GRAPHICS SUPPORT FOR A WORLD-WIDE-WEB BASED ARCHITECTURAL DESIGN SERVICE[†]

Andrew Rau-Chaplin[‡] Brian MacKay-Lyons[*] Timmy Doucette[‡]

Jedrzej Gajewski[‡] Xiangqun Hu[‡] Peter Spierenburg[‡]

[‡] Technical University of Nova Scotia
School of Computer Science,
P.O. Box 1000,
Halifax, Nova Scotia,
Canada B3J 2X4,
Email: arc@tuns.ca

[*] Brian MacKay-Lyons
Architecture + Urban Design,
2042 Maynard Street,
Halifax, Nova Scotia,
Canada B3K 3T2

ABSTRACT

This paper describes the design and implementation of a Web-based graphical editor and visualization tool for 3D architectural forms. This Design Development Tool enables end-users to Select, Customize and Visualize house designs drawn from a large library. Having used this tool to select a base house design, the user can customize their selected design using a constraint-based graphical editor, and then view and walk through a 3D model of the resulting house.

The work described is part of the larger LaHave House Project which explores the creation of an automated architectural design service based on an industrial design approach to architecture in which Architects design families of similarly structured objects, rather than individual ones. The houses in the library have been generated in terms of a modular kit of over 1400 3D parts.

Keywords: Automated Architectural Design Service, Knowledge-based Computer Aided Architectural Design, VRML, Java.

[†] Research Partially supported by the Natural Sciences and Research Council of Canada, IRAP, BML Architectuure + Urban Design, and the Technical University of Nova Scotia

INTRODUCTION

The LaHave House project is an ongoing research project of the Faculties of Architecture and Computer Science at the Technical University of Nova Scotia, Canada. The goal of the project is to explore the creation of an **automated architectural design service** based on an industrial design approach to architecture in which Architects design families of similarly structured objects, rather than individual ones. To support our vision of an automated architectural design service, we have developed a Design Development Tool (DDT) that enables an end-user to Select, Customize and Visualize designs drawn from our design library. The design library contains symbolic descriptions of houses that have been created by a generative grammar [5,9,10] and one of the main tasks of the DDT is to transform these symbolic descriptions into 2D and 3D representation (i.e. floor plans and 3D models) that can be visualized and manipulated. Our latest prototype is an Internet-based tool written in Java, HTML and VRML [7]. This paper focuses on the many interesting issues posed in the construction of such an on-line graphical visualization and editing tool. A complete overview of the LaHave house project, with a focus on how the house designs are generated, is given in [8].

Overview of the Design Development Tool

In order to support our goal of an automated architectural design service we needed to build a user interface that was appropriate for end-users with little or no design experience. On the one hand, the tool needed to be powerful enough to give end-users the ability to create highly idiosyncratic designs, while on the other hand restrictive enough to ensure that neither the architectural nor structural integrity of the house design was compromised. Clearly the usability of the DDT hinges on striking the right balance between these two opposing goals. Our approach to date has been to combine "design by selection" [5] with a powerful, but restricted, form of user customization and to provide tools to help the user visualize the consequences of their design decisions.

The structure of our current Design Development Tool (DDT) is shown in Fig. 1. It consists of a shared user interface and three software components supporting Selection, Customization and Visualization, respectively. These software components depend in turn on 3 major libraries - a house design library, a library of 2D tiles (walls and rooms in plan view) and a 3D parts library (walls, rooms, roofs etc. as 3D geometric descriptions).

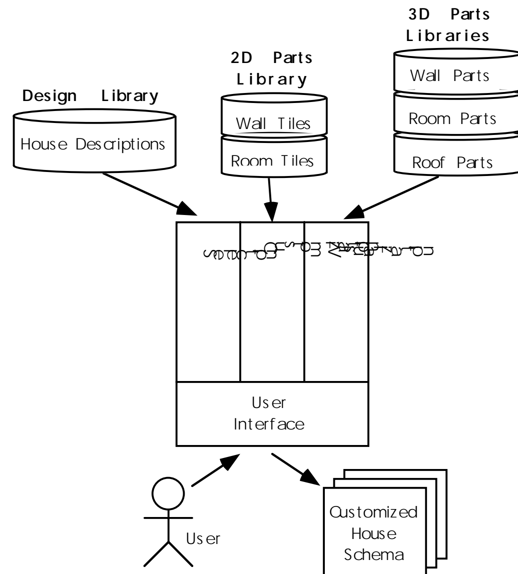

Fig. 1. Components of the LaHave house Design Development Tool

The Libraries

The *design library* is a library of "base house designs" which are used as a starting point for an end-user driven design development process. One can think of this library as the digital analog to the pattern books of the 19th century in that both contain house organizations, as much as they contain individual house designs. The design library contains houses that differ radically from each other in form, organization, size, amenity level, and "style", but despite this diversity share an underlying deep structure. This shared deep structure results from them all being productions of the same Shape Grammar [5,9,10] and is intrinsic to the system as a whole, because it is precisely this shared deep structure that allows us to create an effective design development tool. Each base house design or *house schema* consists of a symbolic description of the house's form and interior organization. The design of individual spaces in the house, for example the layout of the kitchen or bathrooms, is expressed in terms of pre-designed room arrangements, called *tiles,* which are allocated from 2D or 3D libraries, depending on the type of representation required. Our current 2D tile library consists of over 600 room tiles and 300 wall tiles, while our 3D libraries consist of over 600 room tiles, 400 wall tiles, and 300 other tiles representing roof parts etc.

The Selection Component

The task of the Selection component is to help a user select one of the "base house designs" in the library from which to start developing their own design. First, the user completes a "questionnaire" concerning their housing needs and dream (Figure B). Suitable bases house designs are then selected from the design library for users to browse (Figure C). Associated with each base house design is a "digital brochure" consisting of 2D floor plans, 3D still images and summary information (i.e. square footage, # of bathrooms etc.) in the form of a typical "real estate cut sheet" (Figure D).

By a process of examination and elimination the user is expected to decide on a single base house design which best satisfies their needs/dreams. This design then becomes the starting point for customization.

The Customization Component

The task of the Customization component is to allow the user to modify/evolve their base house design into one that more fully meets their needs. The current Customization component is a graphical 2D constraint-based editor. This editor allows users to replace components of a house design with new components under a set of functional and spatial constraints (See Figure E). We briefly explored the idea of creating a 3D customization tool, but it quickly became apparent that neither our target users nor our target implementation platform (VRML) were up to the task of interactive 3D editing.

The Visualization Component

At any point while browsing or during customization the user may need to visualize their current house design in 3D in order to really understand it. The task of the Visualization component is to transform a house schema into a complete 3D model. The Visualization component consists of two programs. The first is a compiler that parses a house schema and, using a 3D kit of pre-manufactured parts/models (analogous to the 2D tile library), constructs on-the-fly a complete 3D model of the house. The second program is a 3D browser that allows an end-user to view and "walk-through" the completed 3D model. In our current prototype, the user requires locally only a Java/VRML enabled browser (Figure E) as the rest of the system resides on a UNIX server that is reached by the users browser over the Internet (Fig. 3).

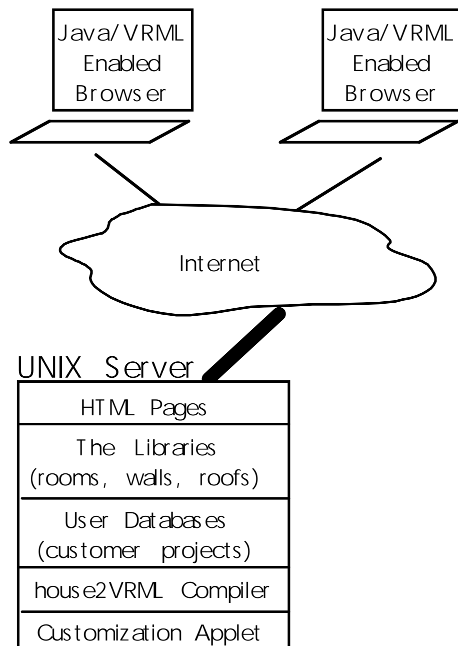

Fig. 3. Software model for the Design Development Tool

Figures G and H illustrate the level of 3D presentation we currently achieve. The two photographs are of houses commissioned by clients of the project's principal Architect [3,4], while the six computer generated images show houses drawn from the design library (i.e. generated by shape grammars) and assembled and displayed by the Visualization component. Note, our focus to date has been on assembling a correct 3D model, rather than on beautiful rendering. A much more detailed description of the Visualization component is given in the next section.

3D VISUALIZATION OF HOUSES IN VRML

Our initial implementation of the Visualization component was prototyped in a programmable CAD package with basic rendering features. This allowed us to explore the challenges of on-the-fly 3D model construction in an environment rich in primitive geometric operations, but was much too slow and required each user have a copy of an expensive CAD package. In deciding to develop a network based design development tool we felt we had to address a number of technical and design challenges.

1) **Ease of use**: Understanding and working with 3D models can be very difficult. We needed to find ways to help our target users, who have no architectural or design experience, to understand the real 3D space being modeled and to navigate easily inside and around their house design.

2) **Speed**: In our context, the 3D models of houses can't be precomputed since the user may frequently need to visualize their current design as they are customizing it. Therefore, the Visualization component must be able to construct on-the-fly a 3D model, download the model to the users machine, render it, and provide fluid walk-through with an elapsed time of only a few minutes.

3) **Scalability**: Our house schema currently describe houses in terms of a set of over 1400 3D parts called tiles. Clearly, the more parts in the library the richer the space of possible house designs, but as the number of parts grows so does the burden of maintaining, processing and downloading them. Everything possible must be done to ensure that, 1) the parts library scales gracefully (i.e. new parts can be added without restructuring the libraries) and 2) as far as possible, only truly different/new parts are added to the library, not just parts that are simple modification of existing parts.

4) **Level of Abstraction**: The Visualization component must strike the right balance between abstract and realistic presentation. If the model is too abstract the user may not be able to mentally visualize the finished product, while if it is too realistic the user may be fooled into making fundamental design decisions based on features, like paint colour, which are only surface deep.

Why VRML?

VRML is a platform-independent 3D modeling format [7,1], based on the SGI's Open Inventor format, which is evolving into the standard for on-line Web-based modeling. We choose to implement the 3D Visualization component of our Design Development Tool in VRML 1 .0 for a variety of reasons including cost (free), easy integration with other Web-based systems, reasonably wide availability of viewers, and platform independence. Choosing to work with Version 1.0 of any software is always a risk, especially when that software is evolving as quickly as VRML is. The VRML viewers we have had to work with produce reasonably crude images, lack many important features and are far from bug free. Despite these problems we are satisfied with our choice of VRML and believe it is rapidly becoming a good platform for supporting on-line user-driven walk-throughs.

Our Approach

In order to address the primary challenges we identified, especially those of Speed and Scalability, we decided that a high degree of parts reuse was essential in constructing the 3D models of our houses. In deciding on the granularity of parts it was important to strike the right balance. A fine granularity would mean fewer parts to maintain (i.e. fewer scalablity problems) and more compact models (i.e. faster download times), but would increase the complexity of the model assembly problem and make the model more complex to render. A coarse granularity would greatly simplify the model assembly problem, but would increase the file size (thereby increasing download time) and the number of parts required for a model, thereby making the problem of parts creation and maintenance difficult to solve.

In the end we decided to create two basic types of parts: parts used to form the external shell of the houses (e.g. walls, floors and roof parts) and part used to populate the interiors of the houses (e.g. cabinets, stairs, and hearths). The parts used to form the exterior shell were designed in segments which would then be pieced together to form different houses. For example, with reference to Fig. 4, a simple eight by six foot shed with a door on the Southeast side might consist of 6 wall parts: 3 four foot sections (a,b,e), 2 six foot sections (c,f), and a four foot section with a door (d). Also a floor and 4 roof parts: 2 gable roof sections, and 2 gable ends. A window might be added to the same shed by replacing one of the four foot wall sections (e) with another four foot wall section that includes a window (g).

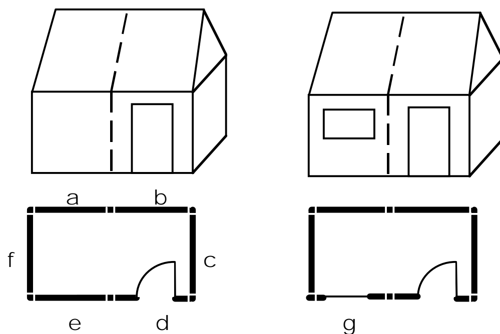

Fig. 4. A sample house assembly

Several strategies were implemented to maximize the reuse of parts. These include, keeping the parts symmetric where possible so that the same part might serve in multiple situations, and designing walls so that they can be connected to other walls at either 90°, 180° or 270° degrees without change.

Part Design

A part whose geometry is symmetric about either the x or y axis can typically serve in more orientations than an asymmetric part. For example, wall segments which do not differentiate between the inside and outside surface treatment can be used on any side of a house by simply translating the part. In designing the parts library we have attempted to design symmetric parts where possible and to use differences in materiality (rather than geometry) to distinguish different surfaces.

The parts needed to be designed and created carefully in order to ensure that the composite house models could be assembled both easily and accurately. In particular, since many of the parts are interchangeable consistence is critical. A 4-foot wall without a window and a 4-foot wall with a window have to fit precisely in the same location depending on which has been specified. When we started there were no tools specifically designed for creating VRML objects, so all of the parts were created in a standard CAD program (Autodesk's Autocad). The resulting files were then systematically converted into VRML files.

One of the early decision made in designing the wall library was to create distinct parts for East to West running walls (longitudinal) vs. North to South running walls (transverse). The parts are spatially oriented in a longitudinal and transverse orientation when they are defined. When it is time to use them, no rotations are necessary. It may be to our advantage in the future to have all the wall objects oriented in a single orientation and then rotate them as needed. This will reduce the number of definitions, thus reducing the file size, but tended to create significantly longer rendering times in the viewers we initially started working with. Clearly when considering speed vs. complexity tradeoffs the machine that is being used will typically determine which approach is better. Currently, our VRML house file sizes and download times are very acceptable and rendering speed is the critical factor, especially on older machines such as 486's.

In addition, the wall pieces do not include all the door-swing combinations, as there would be far too many distinct objects that way. Instead, doors are added to the model separately, at the time of assembly, based on door swing information given in the house schema files.

Part Creation

The parts created in Autocad and saved in DXF format needed to be converted to VRML. At the time (and perhaps still) there were no direct DXF to VRML translators that could handle DXF files created using Autocad's 3D modeling extension, so we had to take an intermediate route. The DXF files are first converted to

3D Studio files (3DS) using Autodesk's dxf23ds program and then to VRML (.wrl) using a program called Interchange (by Sydesis Software Inc.).

The conversion programs add a great deal of extraneous information to the converted files such as header information, and a somewhat random collection of lights and cameras. As we needed to define these separately, a series of programs that strip out all the unwanted information were written. The final parts files contain only object definition separators, Coordinate3 nodes that define the points of each object, IndexFaceSets that defines the polygons on the points and MaterialIndex that indexes faces to a color palette.

Each part file consists of several objects. In the following example, a wall, called clfd1f, is made from a wall (WALL01), a window sill (SILL01), the window's glass (GLASS01), and a door frame (FRAME01).

```
DEF clfd1f Separator {

 Separator { # Object 'WALL01'
   Coordinate3 { point [ 0 0 6, ….]}
   IndexedFaceSet {
     coordIndex [ 0, 2, 1, -1, …..]
      materialIndex [0, ….]
   } # IndexedFaceSet
 } # end of object 'WALL01'

 Separator { # Object 'FRAME01'
   Coordinate3 { point [116 106 5, …..  ] }
   IndexedFaceSet {
     coordIndex [0, 2, 1, -1,…. ]
       materialIndex [ 9, 9, …..]
    } # IndexedFaceSet
 } # end of object 'FRAME01'

 Separator { # Object 'SILL01'
   Coordinate3 { point [ 42 14 5,…] }
   IndexedFaceSet {
     coordIndex [0, 1, 2, -1,…..]
     materialIndex [8, 8, …..]
   } # IndexedFaceSet
 } # end of object 'SILL01'

 Separator { # Object 'GLASS01'
   Coordinate3 { point [ 150 108 4,….]}
   IndexedFaceSet {
     coordIndex [0, 1, 2, …….]
     materialIndex [ 6, 6…..]
   } # IndexedFaceSet
 } # end of object 'GLASS01'

} # end of clfd1f
```

The last part of our conversion process was to redefine the material indices for each object in each file. Initially, all faces of each object, for example WALL01 were assigned by the conversion process the same material index. As we needed to assign different materials to, for example, the interior side of a wall as opposed to the exterior side, it was necessary to assign to faces on different planar surfaces different material indices. A program was written that walks across the face sets identifying faces that define the same side (plane) of each object. This allows us to change the material of each side of the object by just changing which material palette is refered to. See the section on `Coloring and the Material Palettes' for more details.

'Inline' vs. 'Define and Use'

There are two quite different ways in VRML to reuse parts. In the first, each object is defined in a separate file and then called by a WWW Inline Node in the main house file. This can have advantages, especially when combined with level of detail nodes, but was rejected in our case due to the fact that every time a piece was to be used, the viewer had to download the piece from the server. This resulted in a lot of unnecessary communication over the Internet and slowed down the overall performance of our application. The second method is to define the objects in the file using DEF Nodes, which associates a name with an object. Under normal circumstances when an object is defined, it is also displayed, but the appropriate use of a SWITCH node allows one to define, but not display an object. The defined objects can then be used as many times as required through the USE Node. Although this approach resulted in larger file sizes, the total upload and render time was greatly reduced.

Overlapping of Parts

The houses to be visualized by the Design Development Tool are to be physically realized using a modular building system called "Smart Sticks". This system is somewhat analogous to a traditional post and beam construction system. Each "wall" piece sits between two post and on top of a beam. In our initial implementation of the Visualization component we modeled each post and beam. This turned out to be prohibitively expensive in term of the number of parts needed to define even a reasonably simple structure, also this type of model presented coloring problems as it was difficult to compute what material to assign to each side of each post to ensure that the post and beam type structure was not visible.

To reduce the number of parts in the models, we decided to combine the supporting structure (post and beams) with the wall parts. The wall libraries were redesigned such that each wall included posts on both sides and a beam underneath. This reduced the number of polygons in our house files and brought the performance back up to acceptable levels. The coloring problem was also solved as we just had to color a wall and not worry about the beams and posts.

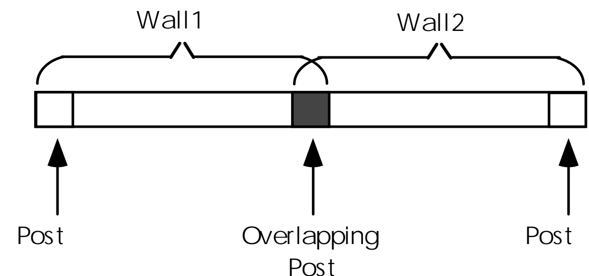

Fig. 5. Assembly of wall parts

This approach did however have its own complexities. Since each wall segment includes the posts on either side, when two wall objects were placed side by side or at right angles to each other, the posts will overlap each other. The overlapping of post means that only a single set of walls needs to be created as they can be joined together at either 90°, 180° or 270° degrees without change, but has a tendency to produce visual aberrations (particularly seaming effects) in many browsers.

On-the-Fly Generation of 3D Models

The creation of a house model involves the assembly of the selected parts in the correct positions/orientations, the definition of default viewpoints, and the addition of appropriate light sources. When the user follows a “visualize link” in the Design Development Tool, the current house scheme is passed to the house2VRML compiler which generates the 3D model. Basically, a house schema defines for each part to be used its position, orientation and an appropriate materials palette. The compiler works in two passes. It makes a first pass through the input file to identify and then define all of the parts and palettes that will be need. This is done so that each part or pallet is defined only once, rather than every time that it is referenced. Once the definitions are complete, a second pass over the input file is made in which individual parts are assembled into a complete model by reference to the earlier definitions. When this is complete, the model is passed to the user’s VRML enabled broswer to be displayed.

Placing the Parts. The positions of each object in a house schema file is defined in terms of a floor number and x and y coordinates.

House Schema	*VRML*
'id' '103'	Separator {
'l' 450 348	Translation {
'wc' 'clfd1f'	translation 653 2 550}
't' 'esie'	USE esie
'asu' '9' ' '	USE clfd1f }

These coordinates are converted to the x, y and z coordinate system of VRML. The ‘l’, ‘wc’ and ‘t’ in the house schema file stand for location, wall class and type (material palette definition), respectively. The other fields are used by the Customization Applet and Design Engine programs.

Coloring and the Material Palettes. As described earlier, the face sets that define each object are grouped into subsets of faces that define planar surfaces of the object. Each such subset is mapped into a different location in a material palette. This allows us to change the material of each side (planar surface) of an object by just changing which material palette we are using [1]. Using the same example as before, the part of the definition section that indexes the materials for the window sill object of the *clfd1f* wall would look like the following.

```
Separator { # Object 'SILL01'
  Coordinate3 { point [42 14 5,….] }
  IndexedFaceSet {
    ccoordIndex [ 0, 1, 2, -1,….]
    materialIndex [8, 8, 8, 8, 8, 8, 8, 8, 8,
                   8, 8, 8]
  }
}
```

In this case, all the sides of the window sill object are being indexed to the same entry in our palette, #8. This will result in the window sill being the same color no matter what angle you look from. Here is a sample of the *esie* palette, where it is being referenced.

```
  DEF esie Material {
     diffuseColor [
        1.0 1.0 1.0,  #normal
        0.584 0.505 0.384,  #exterior
        1.0 1.0 1.0,  #interior
        0.0 0.5 1.0,  #wall
        0.0 0.0 0.0,  #glass
        0.8431 0.7215 0.4901,  #door
        0.8431 0.7215 0.4901,  #sill
        0.8431 0.7215 0.4901,  #frame
        1.0 0.2 0.0,  #Roof
        ]
     transparency [
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.25
        0.0, 0.0, 0.0, 0.0, ]
  }
```

The transparency section of the definition corresponds to the defined materials. Here, the 7th entry in transparency refers to the 7th row in the materials definition. This results in the glass material being 25% transparent. The transparency attribute adds significant realism of the models, as it allows the viewer to see some of the inside of the house.

Generation of Viewpoints. We found from experience that the most useful thing we could do for a user, in terms of making navigation around and inside the models manageable, was to predefine important viewpoints. This is very consistent with the idea of an automated architectural service in that it allows us to suggest to the user what positions and angles it is important to look at their design from. Defining appropriate viewpoints requires both knowledge of the geometry of the house and knowledge of its functional organization. For example, if a house contains a kitchen with a large kitchen island that looks out on the dining room it may be important to define a viewpoint at head height looking out over the island into the dining room to help a user experience what it might be like to live in the house. Currently we define about 10 preset views so that the user can navigate through the 3D model by moving from one view point to another.

Lighting and Shading. Control of lighting, shading and rendering quality is perhaps where current VRML technology lags farthest behind other 3D graphical environments. Although, VRML supports Material nodes that include diffuse, emissive, ambient, and specula color values, there appears little point in trying achieve subtle effects. Variations between browsers, colour palettes, and monitors from machine to machine conspire to make a model that looks good in one setting into an eye sore in another.

Most VRML viewer support two types of shading: smooth and flat. Unfortunately, due to the way we construct our models, as a composite of parts, smooth shading results in many undesirable visual effects by highlighting the interfaces between the parts. Flat shading produces very abstract looking models, which although not bad looking, have a kind of “dark side of the moon” feeling to them which is intensified by the lack of shadows. Although we are still experimenting with ways to achieve more realistic lighting we have for the moment decided to depend on the “headlight” provided by most VRML browsers as our main light source.

Optimizations. There are many ways to optimize our models in VRML that can decrease file sizes, reduce rendering time and increase the overall performance. It is

important to us to pursue these optimizations simply because when the models are optimized, one can add more detail to them, making them visually richer, while maintaining the same performance. Our primary optimization, the reuse of parts in different conditions has already been discussed. We have also implemented several other simple optimization [1,6]. Unnecessary spaces, tabs and decimal digits take up space in a VRML file and have been removed. Untruncated decimal digits require the rendering routines to perform more work often at no visible benefit. The use of ShapeHint nodes allows browsers that implement backface culling to reduce drastically the number of polygons that have to be rendered.

Perhaps the most important optimization turns out to be file compression. The recent introduction of automatic compression and decompression by many VRML viewers has drastically increased the size of models that can be reasonably worked with. In our experience, if the final models are gzipped it results in an over 90% reduction in file size and an equally large reduction in download times. File compression make such a difference that we are considering moving from a composite model for the main shell of the houses (i.e. where the shell consists of a collection of parts) to a integral model, which although much bigger is easier to render well.

CONCLUSIONS AND FUTURE WORK

Our current design development tool is very much a prototype; an environment in which to explore design development by watching an end-user at work. After constructing a new prototype in HTML, Java and VRML, we are in the process of enhancing its basic features and running a usability test to evaluate its design and performance.

For network based 3D visualization VRML has proved to be a very promising environment. Our current prototype provides enough visual detail to be very functional as a design tool and its performance over a 28.8 modem is good. VRML viewer technology is improving, but there still much room for further improvements in such areas as shading, colour consistence, and navigation controls. The issue of file size is no longer as critical an issue as before given the new compression facilities. We feel we can now comfortably increase our uncompressed file size to around 500K (i.e. compressed files of approximately 50-60K) and still achieve reasonable download times. One way in which we are using this new space budget is by modifying our house2VRML compiler to build an integral description of the shell of each house, rather than a strictly parts based one. This should significantly improve the quality of the 3D rendering, but requires almost twice the space. The issue of rendering speed is still remains. Given a “typical” users machine, for example a 486/50 with 16MB RAM, we find models consisting of between 8000 and 12,000 polygons perform reasonably, while performance on larger models rapidly degrades.

Many interesting open questions concerning 3D visualization in our context remain. In particular we are interested in the architectural presentation issues of how best to present 3D representations of houses to end-users. We are exploring how abstract 3D models, that convey the underlying structure of LaHave house designs, might be used to better “reveal” the design to users.

Acknowledgments

We would like to thank the TUNS Faculty of Architecture and School of Computer Science for their support and acknowledge the work of the following individuals: *Architects:* J. Smirnis, D. Wigle, E. Jannasech, N. Savagen, P. McClelland. *Computer Scientists:* H. Ning, G. Li, D. Gemmell, D. Curry, D. Peters, G. Burrell, H. Lui, S. Gauvin, W. Wu

References

[1] A. Ames, D. Nadeau, and J. Moreland, "The VRML Sourcebook". John Wiley & Sons, 1996.

[2] M. Friedell and S. Kochhar, "Design and Modeling with Schema Grammars", Journal of Visual Languages and Computing, 1991, Vol. 2 , pp. 247-273.

[3] T. Fisher, "Folk Tech", Progressive Architecture, August 1995, pp. 63-72.

[4] B. MacKay-Lyons, "The Village Architect". Design Quarterly 165, MIT Press, Editor R. Jensen, Summer 1996.

[5] W. Mitchell, "The Logic of Architecture". MIT Press, 1990.

[6] D. Nadeau, A. Ames, and J. Moreland, "Optimizing the performance of VRML worlds", Dr. Dobb’s Journal, 1992, No. 249, pp. 16-24, July 1996.

[7] M. Pesce, "VRML: Browsing and Building Cyberspace", New Riders Publishing, 1995.

[8] A. Rau-Chaplin, B. MacKay-Lyons, P. Spierenburg, “The LaHave House Project: Towards an Automated Architectural Design Service”, Proc. of International Conference on Computer-Aided Design (CADEX’96), Sept. 1996.

[9] B. Silverman, "Survey of Expert Critiquing Systems: Practical and Theoretical Frontiers", Comm. of the ACM, 1992, Vol. 35, No. 4, pp. 107-127.

[10] G. Stiny and W. Mitchell, "The Palladian Grammar", Environment and Planning B: Planning and Design 5, 1978, pp. 5-18.

[11] G. Stiny, "Introduction to Shape Grammars", Environment and Planning B: Planning and Design 7, 1980, pp. 343-351.