Besting the Tract Home: A Software-Based Bricolage Approach to Affordable Custom Housing

by

Thomas Clayton Plewe

B.A., Computer Science (2002)
Pomona College

Submitted to the Department Of Architecture in Partial Fulfillment of the Requirements for the Degree of
Master of Science In Architecture Studies

at the

Massachusetts Institute of Technology

June 2008

Signature of Author .....
Department of Architecture
May 17, 2008

Certified by .....
Terry Knight
Professor of Design and Computation
Thesis Supervisor

Accepted by .....
Julian Beinart
Professor of Architecture
Chair of the Department Committee on Graduate Students

Terry Knight, Professor of Design and Computation , Thesis Advisor

James Stockard, Lecturer in Housing Studies, Harvard Graduate School of Design, Thesis Co-Advisor

Takehiko Nagakura, Associate Professor of Design and Computation, Reader

Besting the Tract Home: A Software-Based Bricolage Approach to Affordable Custom Housing

by

Thomas Clayton Plewe

Submitted to the Department of Architecture on May 22, 2008 in Partial Fulfillment of the Requirements for the Degree of Master of Science In Architecture Studies

ABSTRACT

Tract housing has earned its position as the overwhelmingly dominant paradigm of home building and ownership in America because it's such an efficient and therefore cost-effective system. Custom-designed housing has provable benefits over a one-size-fits-all approach, but has remained unreachable for the vast majority of home buyers (or at the very least not worth it) due to the price/time/hassle disadvantage of its inefficient production systems.

In attempting to make customized housing competitive with the tract home on a price/time/hassle graph, this thesis searches for efficiency through using bricolage; nonstandard, ambiguous components; the principles of object-oriented programming; and the consumer-centric standard practices of e-commerce. A paradigm and accompanying software are created to allow a custom house to be designed in hours rather than months, enabling architects to design by arranging pre-designed multi-room components, as selected from a searchable database, into a single structure that uniquely fits a client's needs. Sample houses are designed and economic estimates are made to gauge the potential competitiveness of such a system with tract housing, as well as the system's potential effect on the overall economy of architecture.

Thesis Advisor: Terry Knight
Title: Professor of Design and Computation

ACKNOWLEDGEMENTS

To Terry Knight for happily allowing me to defy some significant tenets of architecture.

To Jim Stockard for selflessly helping and encouraging this project and being the consummate teacher.

To Takehiko Nagakura for helpful insights into my various programming projects.

To Rick Mclain for going a year ahead.

To Simon Kim for going 3 months behind.

To Mariana Ibanez, for making sure Simon goes at all.

To John Plewe for not going at all, at least not yet.

To Jenni Nelson for letting me go at all.

To Steven Plewe for making many concrete sculptures.

To Janet Plewe for remaking a concrete condo.

To all the other Plewes and their add-ons for being involved with concrete in a variety of productive ways.

To David Randall and Adam Cobabe, because I don't think I got either of them a Christmas or birthday gift.

To all my people from Claremont, blessed be the name.

To all my people from Boston and New York, blessed be your place names as well, on a linear scale which increases the further one progresses West.

Besting the Tract Home: A Software-Based Bricolage Approach to Affordable Custom Housing

by

Thomas Clayton Plewe

INTRODUCTION

Famed houser Catherine Bauer observed that "what is primarily needed, not only for low-income slum dwellers and minority groups, but for the great mass of middle-income families in all their infinite variety of taste and need, is more choice in location, dwelling type, and neighborhood character" [1]. This particular observation came in 1957, well before any of the housing developments of the 70s, 80s, and 90s that most now associate with bland tract housing. While advances in housing options have increased the relative diversity of choices since that time for certain segments, the statement resonates at least as loudly to those now pondering modern suburban sprawl as it did to Bauer in 1957.

Much of the bitter aftertaste inherent in the term "suburban sprawl" could be argued to come not from the fact of sprawling-ness or sub-urban-ness, but rather from the same-ness that has defined suburban building for so much of its history. Architects often think of this sameness in terms of design blandness, the tediousness of the eye passing over the same pattern over and over again, or perhaps in terms of individuals losing their identity as they quietly disappear into some unidentifiable corner of the greater mass. If each new housing development that popped up offered a varied and stimulating space to walk through, architects and citizens alike would surely be less apprehensive and perhaps offer a more upbeat term than "sprawl", even if the technical sprawling-ness remained constant. While environmental impact and resource distribution are serious problems, the entire population of the world (approximated a 6 billion) could stand in the state of Utah with each person standing 20 feet from the nearest person.

Figure 1. A tract housing development in San Jose, CA.. Image by Sean O'Flaherty

While sameness is bothersome to many of the design-conscious, it has perhaps more measurably insidious implications from a economic perspective. It means not only blandness, but also fewer choices. Fewer choices means inherent inefficiency, because when someone has to buy a house that has an extra bathroom they don't use or exterior detailing they don't particularly like, they necessarily sacrifice some other asset that could have benefited them more, e.g. better windows or more living room floor space. Everyone's resources (i.e. dollars) are limited, so a lack of customizability necessarily stops resources from being used in the best way possible.

When the American-dreaming family is considering purchasing a new home, the majority of their options lie either in large condo complexes or 3-4 bedroom houses in tract housing developments consisting of a few very-similar models to choose among. Tract housing developers limit their house options for good reason: aiming all of their homes at the median family, a 2-parent household with 2-3 children, they target the largest segment of the market while roughly fitting those who are off the median in one direction or another. Having only a few models to build saves the developer significant money in design and construction costs over a system where everyone has a custom designed home. However, while a 2-parent 3-child household may be the median new home purchaser, most households vary from that median in some meaningful way, and thus most have an inefficient fit with their tract home. Figure 2 illustrates this phenomenon, with the bell curve representing the distribution of family types in America (where the average family makeup sits in the very middle) and the vertical bar illustrating the range where nearly all new tract housing is targeted, leaving most families with a less-than-perfect fit.

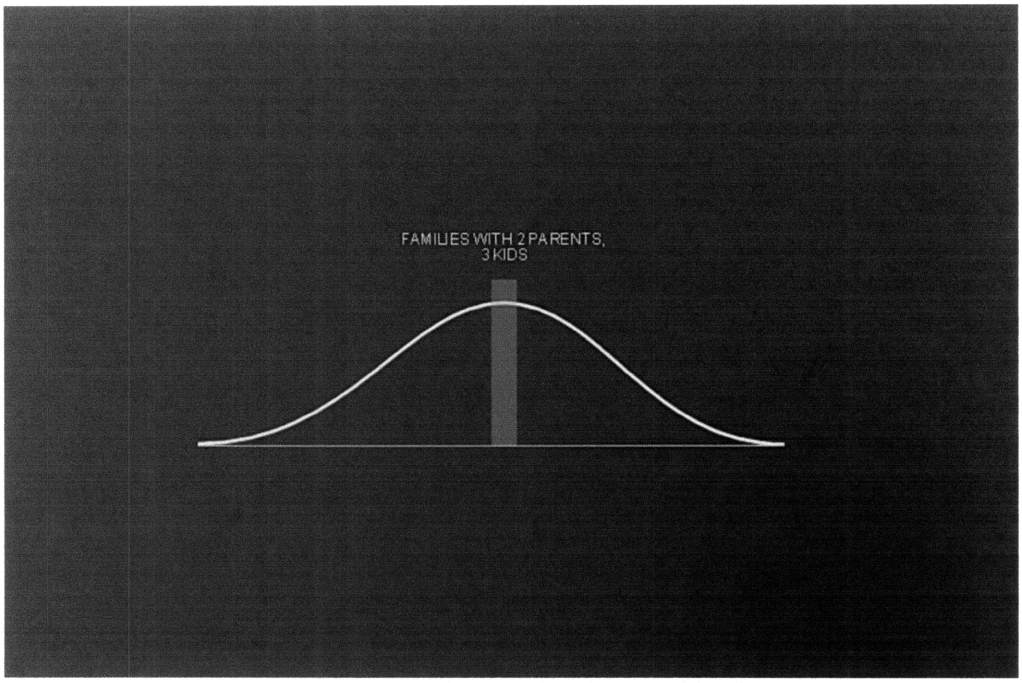


Figure 2. Graph showing distribution of family types and housing built.

On the other end of the scale, fully-custom-designed housing is prohibitively expensive for almost everybody. Only if they are very wealthy can a family consider building a house with real features and spaces customized to their needs, spending countless hours and 15% of their home's purchase price to enter a close relationship with an architect eager to maximize the size of that 15%. Older homes are an option that brings more variation, but since their supply is capped by definition and the need for more housing is growing rapidly, this thesis will focus only on new construction. When looking among these options for a new home, what happens to the non-wealthy artist couple with a young child, who want a house with a very large studio space and a modern living room but need the ability to sacrifice expensive finishes and bedroom floor space to afford the home at all? What about the party-throwing middle-class empty-nesters with a live-in parent who need a relatively small home that has two kitchens? What about the 2-parent 4-child family who can't afford the tract home where the living room and garage have been scaled up in size and quality to match the increase in expected value that generally comes with their needed additional bedroom?

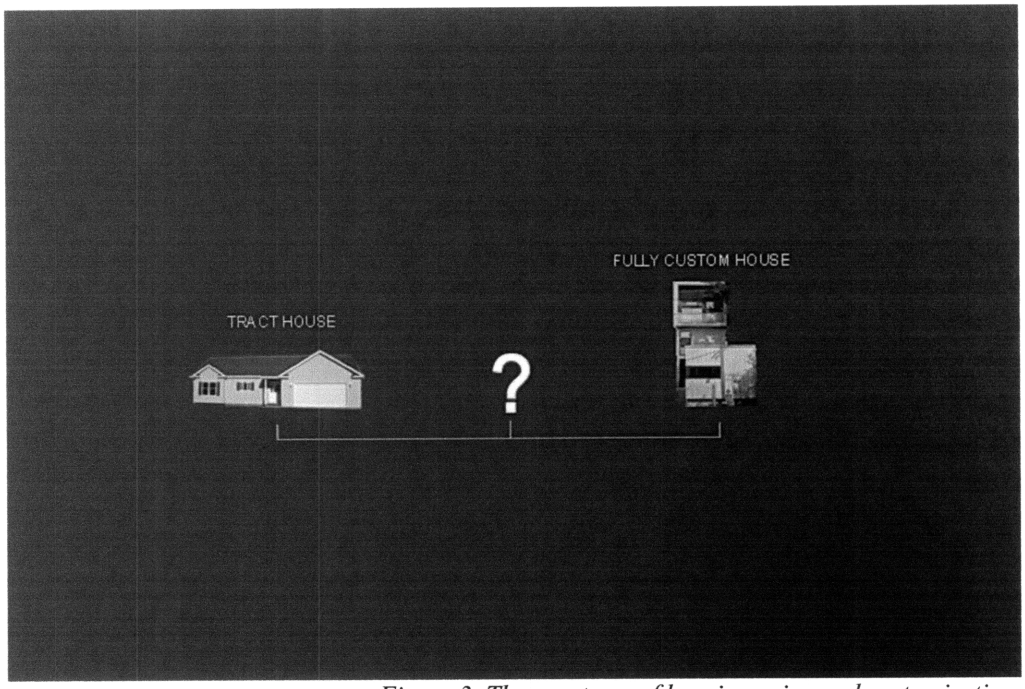


Figure 3. The spectrum of housing price and customization.

With a spectrum established, with tract housing on one end of the affordability/customizability axis and fully-custom-designed housing on the other, what exists in the middle? Sadly, not much in terms of what actually gets built, and that middle zone is the space that the software and paradigm created in this thesis intend to target. However, to best and therefore replace the tract home, a solution can't simply fall halfway between custom and tract housing on both affordability and customization; it has to be able to reach the same economic ballpark as the familiar tract housing system while offering considerable gains in customization.

Can such a solution exist? This thesis asserts that it can, that through creating and using an object-oriented bricolage design paradigm and software, custom-designed housing can become a viable alternative to tract housing for mainstream America.

H++ OVERVIEW

The search for a solution that brings about mass customized housing is certainly not new. As technology has advanced, the realization of this goal has seemed more and more within our reach even though tract housing continues its dominance. Many attempts at inexpensive customization of domestic architecture have focused on manufacturing techniques, from Walter Gropius and Konrad Wachsmann's factory-made house[2] of the 1940s to the Dwell/Empyrean pre-manufactured homes of this century[3]. Kent Larson and the house_n lab at MIT have been working on an "open-source" standard that would allow physical components from different manufacturers to connect and communicate with each other[4]. Whether due to lack of economic scale, the difficulty of coordinating separate players in the industry, or public perception of pre-manufacturing, none of these solutions have yet mounted a significant challenge to traditional tract housing (i.e. traditional tract housing continues to dominate new construction).

Attempted solutions for viable mass customized housing have also targeted the design stage of the process. Previous MIT theses include José Duarte's PhD thesis on creating a shape grammar to generate custom house designs based on those of architect Álvaro Siza, as seen in his houses at Malagueira, Portugal[5]. The Malagueira development itself was an attempt at large-scale, affordable custom housing, though Siza generated all of the designs himself. Also from MIT, in association with Kent Larson, came Giles Phillips master's thesis, which proposed a method of design that was based on users utilizing a Google-like search engine to participate in the design process[6]. Users of this system could help create custom home designs based on a component housing model developed at the house_n lab. Perhaps not enough time has passed between the creation of these methods and the present for something like them to have taken hold, but still the fact remains that nothing has yet gained real traction in taking on the tract home in terms of real-world implementation.

The solution proposed in this thesis is part of the latter of the two solution types just described, focusing on the design tasks of home building rather than the construction tasks. The paradigm and accompanying software that have been created are collectively named H++, a reference to housing (hence the "H") and the C++ programming language, whose design and reasoning as an object-oriented programming language inspired important elements of the paradigm. H++ embraces three important ideas in its attempt at challenging the tract home for dominance in American building: 1) divide and diffuse the design process in an object-oriented manner, so that design work can be re-used as much as possible; 2) focus on design, specifically digital design, as opposed to construction, in order to be able to apply the various signifiers of modern e-commerce; and 3) embrace bricolage design, a

visual and functional collage of distinct identities, which we've come to embrace readily on a city level but much less readily on a single-structure level.

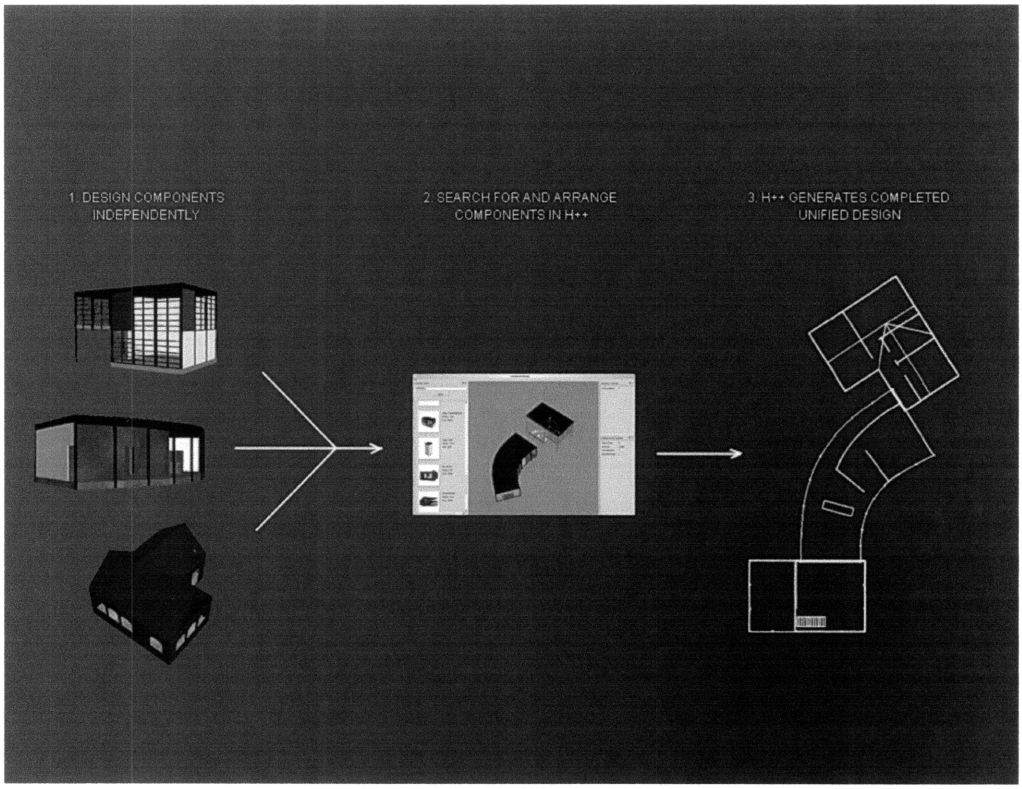


Figure 4. H++ wokflow diagram.

Designing a house using H++ happens in two distinct phases (see Figure 4). In the first phase, multi-room stand-alone components are designed by individual architects. One such component might include three bedrooms and a bathroom, another might include a master suite and a living area, another might include a garage and a kitchen, and another might include nothing more distinct than a public area and a private area. There is no clear definition for what a component includes, any combination of spaces may be useful in some given situation, as will be described further on. But each component is fully designed by its architect, including all details possible, and is uploaded into a common database as a standardized digital file. This database could contain hundreds or thousands of different components designed by hundreds or thousands of different architects. In the prototype software created for this thesis the database is stored locally on a single computer, but in a future version it could be centrally located online and accessed by each individual copy of the H++ software over the internet. Note that the component designing, as implemented in this thesis, happens outside of the H++ software, using standard 3D CAD software.

The second phase of house designing with H++ happens within the H++ software, and relies on the database just described being in place and populated with component designs. When an architect gets a new client, he talks with them to determine their needs and then starts envisioning their future house as being made up of two to four (or more or fewer, depending on the situation) of the components as described above, which may well be designed by architects other than himself. Say, for example, the client is a young artist couple with a small child, who want a house with a smallish but modern living and kitchen area, a nice master bedroom and bath, and a very large studio space whose quality is less important than its size. Of course they'd love another few bedrooms, but they can't afford them at the moment if they want their large studio space. The architect working with them might group in his mind as one component the living, kitchen, bedroom, and bath areas, because what the couple is looking for in each of those areas is similar in relative scale and quality, and there's likely to be a component that includes those areas together. Then in his mind he would think of the studio as its own component, or maybe a studio with a small utility room attached – these spaces stand apart from the rest of the house conceptually as their own component because their size is much more important to the client than their finish. The architect then turns to H++ and uses the interface to search for components that match his conception, loading the appropriate ones into a 3D workspace. In the 3D workspace he arranges and combines these few components into a single house, with the software helping him to merge exterior walls of neighboring components into each other to connect the spaces and unify the structure. Perhaps the clients want him to add a third component with extra bedrooms into the design, but they mark it as a future

addition they can build when they have the money. The software takes care of the details automatically, and outputs the final unified design based on the individual component designs as created by each component's original architect.

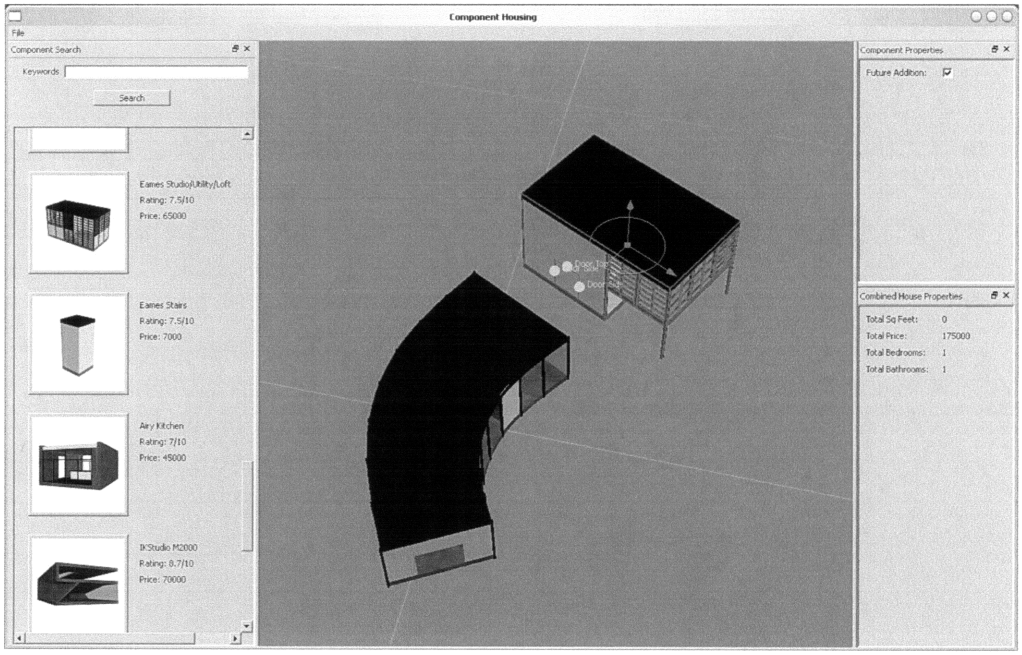


Figure 5. Screenshot of the H++ software in use.

This second phase may require several iterations, but it is entirely possible that it could happen over the course of an afternoon, especially if the client has researched components online beforehand and have a good idea of what they want. Since each component gets used many times in many different situations, its design costs get dispersed and diminished for each individual client, and each client only ends up spending a few hours interacting with an architect. The architect arranging the components gets a flat fee, and each architect whose component gets used gets a flat fee. In dramatically less time than the standard custom housing design system, the client can walk away with the designs for a completely unique house that has a significant level of customization as compared with a standard tract home, at a fraction of the usual cost of having an architect make a complete custom design.

This process, its benefits, and how it can challenge the tract home as the dominant housing paradigm in America will be discussed below by looking individually at each of the three guiding principles mentioned previously. Restated for convenience, they are 1) divide and diffuse the design process in an object-oriented manner, 2) focus on design, specifically digital design, and 3) embrace a bricolage aesthetic.

DIVIDING AND DIFFUSING THE DESIGN PROCESS

Most modern programming languages, including the ubiquitous C++, are object-oriented languages. Object-oriented programming espouses encapsulation of functionality, or in other words breaking down a complex system into component parts, where each component only has to worry about its own internal details and doesn't care how the rest of the broader system might work[7]. Consider, for example, a 3D modeling program written in C++. Such a program might have the following components, referred to in C++ as "classes": a vertex class, a triangle class, and a shape class. The vertex class contains as its data 3 numbers that represent a given vertex's position in 3D space. The class also defines several functions that one might want to perform on a vertex, for example a function that tells you the vertex's position in 3D space, or a function that rotates a vertex around some axis by a certain number of degrees. The triangle class then contains as its data 3 of these vertices – note that the triangle doesn't care how the vertex functions work internally, it just needs to be able to call certain functions on its three vertices and trust that they're implemented correctly within the vertex class. The shape class may then contain a list of triangles that make up the 3D shape; a shape doesn't care how the triangles work, it may not even know that they contain vertices, so long as it can call certain necessary functions defined in the triangle class (such as a function to draw the triangle on the screen, which the shape will need to use when its own drawing function is called).

Good C++ programs follow the mantra of "private data, public functions." That is, each class keeps its data and internal workings hidden from other classes, exposed only through certain specific functions that other classes can call. The more a given class follows this principle, the easier it is to use it within different contexts, and the more likely it is to work properly within a large system because all of its important details are isolated and protected.

The C++ class is the main inspiration for the H++ housing component. The data structures and workings of the functions in a C++ class are analogous to the architectural details of an H++ component (which could include all kinds of specifications such as floor surface material or construction details). When an architect is arranging and connecting two components in H++, he doesn't have to worry about the construction details of each component, he only has to be concerned with whether the exterior walls can be merged together into a single wall to unify the structure (which the H++ software indicates as the user is dragging a component). This is analogous to two classes in C++ communicating with each other through their public functions with each not worrying about the inner workings of the other or what the other is doing with its own internal data. A house is a complex system, similar to a piece of software, but using H++ allows an architect

assembling components to only have to worry about how to position two to four pieces relative to each other (after the desired components have been selected). It's providing a layer of abstraction, and the software gets to worry about figuring out the details.

One of the interesting ramifications of diffusing and abstracting the design process in this way is that there become two distinct architectural tasks that are performed more or less in isolation: component designing and component compositing (which itself includes both choosing and connecting). Many architects more or less separate these tasks in their own minds when designing, but H++ explicitly separates them and makes it possible for an architect to be arranging 3 components designed by 3 different architects, all of whom have never met each other. Only the component compositor has a say in how the components relate to each other in the final design. Perhaps some architects would be better suited for component designing while others would be better compositors. Perhaps the training for each class of architect would be different.

Deciding how to conceptualize a house design in terms of these types of components is perhaps one of the more difficult jobs for both the component designers and compositors. Each has to ask themselves "why would I want to consider this set of rooms/spaces/functions as a single component apart from the rest?" In the example with the artist couple previously described, the living room, kitchen, bath, and bedroom were distinct from the studio in both scale (the living areas could be normal-to-small-sized living areas relative to each other while the couple desired a relatively large studio space) and quality (the studio was mostly for working, so its interior finishes didn't matter as much so long as it came with space). Functionally, there is also a living/working separation. Other reasons for grouping rooms or spaces together in one's conception of a component could include physical proximity or architectural style (perhaps some clients like modern airy living/kitchen areas but prefer more enclosed stone-clad sleeping areas). One can imagine clients who may desire two separate master suites, so the component compositor architect might search the database for a component that consists only of a master suite, and possibly use two of the same component in the house design.

As an exercise in conceptualizing a house design in terms of multi-room components, and to help imagine why certain spaces might be grouped together into a single component, the iconic Eames house (see Figure 6) was created as a 3D model and then divided in several different ways. In each situation, the original house was imagined as if it had been made out of a handful of components, but in each case the components were different. In the most basic deconstructing of the house, the living room is a single component, the dining/kitchen/bathroom/bedrooms are a single component, the patio is a single component, and the studio/utility/loft space is a single

component (see Figure 7). This is an easy division because all of the components are separated by a simple vertical plane. With more interesting decompositions, such as the one with the kitchen and dining rooms grouped with the patio and utility room into a single component (see Figure 10), one is led to ask what kind of house or client request might end up requiring a component with those spaces grouped together.

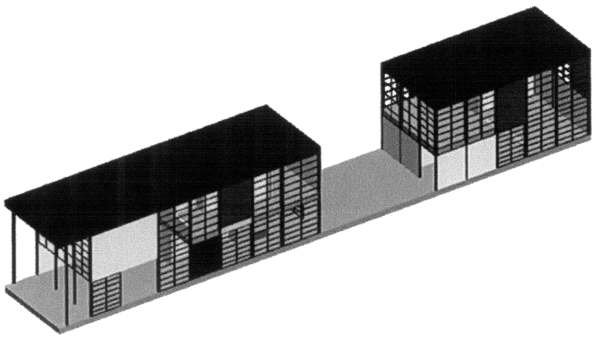


Figure 6. Original Eames house.

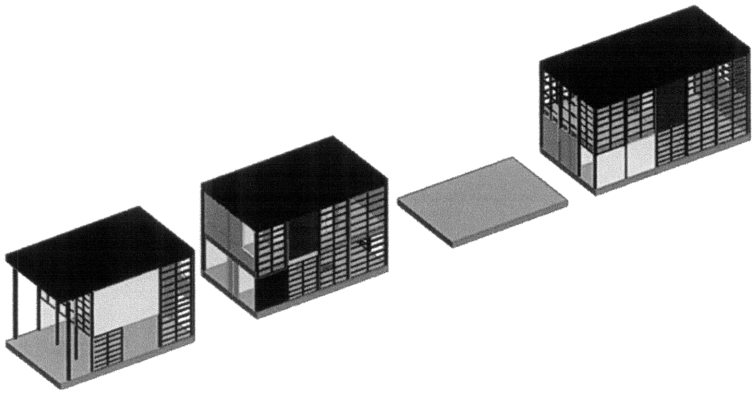


Figure 7. Eames house division scheme 1.

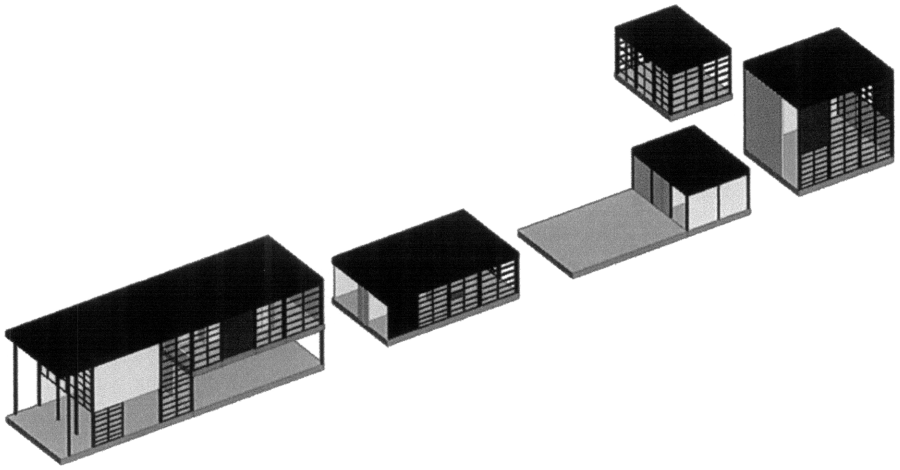


Figure 8. Eames division scheme 2.

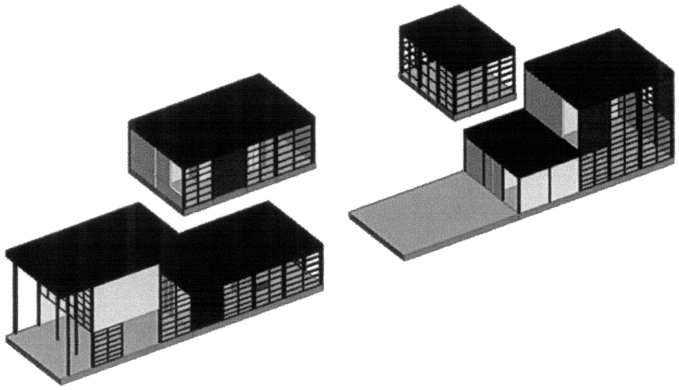


Figure 9. Eames division scheme 3.

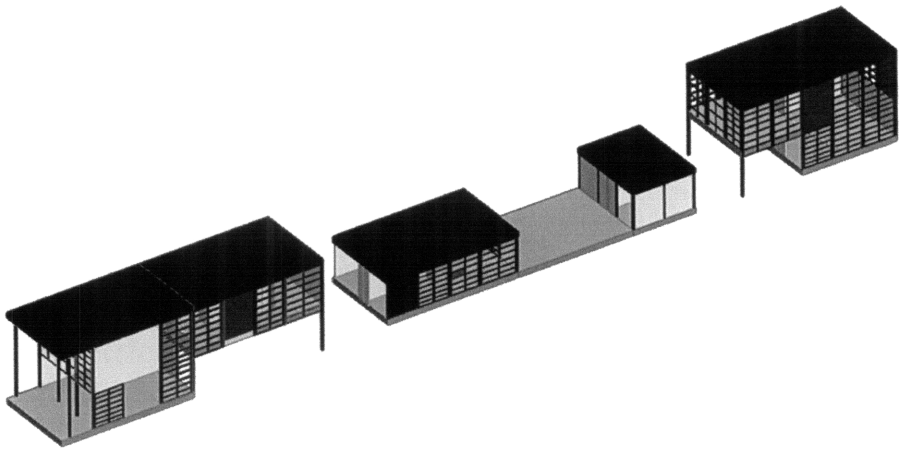


Figure 10. Eames division scheme 4.

FOCUSING ON DIGITAL AND DESIGN

Componentizing the process of designing houses opens up many of the efficiencies inherent in all component systems, such as reusing the same work multiple times and being able to look at a complicated system from a broader viewpoint. The H++ component concept is different from others in part because its components are non-standard, with the definition of what constitutes a component being left wholly to the component designers and compositors. Another important way that H++ is different from other component systems commonly seen in architecture is that most other systems deal with physical components, say wall or floor sections, that are designed to interface with other similar components. While H++ could certainly work well in concert with some level of pre-manufacturing, its focus is on pre-*designed* components, and all it does in the end is create a design that could be physically built in any number of ways.

By focusing on the digital aspects and letting the physical be implemented elsewhere, H++ avoids many of the pitfalls of physical component systems. Those creating physical component systems have to worry about coordinating disparate manufacturers and getting contractors used to new methods of construction. Such systems frequently end up not saving money because not enough people adopt them to make the economies of scale kick in. At its base level, H++ simply creates a design in a more or less isolated environment; H++ could be useful even if its database contained only 10 components designed by a single architect, though obviously more would be better in many cases.

This digital/design focus also allows H++ to take advantage of the various distinct markers of an e-commerce system. iTunes®, software from Apple® (see Figure 11), a program for finding and downloading digital music, is a perfect example of an optimized e-commerce system to use for comparison purposes.

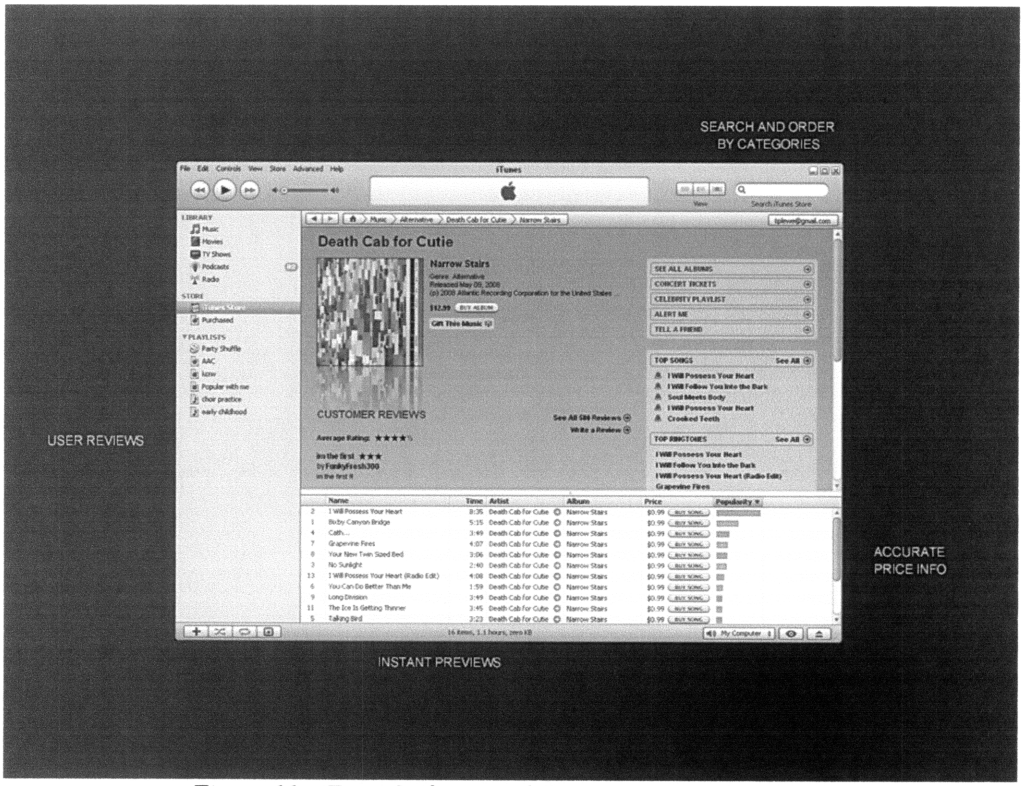


Figure 11. iTunes® from Apple® is a consummate example of e-commerce.

One of the most basic principles of e-commerce is the ability to search according to different parameters among a large set of options. In iTunes®, a user can search for and sort songs by genre, date released, popularity among users, and myriad other pieces of data, which are each useful in different circumstances. The ability to search for components by the rooms contained within has already been mentioned as a part of H++, but there are limitless other data by which one could search for a given component. Consider for example: square footage; projected cost; keywords such as modern, airy, enclosed, colorful; materials like bamboo or stained concrete; traditional styles, e.g. Georgian columns; energy use or carbon footprint; zip codes in which the component has already been approved to meet local design codes; ceiling height; projected durability; handicap access; time required to build; etc. Some of these pieces of data would be supplied by the architect who designed a given component. Others would come from another hallmark of e-commerce systems that could be easily implemented in H++ (though it has not been implemented in the prototype other than as a simple tag): user reviews.

With H++'s database of components being used over and over in different situations, over time each component could aggregate user reviews that might illuminate certain qualities of the component not made clear in the architect's own description. In iTunes®, there is a popularity indicator based on how frequently a song has been purchased, but there are also written reviews of albums by those who have purchased them. If an architect designs a component that tends to have a leaky roof and those who have used the component are able to write about it, future customers can be warned ahead of time. Additionally, the architect of said component could modify the design and release version 2.0 of his component, which fixes the flaws of the previous iteration.

One of the most important pieces of data coming from user reviews would be the actual cost of building a component, which would vary a lot based on how the component was integrated with the larger design, who built it, and where it was built. But one could get a good ballpark figure, especially relative to other components. Accurate cost prediction is standard in e-commerce but severely lacking in custom home design, and that uncertainty alone can be prohibitive for many prospective clients. Only if a design is built over and over can one start to predict its actual cost, and that repetition (without a paradigm like H++) results in the tract housing this thesis is targeting.

The final important e-commerce attribute that applies to H++ is the instant preview, which in iTunes® lets you listen to a 30-second clip of a song without paying anything. Theoretically, clients could begin their housing design process by going online and searching through images and descriptions of components themselves to see what they like, which is one level of instant preview. When the client sits down with their component compositor

architect who searches for and finds a given component in H++, that architect then loads it into the 3D workspace, at which point he can align it with a neighboring component and move the camera inside the model to show the client around from a first-person perspective. It would be a simple matter to export a file of the completed design to a 3D printer using standard 3D file formats. Within a day of meeting with their architect, clients could have in their hands a 3D model of their new home, as well as several renderings of the interior spaces to go along with the various design drawings generated by the software.

EMBRACING BRICOLAGE

The H++ software and paradigm as so far described may cause alarm to some architects due to its fostering of a bricolage (or collage) aesthetic, potentially resulting in houses that appear pasted together from pieces with disparate visions. "You can't simply take one architect's design and ram it up against another's to make a house, there's no regard for context in that," they say. "Someone could make a house that pastes a glass cube onto a barn."

It should be noted first of all that H++ is not trying to best the 2-million-dollar custom showpiece home, where the angle and finish of a bedroom wall continues as some significant reference through the entire structure until its termination at the corner of the lot. The H++ house is trying to best the tract home, as a compromise between the tract and custom home. So the question asked should be "would I rather live in this than in a tract home?" Also, the argument for embracing bricolage with H++ should be prefaced with the point that the extent to which a given house design is collage-like is entirely up to the component compositing architect. That architect may choose to only use components of his own design or those hand-picked by some developer to "match" each other, so that the resulting house designs are suitably unified to their tastes.

Much can be said for bricolage design however, creating unexpected situations by bringing together designs that weren't originally intended for each other. Consider buildings such as the Louvre or the German Reichstag; the original architects of each building would never have imagined anything like the future additions later "pasted on" by I.M. Pei and Norman Foster, and would likely consider them heretical abominations that destroyed their original vision. Many now enjoy the contrast, unexpectedness, and inventiveness that can only come when multiple visions are merged without pre-meditation by all the designers involved. The control over context that Norman Foster had in adding his dome to the Reichstag could be considered analogous to the control over context that a component compositor architect has when choosing and arranging components. In neither case is the mash-up totally random.

Even when mash-ups of disparate designs are much more random than the examples just cited, positive things can happen. Look at any modern downtown area in a large American city and you're likely to find an old classical Cathedral bordered by a stoic modern tower on one side and a brick building from the early 20th century on the other side. The diversity from one block to the next is one of the things that make a place like Manhattan so exciting to be in. Compare Manhattan with Le Corbusier's City of Tomorrow, the futuristic plan where Corbusier had control over every design aspect and could thus ensure that every piece related to every other piece. When one designer has ultimate control, whether in the City of Tomorrow or in the

Golden Meadow Valley Estates housing development in Anytown USA, a feeling of lifelessness often tends to be the result. While the function of a city is certainly different from the function of a house, it could be argued that having real variety of design from one part of a house to another could contribute a sense of liveliness over a totally unified house design.

Other art forms also suggest value in a compositor bringing together disparate visions into a single work. Much of hip-hop music is made by taking samples from older songs and remixing them with additional beats and vocals to make an entirely new work that never could have been conceived by the creator of the original song. In a more abstract way, a character in a movie is an example of bricolage design. The character starts as a written document created by a writer who doesn't know the eventual actor; that writing is then interpreted by an that eventual actor, who is chosen and molded by the director to create a character that none of the individuals involved could have created on their own.

Still, one of the more difficult tasks in getting architects to use H++ will be convincing them to cede control over large aspects of house design that they're used to controlling. As with most things, the convincing will probably only come if architects find they can get more enjoyable work and make more money by using the system than they can without it, or if the system is demanded enough by clients due to their own cost/benefit analysis.

ECONOMY OF H++

The economics of architectural design under H++ look very different than the traditional several-months-long architect/client partnership that results in the architect getting 15% of the final construction cost of a home. It has to look dramatically different in order to replace the tract house as the dominant model of housing for middle-class citizens, because a 15% design fee and all the hassle is never going to be worth it to the majority of people. Can an H++ house approach a tract house in affordability? Are there any economic or hassle incentives that would push architects to use the system?

Let us first consider H++ economics through the eyes of architects using the system. Here we will assume we're using the future non-prototype version of H++ which uses an online database, handles financial transactions, and accumulates data from its users. For a component designer architect, the design process can happen completely independently of any client, i.e. the architect can design a component and upload it to the universal online database without anyone having commissioned the design. Each time a component is used in a house design, its architect automatically gets a fee (say, $100) administered by the system, paid for as part of the cost to the client whose house is being designed.

If an architect designs a good component that people like, he can make money off of it in perpetuity. His success is determined not so much by his ability to sound smart when selling himself to a client, but more by the quality and usability of his design. Of note, the barrier of entry for a new architect entering the field is lowered considerably with H++, because the new architect is not required to have some connection with a wealthy client to get that first big commission to launch his career. If he wants to design and upload a component, he can, and if clients browsing through the components online before meeting with their compositor architect really like it, they'll use it and he'll get paid. The democratization of architecture inherent in this process reflects the democratization brought about by technology in so many other fields, including news reporting, music and video production, software creation, etc. While some may resist that democratization, it's inevitable, if other industries are any indication.

Component compositing architects, who may also be component designers, have an altered economic model from standard architecture practices as well. Unlike component designers, these architects do actually meet with clients, but the amount of time they spend with the client could be as little as an hour. They would likely charge an hourly rate in addition to the fees incurred by using the software (which would automatically charge them for using the component designs of other architects). Some architects could use the H++ software strictly with components they've designed themselves, collecting both an hourly fee as well as a design fee for each component – the

total cost to the client would still be the same. Either way, the hassle of negotiating payments with clients and dealing with them for months is greatly reduced. Both client and architect know what the design work will cost (the compositor architect's hourly rate multiplied by a few hours, plus the design fee for each component, which is displayed and tallied by H++ while searching for and combining components). Additionally, thanks to the user feedback and reviews mentioned above, the client has a very good idea of what the total cost will be to actually build the home.

From the client's perspective, simply removing all of the unknown costs makes a big difference. But can the H++ house meet the tract house in price? Let's assume that the total cost for designing a house using H++ is in the neighborhood of the cost for buying a complete house design from a book of standard plans (in current US dollars, we'll approximate somewhere between $500-$1000). That would be enough to cover the fees for the component compositor working for a couple of hours as well as the $100 that goes to each component designer. Then say the client uses a standard local contractor to build their house from the generated designs, which we'll assume costs about as much as building a house from plans purchased from a book of standard pre-designed plans. So far, we're even with the cost of buying pre-made plans and having them built, which is some percentage higher than a tract home (because tract housing developers find economization through building the same house over and over) but much less than a custom designed home with 15% architect fees. However, with our H++ house we have many opportunities to save money that are not available to tract house purchasers. Certain areas of the house can be made of components with less-expensive finishes or details than other areas, the higher customization allows a better client/house fit with reduced superfluous space, and perhaps a component such as one containing extra bedrooms can be included in the design but marked as a future addition which can be built when funds allow. This is to say nothing about economizations that could be brought on by some level of pre-manufacturing.

Based on these projections (see Figure 12), it is entirely possible to imagine a house, such as the sample described earlier for the young artist couple, which would end up being more affordable than a tract house while being a better fit for the clients and having a completely unique design.

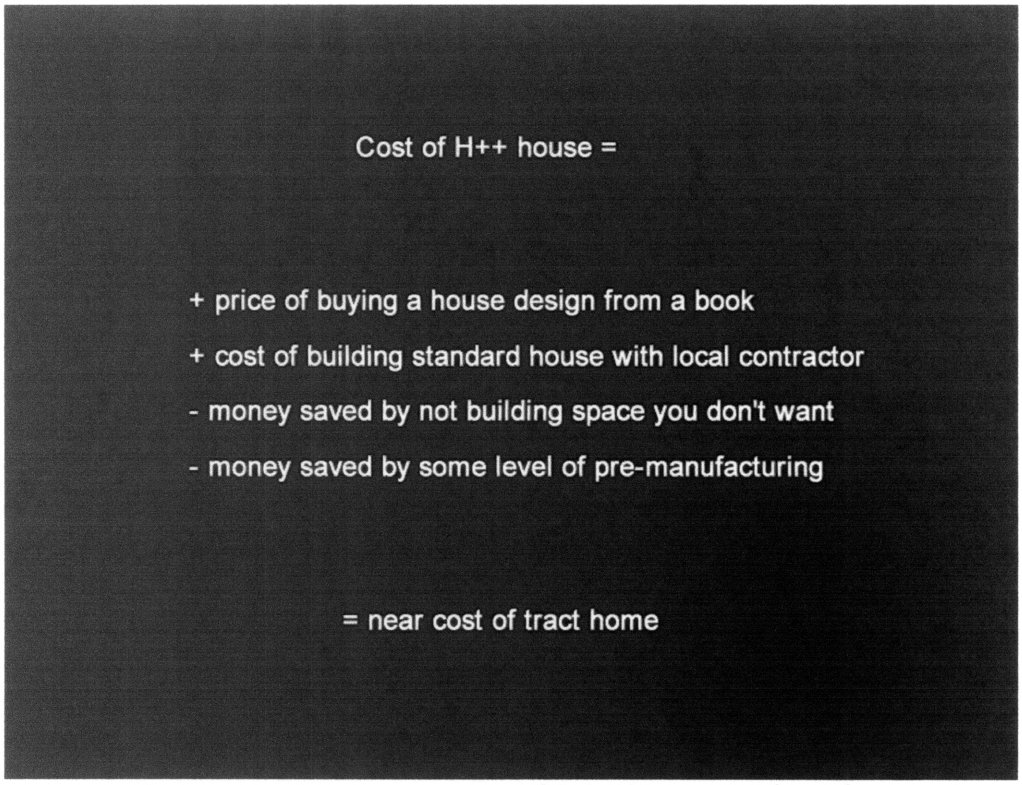


Figure 12. A cost analysis of an H++ house.

H++ IMPLEMENTATION

Programming

The H++ prototype software created for this thesis was programmed using C++, with the OpenGL library used for 3D drawing and the Qt library used for creating the windowing system. By using C++, OpenGL, and Qt, the code is able to be compiled for Windows®, Mac®, or Unix. The program used here was compiled for Windows®.

Database

The component database in the non-prototype version of the software would exist on a central online server and be accessed by each copy of the H++ software over the internet. In the prototype created for this thesis, this database is simply a folder placed next to the H++ executable file. The folder contains a master xml file called database.xml, which is simply a list of other xml files, one for each component in the database. Each database component includes its own xml descriptor file (the one listed in database.xml), a 3D model file (with the same base name as the xml file, in obj format), and a preview jpg thumbnail file (also with the same base name as the xml file). The obj file has a companion mtl file, as specified by the obj format, which includes descriptions of the materials used in the 3D model. When H++ starts up, it loads into its internal database the information in database.xml and each of the individual component xml files, so that when a user searches for a component it can return the relevant results and load in the appropriate 3D model file. A sample database.xml file might look like the following:

```
<component file="beds.xml">
<component file="living_beds.xml">
<component file="kitchen_dining.xml">
<component file="kitchen_dining_beds.xml">
<component file="kitchen_dining_patio_utility.xml">
<component file="living_porch.xml">
<component file="living_porch_kitchen_dining.xml">
<component file="loft.xml">
<component file="patio.xml">
<component file="patio_studio_utility.xml">
<component file="patio_utility.xml">
<component file="studio.xml">
<component file="studio_loft.xml">
<component file="studio_utility_loft.xml">
<component file="stairs.xml">
<component file="kitchen1.xml">
```

Each individual component's xml file includes several optional tags that define different attributes. For the future implementation of H++, this file could be automatically generated based on different inputs such as a special 3D model file exported from custom modeling software, data collected over time from use of each component, etc. However, in the prototype implementation, this file is hand-written. Following is a sample of beds.xml, the file for a component containing two bedrooms and one bathroom, which accompanies the 3D model obj file:

```
<properties>
  <data rating="7.5/10">
  <data bed="2">
  <data bath="1">
  <data name="Eames Bedrooms">
  <data price="30000">
</properties>

<search>
  <data terms="eames case study bedrooms beds bath">
</search>

<bounding>
  <data shape="0">
  <data faces="0 1 2 3 4 5">
</bounding>

<slider>
      <data name="Door 1 Side">
      <data shape="0">
      <data vector="1 0 0">
      <data position="220 221 222 223">
      <data vertices="220 221 222 223">
</slider>

<plan>
      <data shape="0">
      <data faces="22 23 24 25 26 27 28 29 30 31">
</plan>
```

The <properties> tag includes metadata like average user rating, rooms contained within the component, price, and square footage. Some of these pieces of data are displayed next to the thumbnail in the search results, and are used to figure out aggregate data for the entire unified house structure.

The <search> tag is one of the most important items contained in this file. Here are listed each of the search terms that will trigger this component to show up as a result of a user's search query. In a future implementation,

search terms would be automatically culled from the architect's description of the component, user reviews, and other data. In the prototype H++ software, these terms are all manually entered here. If a user searches for multiple terms, only components matching all terms will be returned as results to the query.

The xml file then defines the <bounding> tag, which identifies certain polygons (also called faces) within the 3D model file that delineate the walls of the component that can be snapped to and merged with the walls of other components. In this example file, the first six polygons that appear in the obj file have been created to identify these exterior wall locations. These polygons are included with the obj file in addition to the geometry of the component itself and are hidden by the software when displaying the component, but used for determining when to snap neighboring components together.

Also included in this file are <slider> tags (there can be more than one of these) that allow the component designer to specify vertices in the 3D model that can be adjusted parametrically by sliding along a vector. For example, if an architect wanted to make a doorway have variable width, he would set this tag up so that all of the vertices on one side of the doorway slid along a horizontal vector in the plane of the wall. This tag identifies the name of the parameter, the index of the shape in the 3D model file that is affected (usually there is only one shape in the obj file, which is index 0), the set of vertices that will move, the vector they can move along, and the set of vertices used to position the control handle for the user to grab in the software interface.

Finally, the xml file includes the <plan> tag, a list of polygons that will be used to draw the floor plan of the component when the floor plan generation function is called. These polygons are to be included in the 3D model file in addition to the geometry of the component itself, and will be hidden by the software when displaying the component in the 3D workspace. The polygons identified here are the only thing that will be drawn when generating a plan using the component.

Obviously, in the final implantation of H++ the component designer would have control over some data currently listed in the component xml file (like adjustable parameters) but not others (like user ratings).

Figures 13 and 14 show a sample component's 3D model file. In Figure 13, the component's main 3D geometry is drawn in wire frame and the bounding and plan polygons are displayed flat shaded. All of these are included in the obj file loaded in by the H++ software, but only the main 3D geometry is visible in the 3D workspace. When other components are dragged near any of the 4 bounding polygons (shown at the top in Figure 14) the nearest of the other components' own bounding polygons snap to this component's nearest bounding polygon. When the user chooses to generate a

plan drawing of their house design, the camera goes to a top-down view and draws only the plan polygons for each component (such as those seen in the bottom half of Figures 13 and 14) and outputs that to a file. The plan polygons can form a plan drawing that is as detailed as needed, though they are very simple in this example.

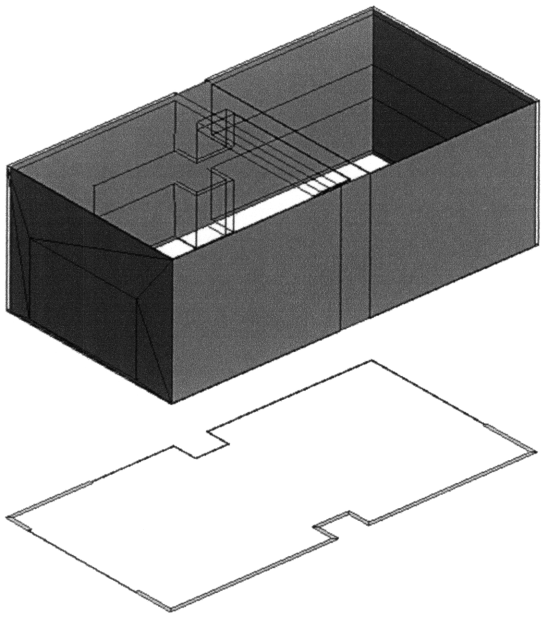


Figure 13. A component's 3D model file, showing bounding and plan polygons with the component geometry overlaid in wire frame.

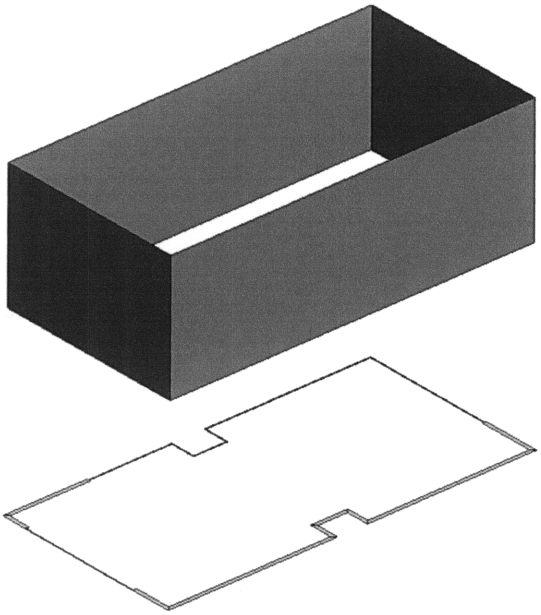


Figure 14. A component's 3D model file showing only the bounding and plan polygons.

User Interface

The leftmost panel in the H++ prototype interface (see Figure 15) includes a simple text entry field in which to type search terms, and a button to activate the search. The search function will return results based on matching the words entered in the search box with those listed in each component's individual xml descriptor file. If a component designer creates a component consisting of 2 bedrooms and 2 bathrooms, which is designed for modern concrete construction, he might want to include the words "bed", "bath", "modern" and "concrete" as search terms relevant to the component. Then when a user searches with the search query "bed bath modern concrete", each component that includes all of those search terms will show as a result.

Results to search queries are listed as thumbnails below the search entry field. Next to each thumbnail is shown some relevant text, including the name of the component, its user rating, and its cost. Any other number of interesting pieces of data pertaining to the component could be displayed here. Clicking on any of these thumbnails will load that component into the 3D workspace.

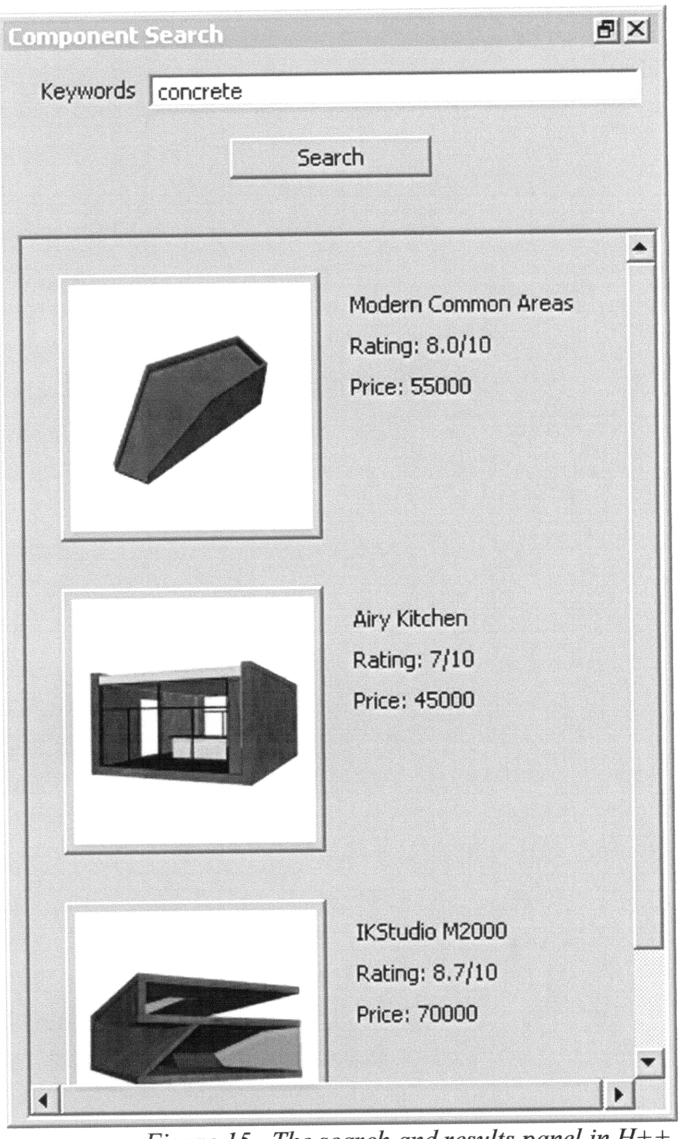


Figure 15. The search and results panel in H++.

The 3D workspace is the main window in the H++ software (see Figures 16 and 17). Here the architect can load in as many different components as he likes, using the search panel just described to locate the desired components. When the user clicks on a component in the 3D space, the component becomes selected and a manipulator is displayed at the center of the component (the green tool seen in the center of Figure 16). This manipulator allows the user to click on different axes to drag and move the component along that axis, or along the ground plane if the center handle is selected. The component can be rotated by clicking and dragging on the ring surrounding the manipulator.

If the xml file corresponding to a given component has defined vertices that can be adjusted parametrically (for example a door opening that can be made wider or narrower), each group of these will display as a small handle with a text label (the 3 yellow balls with adjacent text in Figure 16). These handles are only displayed when the component is selected. Clicking and dragging on these handles will adjust the component according to the vectors specified by the architect of the component.

When a component is selected and being dragged, if one of the faces marked as a snapping face (in the component's xml file) comes near to a snapping face on another component, and they are nearly aligned, the software will cause the components to snap together so that they're perfectly aligned with each other and the user knows that the software considers the two components to be merged. Figure 17 shows two components snapped together. In the prototype software, all snapping walls can be snapped with all other snapping walls, but in a future version certain walls could be tagged to only snap with other walls of similar construction, assuring that they could in fact be merged in real life construction.

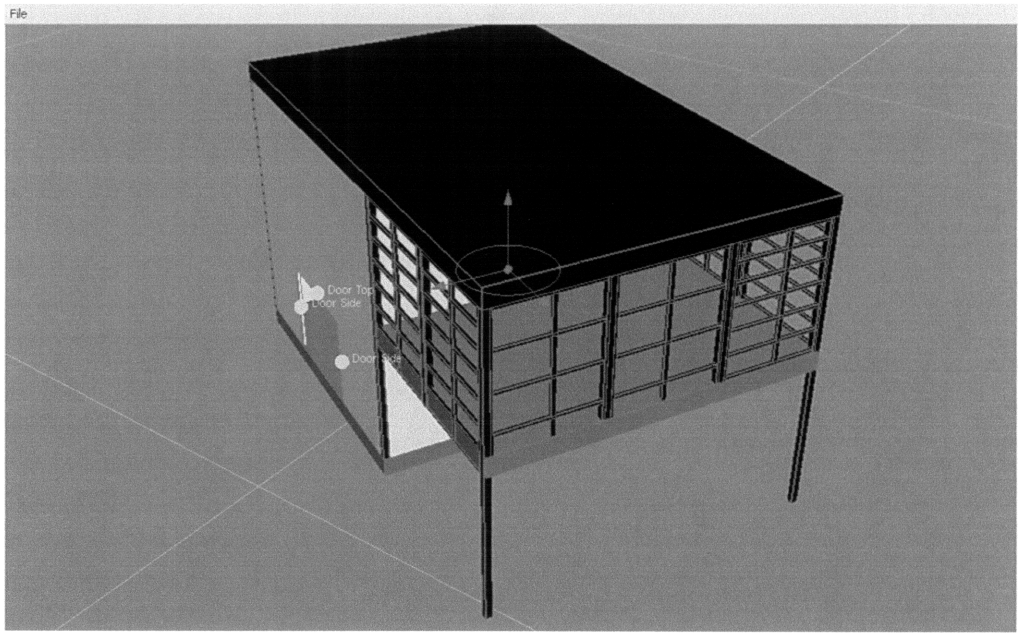


Figure 16. The 3D workspace in H++, showing a selected component, including the manipulator and three parametric control handles for adjusting a door opening.

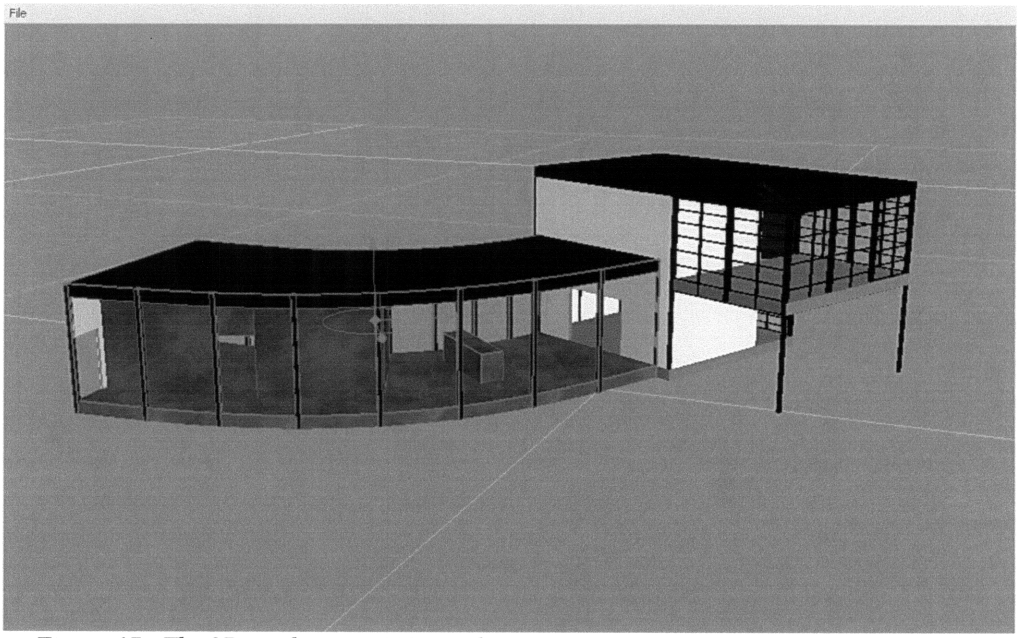


Figure 17. The 3D workspace in H++, showing two components which have been snapped together to form a single structure.

The window on the top right of the interface (see Figure 18) displays properties specific to the selected component. In the prototype H++ software, this window is only a placeholder, but here is where the user could choose options like marking a component as a future addition. This way plans could be generated for the house without the component, but additional plans would also be generated for the component as an addition to the house. In a future version of the H++ software, certain components could define specific doors or windows that can be toggled in and out of the design by using check boxes in this window, or perhaps there could be drop down menus that would let the user specify alternate exterior cladding or other options.

The final window, which is displayed on the bottom right of the interface (see Figure 19), displays the properties of the aggregate house structure made up of all the components currently in the workspace. It will use the data provided in each component's xml descriptor file to tally the total price, square footage, number of bedrooms, and number of bathrooms. Other data which could be displayed here in a future version includes information like the approximated construction time or total estimated energy use of the completed house.

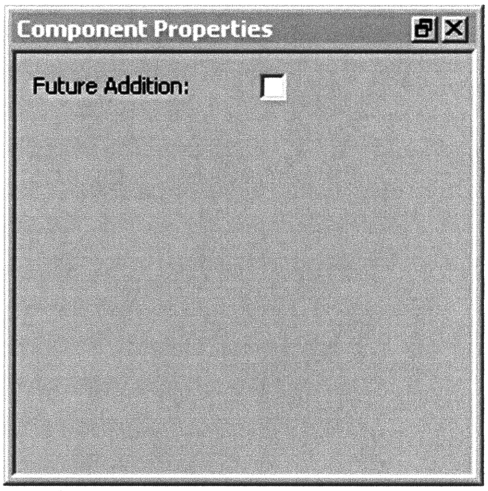


Figure 18. The component properties panel in H++.

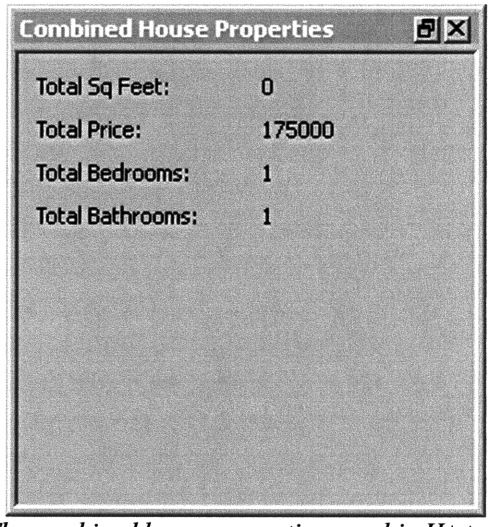


Figure 19. The combined house properties panel in H++.

The file menu at the top of the interface (see Figure 20) allows the user to load obj 3D model files into the workspace without having to access them through the database. This is mainly for playing with different shapes, as files loaded in this manner do not support the features of H++ that depend on the component's xml descriptor file. Components can also be deleted from the scene using this menu. The final item on the menu will generate a bitmap at the location "C:\plan.bmp" which is a basic plan drawing of the completed composition. This plan will only draw properly if each component in the 3D workspace has plan polygons properly delimited in its xml descriptor file. The drawing is generated using OpenGL drawing commands at a low resolution, and is only a prototype of what a future version of the software could produce. Figure 21 shows the plan generated from the component model seen in Figures 13 and 14.

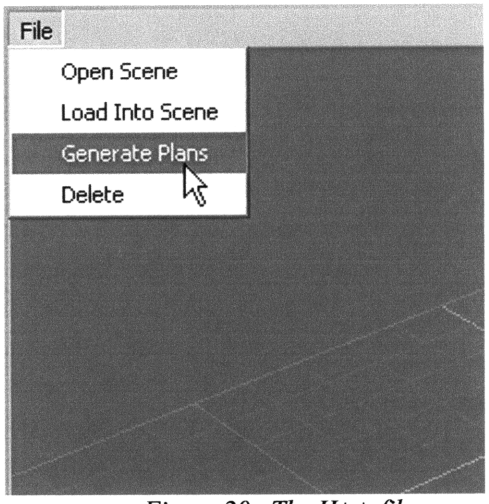


Figure 20. The H++ file menu.

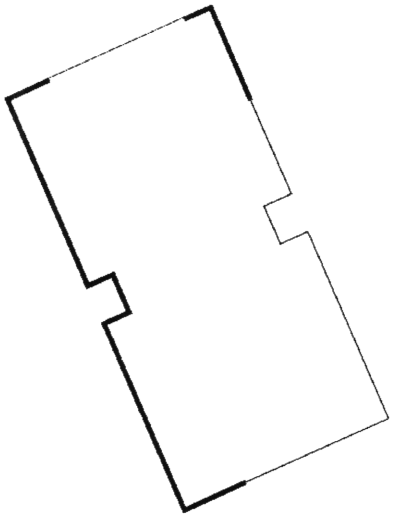


Figure 21. Plan generated in H++ from the component model seen in Figures 13 and 14.

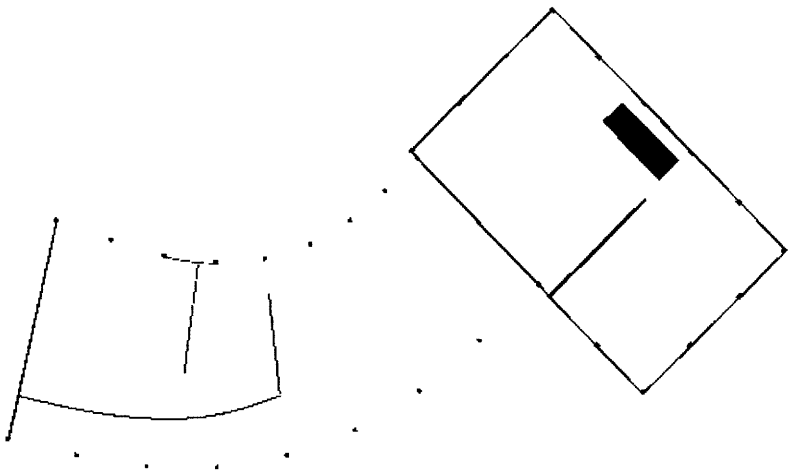
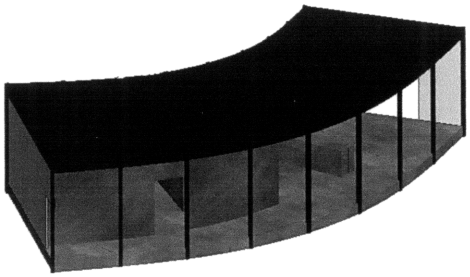


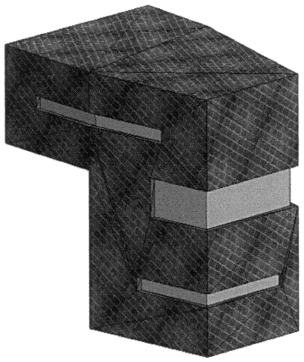
Figure 22. Plan generated in H++ from the composition of two component as seen in Figure 17.

EXAMPLE COMPONENTS

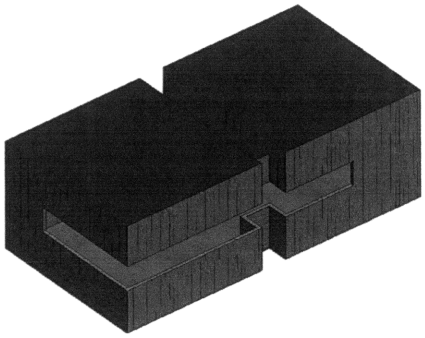
Following are some of the components created to make example house designs using the H++ software prototype. Each component model created has an accompanying xml descriptor file. Some of these components include parameterized aspects while others do not. Many of the components came from the Eames house exercise detailed previously.



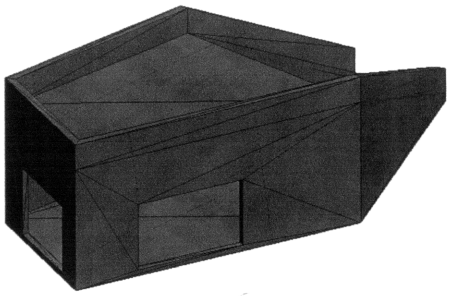
Component example 1. Contains bedroom, bathroom, kitchen, and living area.



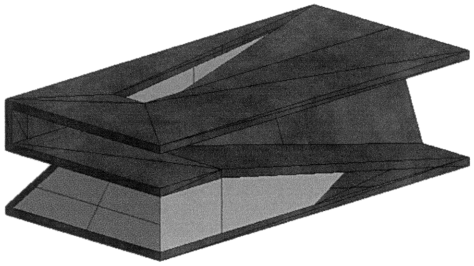
Component example 2. Contains 3 bedrooms, bathroom, and living area.



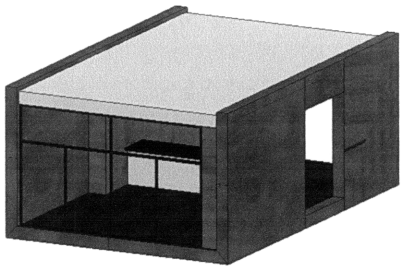
Component example 3. Contains garage and storage.



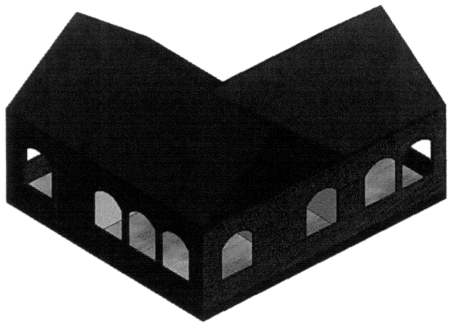
Component example 4. Contains living area.



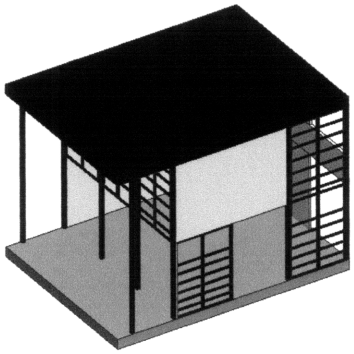
Component example 5. Contains living area (from IK Studio).



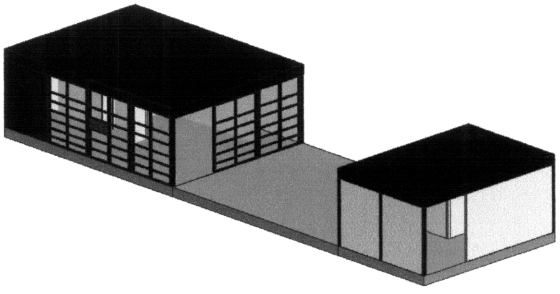
Component example 6. Contains kitchen and dining.



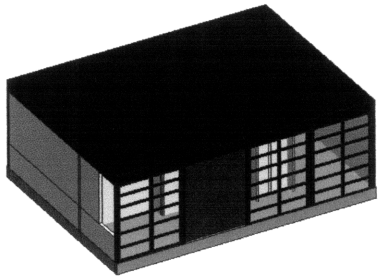
Component example 7. Contains 3 bedrooms.



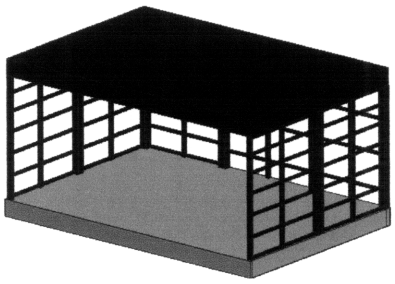
Component example 8. Contains living area.



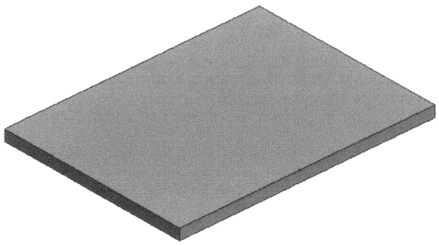
Component example 9. Contains kitchen, dining, bathroom, patio, and utility.



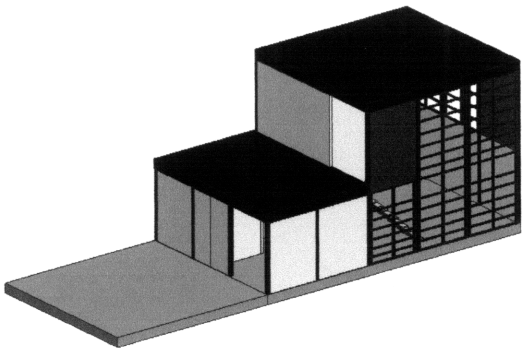
Component example 10. Contains 2 bedrooms and bathroom.



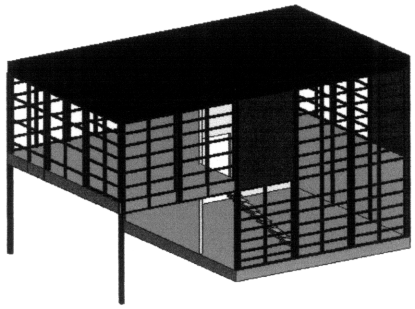
Component example 11. Contains loft.



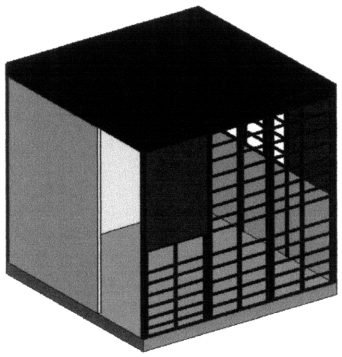
Component example 12. Contains patio.



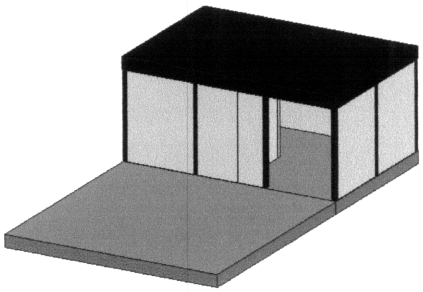
Component example 13. Contains studio and utility.



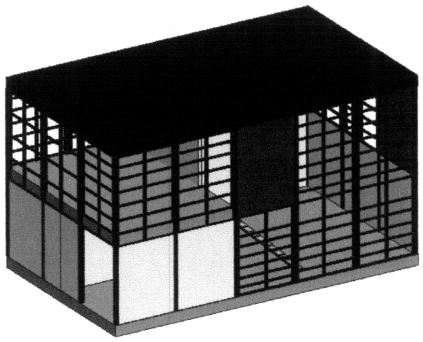
Component example 14. Contains studio and loft.



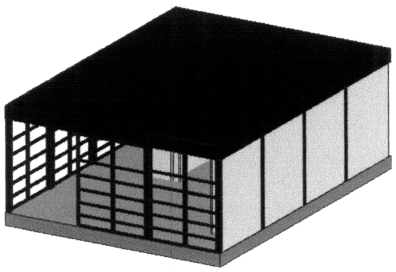
Component example 15. Contains studio.



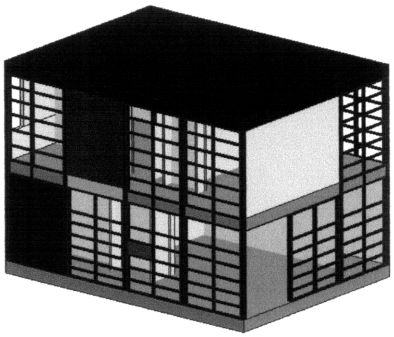
Component example 16. Contains patio and utility.



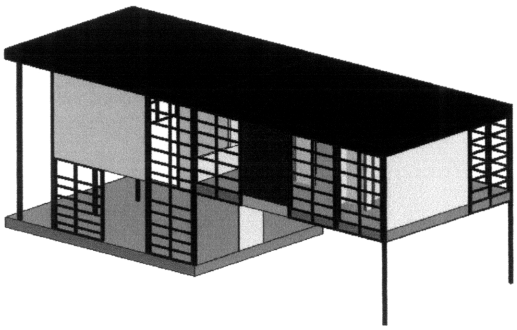
Component example 17. Contains studio, loft, and utility.



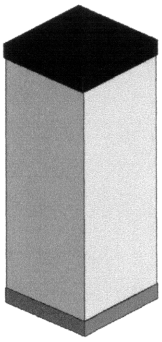
Component example 18. Contains kitchen, dining, and bathroom.



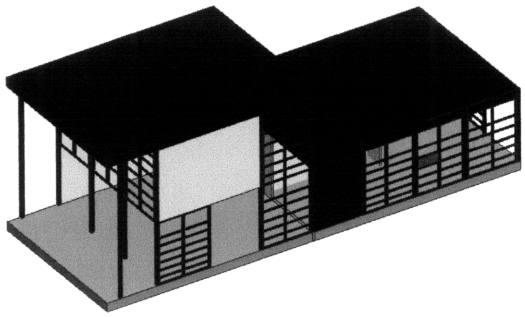
Component example 19. Contains kitchen, dining, 2 bathrooms, and 2 bedrooms.



Component example 20. Contains living area, 2 bedrooms, and bathroom.



Component example 21. Contains staircase.



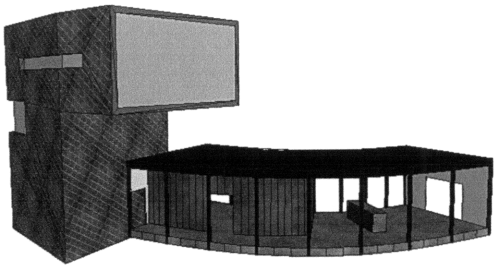
Component example 22. Contains living area, kitchen, dining, and bathroom.

EXAMPLE HOUSES

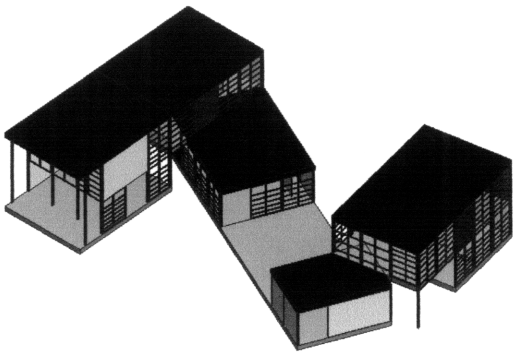
Following are some of the example houses created using the components listed above. It is interesting to observe the natural vernacular that emerges due to the high percentage of components coming from the original Eames house in the database. This is an example of how influencing or restricting the database of components lets a natural style emerge without enforcing heavy-handed design guidelines. Some developer could restrict the components used in his development to only those descended from the Eames design, or use half Eames components and half others as has been done here, and the neighborhood would automatically take on a distinct style of its own.

While the houses presented here are all fairly sprawling in design, this is not a necessary result of using H++. The sprawlingness is encouraged because of the design of the particular components used in this thesis, but components could be designed to stack tightly atop and adjacent to one another to create much more compact housing units if desired.

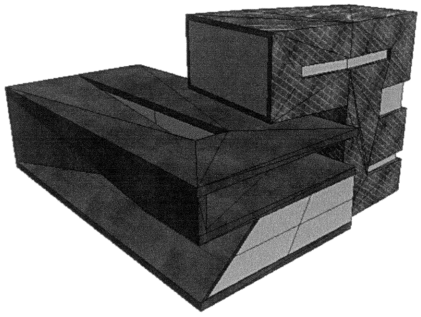
Also of note, a few non-single-family-housing structures were created. While this thesis focuses on housing, structures with other interesting uses can be generated from the same components.



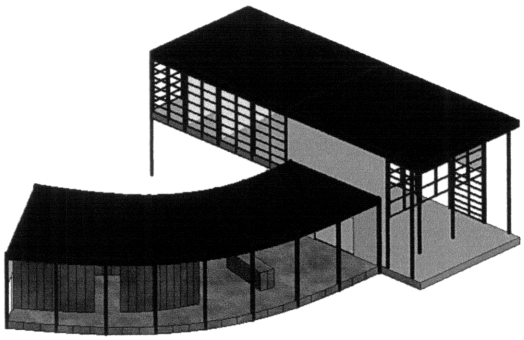
House example 1. The bedroom wing looms.



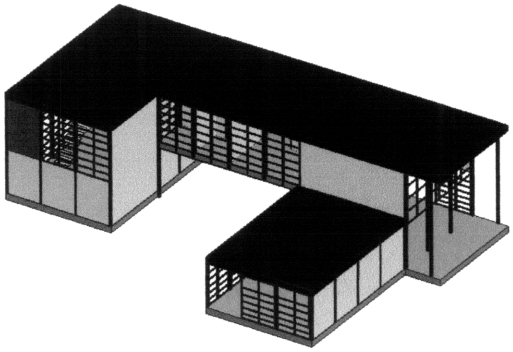
House example 2. Here the components are only connected by floor/ceiling plates, requiring outdoor excursions between house sections.



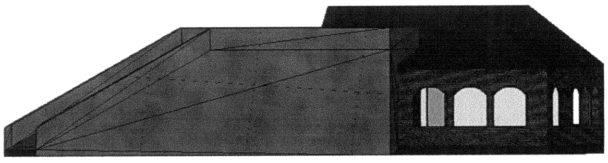
House example 3. The concrete component has indeterminate space, which the client can turn into whatever may be desired.



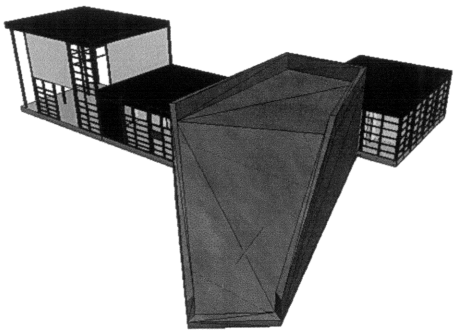
House example 4. Two unique living areas extend into each other.



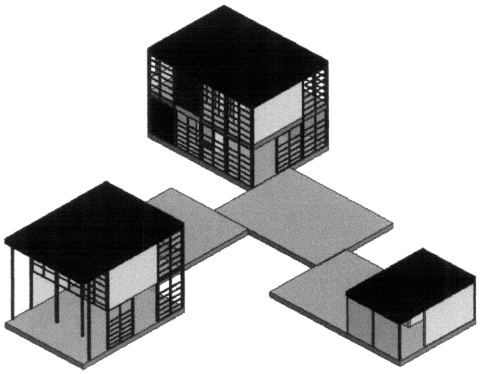
House example 5. The Eames house reconfigured.



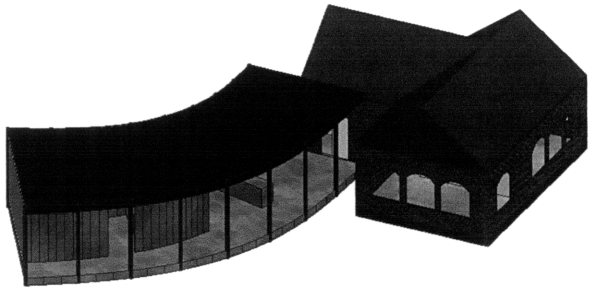
House example 6. Traditional brick wraps around modern concrete.



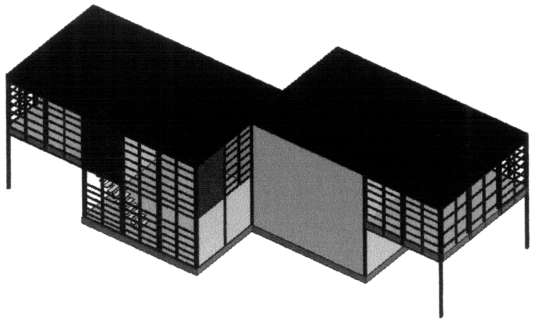
House example 7. A unique living space connects two Eames components.



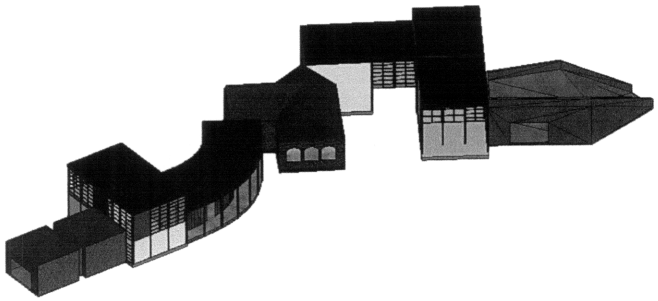
House example 8. A pavilion house.



House example 9. Traditional bedroom wing with modern living areas.



Non-house example 1. Large freestanding studio space.



Non-house example 2. Odd multi-family aggregate with interstitial courtyards.

CONCLUSION AND FUTURE RESEARCH

With the database loaded, it is straightforward to search for and load desired components, adjust and connect them using the on-screen manipulators, and automatically generate a plan of the unified design with a menu command, all in a matter of minutes. While the prototype H++ software only draws simple plans based on polygons included in a component's 3D model file, it serves as a proof-of-concept for a future, more-detailed implementation that could generate all of the construction documents necessary for a contractor to be able to build the house as designed in H++. The workflow of this future version, from the user's perspective as a component compositor, wouldn't necessarily need to be much more complicated than the one created in this prototype. Since the example houses shown were each created in less than 10 minutes total time, the prototype software gives strong evidence that an architect could design a house using this system in a short enough amount of time that the economic estimates put forth would hold true. That is, a custom house could be created at a cost which is competitive with tract housing.

Much remains to be done as future research and implementation. For the system to be useable in a real-world situation, more work needs to be done in taking care of the construction details at the intersection of two components. The prototype H++ software only addresses this on the most basic level , by snapping two planes to each other. A future version could potentially use boolean algorithms to actually merge 3D volumes. A real-world H++ software implementation would need to have knowledge of different types of wall construction and how to handle merging each.

With the software as it now exists, it would be fairly easy to export 3D model files to a 3D printer to create physical models of the homes designed. This could also be a small-scale example of certain kinds of real-world construction. Additionally, automatic camera placement techniques could be used to automatically create renderings of each completed house design.

Another area which could use further research is the potential integration of this system with pre-manufacturing techniques. Each component retains some percentage of its unmodified stand-alone structure after it is integrated within a larger house design, and those unmodified portions could easily be shipped to the construction site pre-manufactured. While the level of pre-manufacturing wouldn't be as high as possible in tract housing, it could be high enough to help bring down costs significantly.

Since H++ relies so much on the results returned from search queries, different algorithms for determining component search results could significantly affect what gets built, potentially steering the entire identity of a city. Further research could be done into various ways to tweak the search algorithm, and to surmise what the results of those tweaks might be.

Even in its current prototype stage, H++ can be useful for an architect to play around with spaces and generate ideas. Though the software can't currently create construction documents, it does allow someone to quickly generate design ideas from which he can then create needed drawings manually.

However, it would be a shame if this was as far as H++ went; the tract house is waiting so patiently to be bested.

REFERENCES

1. Colton, Kent W. *Housing in the Twenty-first Century: Achieving Common Ground.* Harvard University Press, 2003. 384.

2. Larson, Kent, Mark A.Tapia, and José Pinto Duarte, "Automated Design Tools for the Mass Customization of Housing." *A+U.* March 2001: 366.

3. Dwell Homes by Empyrean home page. Retrieved May 15, 2008 from http://www.thedwellhome.com/.

4. House_n Projects. Retrieved May 15, 2008 from http://architecture.mit.edu/house_n/projects.html.

5. Duarte, José Pinto. *Customizing Mass Housing: A Discursive Grammar for Siza's Malagueira Houses.* PhD Dissertation, MIT, 2001.

6. Phillips, Mathew Giles. Design By Searching: A System For Creating And Evaluating Complex Architectural Assemblies. SMArchS Thesis, MIT, 2007.

7. Dale, Nell B. *C++ Plus Data Structures,* Fourth Edition. Jones & Bartlett Publishers, 2006. 93-95.