right according to the step size settings in the Settings dialog. This allows a preview of the action of the camera before actual acquisition begins. The camera will stop when it reaches the right edge, or when the “Stop Grabber” item is selected from the Grabber menu.

### 4.9.3.3 Specifying Video Settings

Once the motor moves properly, video-input settings need to be specified. These are primarily established through the “Camera Settings..” option of the Video menu. The dialog consists of three panes, as shown in Figure 21.
The Source pane controls the input-hardware currently being used. If the default video capture device is different from the desired device, new hardware can be selected from the “Digitizer” menu. Some digitizers have several ports (such as S-Video and Composite) and several video formats (such as NTSC and PAL) which can also be specified.

The Image pane controls the color balance, brightness, and contrast of the image. Fiddle with the sliders to get an optimal image; note that some texture-acquisition hardware includes automatic gain control which attempts to compensate for lighting variations within a scene; in general such features should be disabled because lighting is an important part of the appearance of textures and some regions of buildings are genuinely darker than others. Also, adjust these settings to be optimal within the average lighting conditions of a building (e.g. indoors, under normal fluorescent lights without lots of sunlight from windows) since they should remain constant for all textures within a model.

The Compression pane specifies the compression algorithm to be used when capturing textures. Texture-movies are captured into memory, and will ultimately become uncompressed PICT files when processed with the QuickTime viewer, but can take up large amounts of space both in RAM and on disk before they are processed. Unfortunately, compression can also dramatically affect the amount of time required to capture each frame. The following table summarizes a few of the options:

<table>
<thead>
<tr>
<th>Compressor</th>
<th>Space Per Frame</th>
<th>Time To Compress</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>150 kb</td>
<td>-</td>
<td>Excellent</td>
</tr>
<tr>
<td>Motion JPEG A</td>
<td>~50 kb</td>
<td>Short (~.1s per frame)</td>
<td>Good</td>
</tr>
<tr>
<td>Cinepak</td>
<td>~5kb</td>
<td>&gt; 1s per frame</td>
<td>Fair</td>
</tr>
</tbody>
</table>

Cinepak is only useful in situations where memory or disk space is at a premium. Motion JPEG A offers a good trade-off between space, time, and quality, although on machines with an abundance of disk space and memory, using no compression is somewhat faster and higher-quality. Remember that the entire movie must fit into RAM when a texture is being acquired, and that a texture can require as many as 200 frames (30MB with no compression).

Once these options have been properly set, it may be desirable to specifying a clipping rectangle which limits the size of the capture video frame. Since most of each video frame will be thrown out when the QuickTime Viewer is used to tile the movies, there’s no reason to save the extra frame data to disk. The “Clip Settings...” item brings up a dialog box which allows the clipping rectangle (visible region) to be specified.
4.9.3.4 Recording

Once the clip rectangle, video, and grabber settings have been specified, recording can begin. Select “Start Recording...” from the File menu. A standard file dialog is brought up, asking where the new movie should be saved. During the recording process, the video window is disabled to maximize the frame rate. Note also that no menu items can be selected; recording will end when the grabber reaches the far edge of the rack, or when Command-E is pressed.

After a movie has been captured, if TripodGrabber was opened via the main TOADS program, control will return to TOADS; the new movie will be added to the “Captured Movies” list of the Texture Acquisition dialog (see section 4.9.2).

4.9.4 Quicktime Viewer

The QuickTime viewer program opens movies produced by the TripodGrabber and tiles their frames to form a single texture image. These texture images are exported as Macintosh PICT files, which can be used as textures in the TOADS program.

Opening a movie brings up a numbered list of frames; double clicking on a frame brings up a window showing that frame and some information about it, such as its number, the time it was recorded, the position of the grabber when it was recorded, and whether or not a sound-file is associated with that frame. These two windows are shown in Figure 22. Note that the QuickTime Viewer program will only open movies created by the TripodGrabber—it requires special information attached to each frame to work properly.

![Figure 22: The frame list window and the frame info window.](image)

4.9.4.1 Creating a Panorama

To create a panorama, select the frames which are to be included from the frame-list window and select “Build Panorama...” from the File menu. A new window will appear, and the frames selected in the movie list will be drawn into it from left to right. Figure 23 shows an example of what such a window may look like—oftentimes the frames are not tiled properly by default.

---

2 Sound files are not fully supported in the current TOADS implementation, so are not discussed here. See section 4.13 for a discussion of this and other future extensions to the system.
Note: Panorama windows may draw slowly, depending on the speed of the computer and the number of frames in the movie being viewed. At any time, redraw can be stopped by pressing the Command-Period keys. This is particularly useful when it's clear the panorama settings are far from producing a properly-tiled image, as in Figure 23 below.

First, the program needs to have the orientation of the movie set properly. As in the case of the hardware used here (see section 4.9.5), the camera may be rotated 90 degrees to get the larger dimension of the frame as the height - our capture card produces an image that is 320 pixels wide and 240 high; by rotating the camera 90 degrees it becomes 320 pixels high. The QuickTime viewer can deal with this properly - the "Rotate Clockwise" and "Rotate Counterclockwise" options in the Panorama menu specify which direction the camera has been rotated.

Once the frame orientation is set-up properly, it's necessary to establish the number of pixels in each frame which equal a single movement of the camera. The "Compute Frame Size..." option from the Panorama menu provides the most intuitive way to do this; by specifying the field of view of the camera, the distance of the camera from the wall, and the number of inches in each step of the camera, QuickTime viewer can automatically calibrate the frame size to the proper width. Figure 24 shows this dialog.
The number of pixels from each frame is computed by the program. The central region of each frame is used; this is to minimize distortion which often occurs with low-cost CCD cameras at the extreme edges of the visible region. For the scans used in our program only the central 3 pixels are used. Figure 25 shows the significance of each of the variables. The distance to wall can be computed via direct measurement. The field of view should be available from the lens manufacturer. The camera movement is the product of the step size (given by stepper-motor manufacturers) and the number of steps taken between each image.

Note that the field of view needs to be adjusted according to the following formula if a clipping rectangle was specified in the TripodGrabber program:

\[
realfov = \frac{fov \cdot framewidth}{clipwidth}
\]

Where \( framewidth \) is the width of the input frame from the video capture card and \( clipwidth \) is the width of the clipping rectangle.

The following derivation shows how the number of pixels per frame is computed:

\[
\tan\left(\frac{\theta}{2}\right) = \frac{wallWidth}{2} \cdot \frac{1}{x}
\]

\[
wallWidth = 2 \tan\left(\frac{\theta}{2}\right) \cdot x
\]

\[
pixelsPerFoot = clipWidth / wallWidth
\]

\[
pixelsPerFrame = \frac{\Delta}{12} \cdot pixelsPerFoot
\]
Figure 25: The $x$, $\delta$, and $\Delta$ parameters and their physical significance.

Once the frame size has been properly set, an image like the one in Figure 26 should appear. These images are high quality, good resolution images that don’t show the artifacts between the edges of each frame and don’t suffer from fish-eye style distortion.

Figure 26: A properly tiled panorama.

Once a panorama has been properly set up, it can be written out to a PICT file via the “Save To PICT...” option in the Panorama menu. The user is presented with a standard file dialog, and the PICT is written to the specified location. If the QuickTime Viewer was invoked via the TOADS application, saving a PICT to disk brings the TOADS application to the front and adds the newly saved PICT to the Processed Textures scrolling list.

4.9.4.2 Image Decompression

Due to the low quality of the CCD camera which is used on the current texture rack (see Section 4.9.5, below), the images it captures are highly distorted. The fisheye effect of the lens, which causes severe bending at the edges of the image, can be avoided by using just the central row of pixels from each frame, which aren’t bent at all. Unfortunately, these pixels are compressed vertically, so that the top and bottom
of each frame appears very small relative to its real-world appearance. Furthermore, CCD cameras introduce noise in the form of random bright pixels in dark backgrounds. The TripodGrabber application includes algorithms to correct both of these problems.

Fixing horizontal compression requires measuring the amount of compression. This was done by capturing an image of a vertical pole with markers every twenty inches along its length. By measuring the number of pixels between each band, it's possible to determine the amount of compression. Figure 27 shows the captured image:

![Figure 27: The vertical-compression calibration image. Dark lines indicate the placement of the regularly spaced markers.](image)

By finding the offset in pixels for each marker from the center marker, it was possible to determine that the compression is linear (See Figure 28 below) with a slope of about 56 pixels per 24" (the inter-marker distance). An algorithm for reconstructing the uncompressed image is now straightforward:

1) Determine the new image height. This is based on a desired band size (which we arbitrarily choose to be distance between the center and the first markers – 160 pixels). Then, multiply the number of marker-delimited bands in the image by this height. The image height is approximately 8 feet, and each band is 1.75 feet, so there are 4.7 bands and the new image height will be about 750 pixels.

2) Begin at the center of the new image and walk up (or down) and compute the pixel-row from the compressed image to use for this row in the uncompressed image. First, compute the number of fractional markers this pixel is from the center, assuming an inter-marker distance of 160 pixels and i as the current row:

\[
marker\_no = \frac{i}{160}
\]

Thus, the first pixel will be at marker 0, the 160th at marker 1, and so on. Then, the inter-pixel distance (ipd) for this band in the compressed image can be computed.

\[
ipd = \frac{214 - 56(marker\_no)}{214}
\]

This formula is drawn directly from the linear equation shown in the graph below. The ipd for the first pixel is 1, .74 for the 160th, and so on. The ipd tells how far to move in the compressed image given a movement of one pixel in the uncompressed image. The compressed pixel-row for the ith pixel is thus:
\[
    r_{ow_i} = r_{ow_{i-1}} + ipd \\
    r_{ow_0} = 0
\]

Note that this is a floating point number; the closest integral number is used to determine the actual pixel-row to use.

This algorithm produces satisfactory decompression. Figure 29 shows a door before and after decompression; notice that the compressed door’s handle appears very far down the image, but that the decompressed door looks much better.

Decompression is done via the “Save Decompressed PICT...” item in the Panorama menu. There is no interface – the image is decompressed according to the parameters shown above and written to the specified PICT file.
Figure 29: A door, compressed and uncompressed. Notice that the door handle and grating appear much more realistic in the uncompressed version.

4.9.4.3 Noise Correction

TOADS employs a very simple noise correction algorithm designed to eliminate bright spots in dark areas (which are the most common type of noise experienced with CCD cameras.) This approach is based on the assumption that captured frames in movies have some overlap (that the frame area used for each tile in the image isn’t the entire frame) and that the exact pixel offset between frames is known. Thus, corresponding areas in adjacent frames can be compared and the darkest pixels from each frame can be selected. This option is enabled via the “Noise Correction” item in the Panorama menu; the current frame is redrawn as soon as it is toggled on (or off). Figure 30 shows an image before and after noise correction; notice the white specs are no longer present (decompression has not been performed on these images).

Figure 30: An image before and after noise correction. Notice that the white dots across the top are gone, and that some irregularities in the door frame have been smoothed out.

4.9.5 Texture Rack

The Texture Rack is the hardware responsible for acquiring textures. It consists of low-resolution CCD camera mounted on a motorized track. The track is controlled by a stepper-motor, which allows very high-accuracy control of the position of the motor (and hence the position of the camera). Each step of the motor is on the order of 1/100\(^\circ\) of an inch, and each step moves the motor a constant distance.

The camera provides a 320x240 pixel image. It uses a wide angle-lens, which provides a field of view of 110\(^\circ\) – this allows the camera to see the ceiling and floor in a 10-foot high space when it is just 28” from the wall. The image quality from the camera is low (see Figure 31) and is very distorted by fish-eye effects from the lens; however, the center of the image is undistorted and suffers only from compression along the vertical axis.
By using the stepper motor to move the camera in fine, highly controlled gradations, evenly spaced bands of pixels from the center of each position can be captured. These bands are distorted only by compression along the vertical axis, which can be corrected via a simple linear transform (see section 4.9.4.2 above). These transformed bands can then be tiled together to form an undistorted swath of ceiling-to-floor texture (see Figure 26 for an example of a properly tiled texture.)

There are a number of issues involved in the design, implementation, and calibration of such a capture system.

4.9.5.1 Rack Design

Figure 32 shows the basic design of the texture rack. The belt-fed track and sled assembly was purchased as a single unit. This included the stepper motor, but none of the associated hardware or software to control it. The camera, lasers, edge sensors, controller, and battery pack were added later.
The lasers are used to align the rack parallel and at a fixed distance from the wall. The two outermost lasers project a beam straight ahead, and the angled lasers cross. When the beam from angled laser on the left strikes the same point as the beam from the straight-ahead laser on the right, the right edge of the rack is exactly 28" from the wall. This was done by placing the rack at this distance, moving the lasers until they were aligned, and then gluing them in place. Basic geometry guarantees that this is the only position where the lasers will align. When the angled laser on the right and the straight laser on the left are also aligned in this manner, the rack must be parallel to the wall, because both end points are equidistant. Figure 33 shows this.

The edge sensors determine when the camera sled has reached the left or right edge of the rack. When the rack is over them, an optical switch is closed; this switch can be read by the controller.

The battery pack uses standard Alkaline battery cells to provide a 1.5V and 12V power supply. The stepper motor is driven at 1.5V; the controller and lasers require 12V.

The controller is the heart of the texture rack. It is driven by a Basic Stamp II IC, which is a BASIC-programmable microprocessor with 16 I/O lines and a serial port. The optical sensors are hooked directly into the I/O lines. The BasicStamp is capable of directly reading the sensor’s state – they are “on” if the sled is at the edge of the rack and off otherwise. All four lasers are switched on and off through a single transistor driven from one of the I/O lines. The lasers are left on except when the motor is moving.

![Diagram of laser alignment and sensor placement.](image)

**Figure 33:** Four lasers allow texture-rack to be aligned at a fixed distance parallel to the wall.

The stepper motor is driven by a bridge of four transistors. Stepper motors consist of four coils, which are turned on in alternating pairs to cause the coil in the motor to rotate a fixed amount with each “step”. Each of these coils has a separate input transistor switched by one of the I/O lines. (For more details about
circuits and signals to control stepper motors, see [7]). The IC is controlled via a simple program which reads commands from a desktop computer over the serial port; these commands consist of a number of steps for the motor to move in either direction. The IC returns an acknowledgement indicating the command was completed and the status of the edge-sensors. The controlling program, in this case, is the TripodGrabber application (see Section 4.9.3).

The camera and serial port are connected to the computer via a simple interface board in the controller box. The camera emits a composite-video signal which is passed directly into a video capture card on the desktop computer (the BasicStamp does not see or manipulate the video signal.)

The rack is 43.5" and 4177 stepper-motor steps long. The camera is located 57" off the ground. It takes 28 seconds to move the camera from one end to the other, and 68 seconds to capture a full set of frames via TripodGrabber with 10 steps between each frame, using a PowerBook G3 with a iRez CapSure video capture card.

Figure 34 shows a photograph of the rack, with insets detailing the laser and camera assembly.

![Figure 34](image)

**Figure 34:** Photographs of the texture rack. The left inset shows the laser assembly, the right the camera and sled.

It’s important to note that the texture-acquisition rack is not central to the TOADS project. Any other video input device, including a digital camera, could be used in its place, given appropriate interface programs (like the TripodGrabber) which can capture images from them. The rack is an example of such a device, and was built because it is faster and produces better-aligned images than a digital camera.

### 4.10 Paths

The Paths tool ( ) is used to define habitrails in drawings. Habitrails consist of a series of connected parallelograms, with transparent collision detecting edges on one pair of parallel sides and transparent, non-collision detecting edges on the other pair. It works somewhat differently from the other tools in the TOADS renderer; The process for defining a path is as follows:
- Select the path tool.
- Click and hold the mouse button at the first point in the path.
- Drag to define the first non-collision-detecting edge.
- Release the mouse button and position the cursor so that the two collision detecting edges are located appropriately.
- Holding down the option key will widen or narrow the path.
- Click the mouse button and the second non-collision-detecting edge will appear.
- The next path segment is now being defined, as in step 4.
- Press the escape key when no more path segments are desired.
 Paths are created as a number of lines (rather than polygons), so individual walls of paths can be selected, deleted, and have collision detection toggled on or off. Paths are drawn in red; collision detecting edges are dark red, non-collision detecting light red.

Paths are automatically inserted into the “Paths” layer, which will be added to a drawing when the first path is created. Paths are exported as a set of lines, rather than as a polygon, to allow collision detection to be specified independently for each edge. Each of the lines is assigned an object type of Path by default.

4.10.1 Path Implementation

The implementation of paths is extremely simple, because they are just normal line objects in the DXF model. Collision detection is disabled for some of the edges, and they are assigned a transparent texture by default. Other than this, there are no special comments or data-structures required for paths once they've been inserted.

4.11 Doors

As mentioned above, TOADS is capable of auto-detecting doors in DXF files and converting them to polygons which can open and close. When a model is loaded, every arc object is examined to see if there exists a line object which forms a radius of the arc. When such a radius is found, a door object is added to the model. Door objects don’t have any appearance in the DXF image (they don’t implement the Draw method), but do generate polygons when the model is exported to 3D. Doors also aren’t written to saved DXF files – they are always generated dynamically from arcs.

On export, doors produce two polygons – one for the open state of the door and one for the closed state. The closed position is assumed to correspond exactly to the edge-radius of the arc which was found when the door was generated. The open position is the radius from center to the end of the arc not used for the closed position. Section 4.12 below discusses how openable objects with two-states, such as doors, are handled by the 3D export module and the ToVRML tool.

4.12 3D File Export

At any point in the process of model design, a 3D file can be generated. TOADS uses an intermediate file format which is not directly useable in existing 3D rendering tools. As mentioned previously, this is done primarily to divorce the main TOADS program from any knowledge of specific file formats; users can use the provided VRML conversion tool, or write their own converters, to generate final, useable models.

Exporting is done by selecting the “Export 3D Data...” option from the File menu. The output file is specified through a standard file dialog, and the file is written to the specified location. In addition to the model itself, TOADS also provides facilities to move and auto-convert the PICT files used for textures into JPEG or GIF files commonly used by renderers such as VRML. This option is turned on via the Preferences dialog (see Section 4.8). These options save the user from the tedious task of manually converting many textures and also collect all of the textures from a VE into a single directory along with the model file.
4.12.1 Implementation

Figure 35 shows the main modules of the 3D exporter. IO3D is the module which is responsible for writing and reading 3D format files and for storing data structures which represent 3D polygons and textures. MacIO3D is a subclass of this module which implements Macintosh UI features, such as Standard File and Status dialogs, which reflect the state of the 3D file export.

Exporting is done in three stages:
- Collection of 3D representation of DXF objects
- Collection of textures
- Writing objects and textures

Collecting 3D objects is done via the DXFPicture class' Add3DData method, which calls the Add3DData method of every object in the drawing. Each DXF object type implements this method; objects call the IO3D::AddPolygon method to export their 3D representations. Many DXF objects, such as arcs, circles, and text, export no polygons and hence have no representation in the 3D environment. Vertex-based objects, such as lines, polylines, and solids, use the DXFLine::Make3DPolyFromLine static method to export flat polygons which represent them. This method simply converts a line (two x-y coordinate pairs) into a 3D polygon (4 x-y-z vertices) with a specified height. It converts the internal DXF units which objects are stored in to meters, so the 3D file has meters as its basic unit of measurement.

The textures and ceiling heights used for each polygon are determined through a simple inheritance graph, as shown in Figure 36. Objects can have their own textures and ceiling heights; if specified, these are used. Otherwise, textures for walls are first drawn from the room the object belongs to – if the object isn’t a wall, or doesn’t belong to a room which specifies a wall texture, the default texture (from the preferences) for that object’s type is used. Heights for all objects are inherited first from any enclosing room, and then, if the object doesn’t belong to a room which specifies a height, from the default height for the document.
There are a few more complexities to the object collection process. First, PolyLines may have a single texture wrapped around their outsides; thus, a single texture file is applied to many different polygons. Textures must be subdivided so that each face in a 3D PolyLine has a portion of the texture proportional to its size applied to it. The 3D file format stores information about the range of coordinates in a polygon's texture which are applied to that polygon. The Make3DPolyFromLine method accepts these coordinates, and the PolyLine::Add3DData method generates them before calling Make3DPolyFromLine. Figure 9 above shows how a texture is wrapped around a four-sided polygon.

The second extension to object collision has to do with 3D polygons that can be in multiple positions—doors, for example, can be opened and thus have two positions. The Make3DPolyFromLine method accepts an optional parameter which specifies another 3D polygon which this polygon should change into when it is clicked upon. Doors are thus generated via the following pseudo-code:

```cpp
openPoly = Make3DPolyFromLine(xOpenCoord, yOpenCoord, height, noMotion)
closePoly = Make3DPolyFromLine(xClosedCoord, yClosedCoord, height, openPoly)
io3D.AddPolygon(closePoly)
```

Adding closePoly to io3D also implicitly adds openPoly, since it is linked to closePoly. When a VRML file is finally generated, the door will initially appear in its closed state. When it is clicked on, the closed door polygon will disappear and the open door will appear. When the open door is clicked on, the door will close again.

Collecting textures during Polygon export is easy. When a 3D polygon is created, it is given a text-based path to a texture file. When each polygon is added, a list of known textures is checked to see if this texture has been seen before; if it has, it can be replaced by a reference to the pre-existing texture. If not, the texture is added to the list of known textures and a new reference is generated. If texture conversion is to occur, it is done by the MacIO3D module at this stage. Before a reference to a texture is saved, QuickTime is invoked to convert the file and re-write it into the same location as the main 3D data file.
Writing out the file then consists of writing the list of known textures and references followed by a list of each polygon. The specific format is described in the Section 4.12.2.

Two additional polygons for the ceiling and floor are included in every model. These polygons are determined by taking the convex-hull of the vertices which make up the model. The ceiling is placed at the height of the ceiling specified in the preferences module, and the floor at a height of zero. They are textured with their respective default textures specified in the preferences.

The convex-hulls algorithm used is known as a Graham Scan. It's a well known algorithm from computational geometry [12]. This algorithm produces a convex set of points which form the boundary of a model, thus, it is not a perfect outline of the outermost points, especially if the model has concave edges. Figure 37 shows a floorplan and the resulting convex hull. Notice that the hull is not a perfect outline.

One reason to provide convex-hulls functionality is to enable ceiling and floor detection for rooms; this functionality is not currently enabled, but the convex-hulls code is designed to make it straightforward to generate these polygons in a future TOADS revision.

**Figure 37**: A model and its convex hull.

4.12.2 The Intermediate File

The intermediate file format is specified in Appendix C below. It basically of a list of textures followed by a list of polygons with coordinates specified in meters.. It is a concise, text-based file which is easily parsed and written. Appendix D summarizes the IO3D interface, which provides routines to read and write the intermediate file – it is of interest to users who wish to write tools to generate 3D files for specific formats, such as VRML.
4.12.3 ToVRML

The ToVRML tool converts intermediate 3D files to VRML. It uses the IO3D class to read and parse the file format, and then outputs version 2 VRML corresponding the model. This tool fully supports all of the specifications included in the 3D file format, including:

- Tiled, transparent textures
- Openable polygons
- Viewpoints
- Collision Detection
- Light Sources

The interface is extremely simple – the user is prompted with a standard file dialog from which an intermediate 3D file is selected. This file is converted and a .wrl suffix is appended to its name. The resulting file can be viewed in any VRML version 2 compliant browser, such as Netscape Navigator with CosmoPlayer.

For the most part, the tool just performs a direct translation of coordinates into VRML format. The y and z coordinates are switched between the intermediate format and the VRML file. Otherwise, however, this the translation is just a less concise repackaging of the coordinates.

The VRML gets tricky when it comes to enabling openable doors: embedded java-scripts are required. A script with a static boolean variable is associated with each open and closed door polygon; this static variable activates a switch-node which determines which of the two polygons is visible. Thus, a pair of door polygons looks as follows:

```xml
DEF DoorScriptO Script {
  eventIn SFTime hitIt
  eventOut SFInt32 switchVal
  url "javascript:" function hitIt() {
    if (switchVal == 0) switchVal = 1;
    else switchVal = 0;
  }
}
DEF DS1 Switch {
  whichChoice 0
  choice [
    Group {
      children [
        DEF OpenDoor2 TouchSensor { },
        Shape {
          ... door geometry ...
        }
      ]
    }
    Group {
      children [
        DEF OpenDoor3 TouchSensor { },
        Shape {
          ... door geometry ...
        }
      ]
    }
  ]
}
```
The first two ROUTE commands connect the script to the TouchSensors on the door polygons – it is activated anytime either of the doors is clicked. The last ROUTE command selects which of the two states the Switch node is in, which determines which of the two polygons is visible.

One reason VRML was selected is because it allows embedded scripts of this sort. Other 3D rendering toolkits, such as Inventor, require the user to write and compile separate programs which control the status of objects in the scene. Writing a tool to generate Inventor environments with interactive doors would be considerably harder.

Notice that the ToVRML program makes no attempt to verify the validity or availability of texture files. It’s the user’s responsibility to make sure the appropriate texture files are in the same directory as the .wrl file when the model is to be browsed. However, the auto-convert feature of the TOADS program moves and converts textures automatically, so the user shouldn’t normally have to worry about this.

### 4.13 Limitations and Future Enhancements

Appendix B consists of a list of known bugs and short-term future enhancements for current prototype TOADS software system. In the process of designing this system, however, a number of fundamental limitations and useful long-term enhancements have come to light.

#### 4.13.1 Texture Acquisition Hardware

The current system is predicated on the idea that an easy-to-use, high speed computer controlled texture acquisition device is available. The texture-rack prototype deployed for this project is not ideal, for several reasons. First, its relatively short length means that it doesn’t capture an area much larger than a conventional digital camera, although it does allow more precise control over the width of the area and eliminates the need for tedious photo-retouching and manual alignment of images. Second, its width makes it virtually unusable for texture acquisition in small spaces. Because it must be a fixed distance from the area it’s acquiring, it requires shuffling of furniture and other large objects to get close enough to walls to acquire them properly. It doesn’t offer any provision for scanning floors, ceilings, or tops of objects.

The initial plans for the TOADS system called for developing a semi-autonomous robot which would work in much the same way as the texture rack – by navigating parallel to walls, it could acquire and tile long bands of texture. This plan was abandoned due to lack of a suitable robot, but should be practical given access to new hardware. Such a system could acquire longer bands of texture and function in smaller spaces; it might also be able to make some inferences from the floorplan to locate features and automatically acquire texture for them.

Researchers around the country are working on robot-based systems which do this sort of texture acquisition. At MIT, a large, manned robot system that acquires outdoor textures from a number of static pictures of the environment has been developed [8]. This system includes hardware to capture textures for all exterior faces of buildings which could be fed into an enhanced version of the TOADS program.
4.13.2 Database Enhancements

The original vision of the TOADS system incorporated a number of different types of visual and textual information linked into floorplans; not only would environment-designers be able to feed in information about texture and ceiling heights, but they could provide historical text and pictures of sites, sound and video clips, and links to HTML documents of relevance. Users would have the option of exporting end-products other than 3D environments; for example, a floorplan might export to an HTML document containing an imagemap of the floorplan where each room could be clicked on to receive photographs and historical documents relating to that location. Or, additional textual information could be linked into the environment so that important pictures and text appeared in the VE when users entered a room or reached particular points-of-interest. Researchers at Columbia are currently developing augmented reality systems to allow data such as pictures, sounds, and video-clips to be overlayed on top of real-world views of scenes. This software is based on associating those data points with a blueprint and then bringing up the appropriate data when the user is located or looking at a particular point [9]. TOADS could be adapted to allow users to place text, video, and pictures and then export files compatible with the Columbia system.

4.13.3 Acoustic Environments

Some work was done to allow the system to provide acoustic realism, though these features were never fully incorporated. Though not documented here, the TripodGrabber application includes a feature which allows small samples of sound to be attached to frames in the image. If these sound samples consist of recordings of high-amplitude, low-duration pops – impulses – in the location where the texture was acquired, then the impulse response of different locations in the environment can be expressed. The impulse response of a system (acoustic or otherwise) allows signals to be reproduced as if they were passed through that system by convolving those signals with the impulse response. So, sound-effects can be made to sound as if they were produced in the location where the impulse response was recorded; since rooms have significant damping and echo effects, recreating sound-effects realistically can have an important effect on the degree to which an environment feels natural.

Generating impulses can be done via a number of mechanisms. The approach currently being tested consists of a cap-gun being fired from a fixed-location relative to a microphone. The cap-gun is hooked up to a small switch which flips just before the gun goes off. When the switch flips, a signal is sent to the software which should begin recording. Impulses will be easily connected to locations in the environment by firing the cap-gun as each texture-band is acquired. Caps produce a loud, short noise which is evenly distributed throughout the frequency spectrum, which are three characteristics desired of an ideal impulse. The hardware and firmware (software to read the switch) are currently in place, although they are not connected to the QuickTime Grabber software in any way. Furthermore, the VE export code doesn’t include an facilities for sounds or impulse-responses.

5. Results

The goal of the TOADS project was to produce an innovative, useable system for designing 3D virtual environments in two-dimensions. Though the system is to be used to generate environments for experiments on spatial-knowledge acquisition, this wasn’t the focus of this project. In order to demonstrate the usefulness of the system, however, it is important to examine real world environments built with TOADS.

The first environment created was a model of the lobby of the 7th floor of building 36 at MIT. This model was constructed in December 1998, before the texture acquisition hardware was complete, so textures were
captured via a digital camera. It took about 3 hours to capture and align the textures, and another half-hour to apply and properly set up the ceiling heights and lighting in the TOADS system. The generated VE is available online at [http://pellicle.mit.edu/7thfloor/7thfloor.wrl](http://pellicle.mit.edu/7thfloor/7thfloor.wrl); it requires a VRML browser, such as CosmoPlayer, to be viewed properly (a fast net connection is also recommended.) A sample screenshot is shown in Figure 38. In this image, the drinking fountain is a flat texture – notice that from a distance it's not possible to see the lack of depth. The floor and ceiling are handled through a tiled texture.

![Screen shot of the lobby of the 7th floor of building 36.](image)

5.1 7th Floor Model

With the addition of real texture-acquisition hardware, it became possible to build a more complete model. Once again, the 7th floor of building 36 was used, but this time the model includes the main common space of the lab – roughly four times the number of textures as the lobby model. Figure 39 shows TOADS model, with the designed areas circled.
In building this model, nearly all features of the TOADS system were exercised. Large, flat texture areas were acquired using the texture rack, and smaller textures via a digital camera. All of the furniture is modeled via the junk tool, so can be readily moved and rotated. The area outside of the lobby is set to belong to a single large room which sets up default wall, floor, and ceiling textures different from the default textures for the lobby. The windows on the lobby are transparent textures. Many of the doors and wall paintings were built using designer textures, which allowed them to be set at the proper height amongst plain-wall textures. Figures 41-43 shows several different screen shots from the model.

Figure 39: The Building 36 7th floor model, with fully textured areas indicated in black.

Figure 40: The fridge and microwave.
The bulk of the model was built in a weekend and required about 10 hours to build, including time to acquire textures. This is an extremely significant improvement over traditional architectural model design times. Furthermore, it's very easy to modify this model – if a piece of furniture needs to be moved or added, only a few seconds of editing are required.
6. Conclusions

The TOADS tool suite provides a powerful environment for designing two-dimensional interior-space virtual environments. Users can import and edit DXF models, capture and design textures, and generate VRML environments. The system in its current form is a functional software tool that has been used to generate realistic virtual environments in a fraction of the time required using previous state-of-the-art tools. Some important overall principles that guided the implementation of TOADS were presented: first, designing virtual environments in two dimensions is more intuitive and easier for users than the clumsy three-dimensional design tools normally used. Second, texture-acquisition, editing, management, and application is by far the most difficult part of virtual environment design – TOADS includes a powerful set of tools which simplify this process greatly. Finally, providing intelligent defaults for textures and ceiling heights and making inferences about doors and windows reduces the overhead and data-management burden for users.

The current version is a highly-usable software tool consisting of more than 20,000 lines of C++ code. Considerable effort has been made to make this code maintainable and clear, and this document attempts to summarize its higher level structure and show how the various tools in the TOADS suite work together to provide a complete 3D environment design system. The Macintosh user-interface is designed to be of professional quality, consisting of more than twenty separate dialog boxes, and nine DXF-editing tools. Some time has been invested to make the system lean and efficient; it is faster than commercial quality drafting programs at reading and rendering DXF files. The system has a number of small problems (summarized in Appendix B) which may be corrected in the near future. More far-reaching future extensions include support for HTML-model export, acoustic environment modeling, and fancier texture-acquisition hardware.

The Sensory Communication Group at MIT looks forward to using the TOADS system as an integral part of its VE research, both because it greatly simplifies the process of designing environments and because it provides a novel way of organizing and viewing three-dimensional data.
Appendix A - DXFComment Extensions

Comments are embedded by TOADS into DXF files to store information specific to the TOADS program. The following table summarizes those comments, their function, and the section of this document which discusses them more thoroughly.

<table>
<thead>
<tr>
<th>Comment Name</th>
<th>type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kPolyRoomComment</td>
<td>1</td>
<td>This PolyLine defines a room. Comment appears with the body of a PolyLine definition. Data = filled, height, name where filled indicates if the room is drawn with a fill color, height indicates the ceiling height for the room, and name specifies the name of the room. (4.7)</td>
</tr>
<tr>
<td>kInRoomComment</td>
<td>2</td>
<td>This object belongs to a room. Objects may belong to multiple rooms. Comment appears within the body of an object definition. Data = filled, height, name, floorTexture, ceilingTexture where filled, height, name are as above and floorTexture and ceilingTexture specify the textures for the floor and ceiling of the room. (4.7)</td>
</tr>
<tr>
<td>kTextureComment</td>
<td>3</td>
<td>This object has a texture. Objects may contain only one texture. Comment appears within the body of an object definition. Data = null, transparency, repeatX, repeatY, numX, numY, path where null indicates if the texture is non-empty, transparency the percentage of light which passes through the texture, repeatX and repeatY if the texture is tiled, numX and numY the number of tiles in each direction, and path the location of the texture data on disk. (4.9)</td>
</tr>
<tr>
<td>kPreferencesComment</td>
<td>4</td>
<td>Preferences for the document. There can be only one preferences comment per document; it is a highest-level member of DXFPicture. Data = ceilingHeight, metersPerDXFUnit, grabberSizeInMeters, autoConvert, saveInExportDirectory, startX, startY, startZ, wallTexture, doorTexture, windowTexture, pathTexture, floorTexture, ceilingTexture, compressor where ceilingHeight is the default height for ceilings, metersPerDXFUnit is the conversion factor between DXF units and meters, grabberSizeInMeters is the length, in meters, of the swath the texture acquisition device scans, autoConvert indicates if texture PICT files are to be converted to a different graphics format, saveInExportDirectory indicates if converted PICT files are to be save with the intermediate 3D file, (startX, startY, startZ) indicates the origin of the viewer in the model, and compressor indicates the QuickTime converter which is to be used to export PICT files. The texture fields are texture comments which indicate the default textures for different object types. (4.8)</td>
</tr>
<tr>
<td>kObjectClassComment</td>
<td>5</td>
<td>Indicates the class this object belongs to. Appears only once per object, in the body of the object definition. data = objectClass where objectClass is an integer identifier for the class of the object. Classes, with numbering beginning from 0, are</td>
</tr>
<tr>
<td>Comment</td>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>kCollisionDetectOffComment</td>
<td>6</td>
<td>Indicates that this object should not have collision detection enabled in the intermediate 3D format. Appears at most once per object, in the body of the object definition. No data. (4.5)</td>
</tr>
<tr>
<td>kSolidHasBeenMadeIntoLinesComment</td>
<td>7</td>
<td>Indicates that DXFLine objects have been generated for the edges of the containing DXFSolid object. Appears at most once per DXFSolid object, in the body of the solid definition. DXFSolids are converted to lines so that textures can be applied to each of their walls; this comments prevents this conversion from happening each time the document is opened.</td>
</tr>
<tr>
<td>kObjectHeightComment</td>
<td>8</td>
<td>Indicates the height of the object. Each object may contain only one height comment. Comment appears in the body of the object definition. data = objectHeight, where objectHeight is a floating-point number which specifies the height of the object in meters. (4.5)</td>
</tr>
<tr>
<td>kLightComment</td>
<td>9</td>
<td>Indicates the presence of a light-source. There may be many light sources; these comments appear at the same level as the preferences comment: in the top level DXFPicture definition. data = radiance, type, x, y where radiance is the intensity of the light, type is the class of light source (currently only a value of 1, indicating a diffuse light source, is supported), and (x,y) indicate the x-y location of the light. (4.4.2)</td>
</tr>
<tr>
<td>kDesignerTextureComment</td>
<td>10</td>
<td>Indicates that this object has a designed texture. May appear once per object, in the body of the definition; a normal texture may also appear. Data = designerTexture, where designerTexture is a path to a file which specifies a designed texture. (4.9.1)</td>
</tr>
<tr>
<td>kPolyLineTopComment</td>
<td>11</td>
<td>Indicates that this polyline has a separate texture for it's top polygon. May appear once per polyline, in the body of the definition. Data is the same as for a regular texture comment. (4.9.2)</td>
</tr>
<tr>
<td>kBaseHeightComment</td>
<td>12</td>
<td>Indicates that this polyline has a base altitude – that is, it floats off the floor. May appear once per polyline, in the body of the definition. Data = baseHeight where baseHeight is a floating-point height in meters. (4.9.2)</td>
</tr>
</tbody>
</table>

wall, door, window, path, stair, elevator, junk, and unknown. (4.5)
Appendix B – Bugs and Imminent Changes

The current TOADS implementation is limited in a number of ways; the following is a compendium of known problems, limitations, inconsistencies, and desirable features which should be included in a finalized system. Note that there are still overarching goals of designing a system which is fully extensible that reaches beyond a tool for designing just architectural environments and works for outdoor and underwater models as well.

Known Bugs

- Windows can’t be resized properly on auxiliary screens. DXFWindows can only be made as large as the size of the primary screen; extra screen real-estate on a large second screen will be unusable.
- Objects cannot be removed from rooms once they’ve been added.
- Lights can’t be removed from the environment.
- The texture grabber rectangle doesn’t always appear on the same side of the wall; it should appear on the side which is most outward facing.

Inconsistencies

- Object selection is not entirely intuitive. In particular, some objects, such as Arcs, are still selected via the bounding box. A tighter bound should be used.
- The DXFParser doesn’t support DXF files which contain blocks.
- The Room/Path/Junk definition tools all work by tracing PolyLines in the environment, although their interface is not consistent (in particular, the Path definition tool works differently from the other two and is unlike anything in any program I’ve ever used.)
- Ceiling and floor polygons are not allowed for individual rooms, although textures can be specified. The ConvexHulls code should make this relatively straightforward to implement.

Small Enhancements

- Currently, no warning is given when the system is low on memory, and when memory does run out, the user is forced to close the front-most window, often without being allowed to attempt to save. This is bad.
- Undo and Revert are not supported. Undo, in particular, would be useful.
- DXF files saved with TOADS should have their own file type so double-clicking them opens the TOADS program.
- Multiple points of interest should be supported in VRML files.
- Objects grouping and ungrouping. This would allow walls which are composed of several small lines to be group, textured, and moved together.
- Lights are of a fixed intensity and type. Adding a dialog to allow specification of these settings (which are stored and exported, just unspecifiable), would be useful and easy.
- The TextureGrabber dialog should preview movies and pictures so that the user can tell what texture he’s applying and how it will look.
- Texture coordinates should be specifiable for textures, since there is currently no way to use just a piece of texture file for a polygon.
- The user should be allowed to specify the color with which textured-walls are drawn. Currently, they’re always blue.
Major Enhancements
- Importing and exporting PICT files would be a very Macintosh-like thing to do, since it’s the preferred format on the platform. PICT is currently used for the internal representation of images, so exporting is easy. Importing requires writing a new DXFParser-like front end.
- A quick-texturing window, which allows a user to click on a wall and select a texture from a list of texture thumbnails previously defined would be a lot easier than navigating through the object-info and texture dialog boxes every time a object needs to have a texture associated with it.
- Allowing users to specify their own object types would allow more customizability in terms of textures – some office spaces, for example, might contain many “desks”, all of which should have the same texture.
- Walls have only one texture, even though a single line can form the boundary of two rooms and thus could have different images on each side. Walls should have two textures, one for each side.
- Since the Macintosh supports 3D images natively via QuickDraw 3D, allowing users to preview objects (such as walls and PolyLines) would be possible and useful, since exporting fully to a VE is time consuming.
Appendix C – Intermediate File Format

The intermediate file format is very simple. It consists of a list of textures, followed by a list of polygons, followed by a list of light sources. The exact format is as follows: the first field is the starting viewpoint and viewer settings.

\[ \text{startx, starty, startz, rotx, roty, rotz, rotads, fov, name} \]

where (startx, starty, startz) specifies the starting location, (rotx, roty, rotz) specifies the axis around which the viewpoint rotates, rotads specifies the number of radians the viewpoint is rotated, fov specifies the field of view of the viewer, and name specifies the name of the viewpoint. This format is chosen to be directly compatible with the VRML viewpoint node.

Textures are listed next:

\[ \text{textureCount, transparency, repeatX, repeatY, xTiles, yTiles, filePath} \]

... more textures ...

Where textureCount specifies the number of textures, transparency indicates the percentage transparency of the texture, repeatX and repeatY specify whether the texture is tiled in the X and Y directions with xTiles and yTiles as the number of tiles in each direction, and filePath is the path to the texture image on disk.

A list of polygons then appear

\[ \text{polygonCount, vertexCount, x1, y1, z1, ..., textureNum, collide, tiled, startX, endX, hasOtherPos} \]

otherPoly

... more polygons ...

Where polygonCount indicates the number of polygons. VertexCount indicates the number of vertices which appear, followed by that number of (x,y,z) triples. Most polygons consists of just four vertices, although ceilings and floors often have more than that. TextureNum is an index into the texture table indicating which texture to apply; the tiled field indicates if the entire texture is used, or just a portion of it; startX and endX indicate the region of the texture to use (the entire height is always used). The collide field indicates whether collision detection is enabled for the polygon. If hasOtherPos is 1, the next polygon is the polygon which the current object transforms to when it is clicked on.

The coordinate axis for vertices is shown in Figure 43. Notice that the Y and Z axes are switched relative to some coordinate space definitions (in particular, VRML).
Finally, a list of lights is given:

\begin{verbatim}
lightCount
intensity, xPos, yPos, zPos
\end{verbatim}

Lights are always diffuse (point) light sources, similar to light bulbs which radiate in all directions. Intensity indicates the brightness of the light, and (xPos, yPos, zPos) is its location. As mentioned previously, better support for a variety of light sources is an important future extension to TOADS.
Appendix D - IO3D.h

Fortunately, writers of new 3D export tools aren’t required to parse the 3D file format themselves. The IO3D class provides a simple interface for loading these files, and provides accessor classes for viewpoints, textures, polygons, and lights.

The following class definitions, from IO3D.h, implement a simple system to read and write the 3D intermediate file format. Five classes are provided:
- Texture: Maintains references to texture files.
- Poly3D: Maintains 3D polygons consisting of 4 xyz triplets specifying the corners of a polygon. Also maintains a collision-detection flag and a texture index.
- Viewpoint: Maintains a viewpoint (starting location) in the environment
- Light: Describes a point light source in the environment.
- IO3D: Main storage class. Maintains a list of textures and a list of Poly3Ds and includes routines to flatten the structure to disk and load it back in from disk.

These classes should be compatible with any ANSI Compliant C++ compiler. To load a IO3D structure from disk, call IO3D::Load, as follows:

```c
IO3D *myIO3D = new IO3D();
if (myIO3D->Load("saved3Dfile") == 1) //error
    printf("Error loading 3D data file.");
else
    ... do something with the 3D data ...
```

Once the structure has been loaded, several accessors are provided to loop through textures and polygons. In the following example, a routine InsertPolyIntoNativeFormat() is called for each polygon. A Poly3D and Texture are passed into each call.

```c
for (int i = 0; i < myIO3D->NumPolys(); i++) {
    Poly3D p;
    Texture t("");
    if (myIO3D->GetPoly(i, &p) == 1)
        printf("Error loading poly %d", i);
    else {
        if (myIO3D->GetTexture(p.textureIdx, &t) == -1)
            printf("Error loading texture %d", p.textureIdx);
        else
            InsertPolyIntoNativeFormat(p, t);
    }
}
```

Note that the textures are specified as Mac-style, colon-delimited file names, which will not be acceptable for PC or Unix systems. Furthermore, paths are specified from the root directory of the machine which created them, and are thus unlikely to be valid on a new machine. Finally, texture files on the Mac are typically stored as PICTs, which are poorly supported on other machines. TOADS solves these problems via the auto-convert settings in the preferences dialog (see Section 4.8). Textures can converted to more useable picture format (such as JPEG) and moved into the same directory as the 3D file, so that no directory-delimiters are needed.
Each of the five classes provides accessors which allow their contents to be read. The remainder of this section consists of the IO3D.h header file in its entirety.

/* IO3D.h
Copyright 1999, Sam Madden
MIT License

Header file for intermediate 3D file interface classes.
Provides data structures to maintain lists of textures, polygons, lights, and viewpoints.
*/

#ifndef __IO3D_
#define __IO3D_

/* A simple representation for a texture file —
just a 255 character string.
*/
class Texture {
public:

// Init
Texture(char *name);

// Return a copy of the name of texture file
void GetName(char *str);

// Compare to see if this texture is equal to another
bool operator == (Texture t);

// Specify the location of the converted texture representing this file
void SetConvertedName(char *str);

// Get the location of the converted texture representing this file
void GetConvertedName(char *str);

// Set and get the transparency of this texture
void SetTransparency(float trans) {transparency = trans;}
float GetTransparency() {return transparency;}

// Set and get the x and y repeat settings
void SetRepeat(bool x, bool y) {repeatX = x; repeatY = y; }
void GetRepeat(bool *x, bool *y) {*x = repeatX; *y = repeatY;}

// Set and get the number of tiles in the X and Y direction
void SetNumTiles(int numX, int numY);
int XTiles() {return xTiles;}
int YTiles() {return yTiles;}

private:
char name[255];
char converted_name[255];
float transparency;
bool repeatX, repeatY;
int xTiles, yTiles;
};

/* Just a set of three points — so that Poly3D can pass vertices around easily.
*/

// Vertex3D
class Vertex3D {
public:

Vertex3D(float x, float y, float z) {
    Set(x, y, z);
}

Vertex3D() {
}
void Get(float *x, float *y, float *z) {
    *x = this->x;
    *y = this->y;
    *z = this->z;
}
float GetX() {return x;}
float GetY() {return y;}
float GetZ() {return z;}
void Set(float x, float y, float z) {
    this->x = x;
    this->y = y;
    this->z = z;
}
private:
    float x,y,z;
};

/* Poly3D */

3D Representation of a polygon, consisting of 4 points in 3-space.

Kinda a nasty def, since it allows access to fields, but manipulation is otherwise very klunky through accessors.

TextureIdx is an integer identifier which establishes a correspondence between a texture and this polygon. We specify textures by integer to minimize the space requirements of keeping many polygons in memory. The texture which textureIdx refers to is stored externally (typically by IO3D).

The SetTiled() routine toggles polygons between one of two states: normal: Apply the entire texture file to the entire polygon tiled: Split the texture into (the value of the num_tiles returned from IsTiled()) tiles and use the value returned in the tile_no field of IsTiled() for this polygon.

Note that the method of splitting is unspecified; tiles might all be the same size or might be split based upon relative polygon size.

The collide field specifies whether collision detect should be enabled for this object.
*/

#define kTextureType_Count 2
typedef enum {
    TNormal = 1,
    TTiled = 2
}TextureType;

class Poly3D {
public:
    /* Init the poly */
Poly3D( int textureIdx,
        bool isCollisionDetect);

Poly3D(){
    DoInit();
}
void DoInit();
-Poly3D();

//Fields (public for accessor efficiency)
int textureIdx;
bool collide;

//return the current texture number
int GetTexture();
//set this polygon's texture as tile or untiled
void SetTiled(bool tiled, float startx, float endx);
bool IsTiled(float *startx, float *endx);
Vertex3D GetVertex(int i);
void AddVertex (Vertex3D v);
int CountVertices();
void SetOtherPos(Poly3D *p); //allow polys are moveable through two positions....
Poly3D *GetOtherPos();

private:
    float startx, endx;
    TextureType tex_type;
    Vertex3D *vertices;
    int numVertices;
    Poly3D *otherPoly;
};

/* Viewpoint */

Stores a 7 dimensional representation of a viewpoint, along with a name 
Includes rotation, position, and field of view. */
class Viewpoint {
    public:
        Viewpoint();
        /* The default viewpoint, orientation, and field of view */
        //The viewpoint is the location where the user starts out 
        //Specified in x,y, and z coordinates 
        void SetViewpoint(double x, double y, double z);
        void GetViewpoint(double *x, double *y, double *z);

        //The orientation is the rotation of the viewing camera 
        //Specified in radians of rotation about the axis specified by x, y, and z
        void SetOrientation(double x, double y, double z, double rads);
        void GetOrientation(double *x, double *y, double *z, double *rads);

        //The field of view is the width the of viewing camera, in radians 
        void SetFOV(double fov);
        void GetFOV(double *fov);

        void SetName(char *name);
        void GetName(char *name);
    private:
        double xpos, ypos, zpos, xrot, yrot, zrot, rotrads, fov;
        char name[30];
};
the light class keeps track of lights in the environment
for now, we only support diffuse lighting
class Light {
  public:
    Light(float i, float x, float y, float z) { 
      xLoc = x; yLoc = y; zLoc = z; intensity = i;
    }
    float GetIntensity() {return intensity;}
    float GetX() { return xLoc; }
    float GetY() { return yLoc; }
    float GetZ() { return zLoc; }
    void SetIntensity(float i) {intensity = i;}
    void SetX(float x) {xLoc = x;}
    void SetY(float y) {yLoc = y;}
    void SetZ(float z) {zLoc = z;}
  private:
    float xLoc, yLoc, zLoc;
    float intensity;
};
/* I03D
Stores 3D polygons and textures corresponding to a DXF file translated into
3-space.
Provides accessors for a lists of polygons and textures, as well as routines
to read and write the class to a text file.
*/
class I03D {
  public:
    I03D(); //constructor
    ~I03D(); //destructor

    /* Write to the specified text file
       Overwrites existing contents
       Returns 0 if successful, -1 on failure.
    */
    int Load(char *file);

    /* Read from the specified text file
       Returns 0 if successful, -1 on failure.
    */
    int Write(char *file);

    /* Add the specified polygon to the structure.
       Returns 0 if successful, -1 on failure.
    */
    virtual int AddPolygon(Poly3D *poly);

    /* Add the specified texture to the structure.
       If the texture already exists, return the index of the existing texture.
       Otherwise, return the index of the new texture.
       Return -1 on failure.
    */
    int AddTexture(Texture texture);

    /* Put the polygon with index i into p.
       If no such polygon exists, return -1.
       Otherwise, return 0
```c
/*
 * GetPoly (int i, Poly3D *p);
 */
int GetPoly(int i, Poly3D *p);

/* Put the ture with index i into t.
   If no such polygon exists, return -1.
   Otherwise, return 0.
 */
int GetTexture(int i, Texture *t);

/* Return the number of polygons */
int NumPolys();

/* Return the number of textures */
int NumTextures();

/* Convert the texture to a new format, so it's more readable by export dest */
virtual void ConvertTexture(Texture *t);

/*Return the maximum height of all polygons in the table */
float GetHighestPoly();

/* Scaling factor between DXF and meters */
float MetersPerDXFUnit() { return metersPerDXFUnit; }
void SetMetersPerDXFUnit(float npdxf) {metersPerDXFUnit = npdxf; }

/* Viewpoint */
//The default viewpoint
void SetDefaultViewpoint(Viewpoint vp) { this->vp = vp; }
Viewpoint GetDefaultViewpoint() {return this->vp; }

/* Coordinate tracking -- keep track of max and min coords */
void CoordLimits(float *minx, float *miny, float *minz,
                 float *maxx, float *maxy, float *maxz)
{
    *minx = this->minx; *miny = this->miny; *minz = this->minz;
    *maxx = this->maxx; *maxy = this->maxy; *maxz = this->maxz;
}

//Keep track of the number and location of lights in the environment
int NumLights() { return numLights; }
void AddLight(Light l);
Light GetLight(int i);

private:
    void CheckHeight(Poly3D *poly);
    void CheckMinMax(Poly3D *poly);
    virtual void Idle(bool polygons /*or textures*/), int cnt, int total) {}
Poly3D **polys;
    int numPolys;
Texture *textures;
    int numTextures;
    float metersPerDXFUnit;
    float highestPoly;
    float minx, miny, minz;
    float maxx, maxy, maxz;
    Light *lights;
    int numLights;
    Viewpoint vp;
};
#endif __I03D__
```
Bibliography


