Program Comprehension for Live Algorithmic Design in Virtual Reality

Renata Castelo-Branco
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
renata.castelo.branco@tecnico.ulisboa.pt

António Leitão
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
antonio.leitao@inesc-id.pt

Catarina Brás
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
catarinasaomiguel@tecnico.ulisboa.pt

ABSTRACT

Algorithmic Design (AD) is a design approach based on the development of computer programs to describe architectural models. The programs' outputs are digital architectural 3D models, which are visual by nature and, therefore, benefit from immersive visualization. Live Coding in Virtual Reality (LCVR) is a methodology for the interactive development of AD programs while immersed in Virtual Reality (VR), favoring a more intuitive development process for architectural designs. However, complex buildings tend to require complex AD programs and, despite the added visual aid, as programs grow in complexity, it becomes harder to understand which parts of the program were responsible for which parts of the model. Moreover, LCVR introduces a new level of complexity: interaction with both model and program in VR. This research proposes to ease the programming task for architects who wish to code their models in VR, by supporting program comprehension in the LCVR workflow with traceability and refactoring mechanisms. These features will help users interact with their designs from within the virtual environment.

CCS CONCEPTS

• **Human-centered computing** → **Virtual reality**; *User interface design*; • **Applied computing** → **Computer-aided design**; • **Software and its engineering** → *General programming languages.*

KEYWORDS

Virtual Reality, Algorithmic Design, Live Coding, Program Comprehension, Interaction Mechanisms

ACM Reference Format:
Renata Castelo-Branco, António Leitão, and Catarina Brás. 2020. Program Comprehension for Live Algorithmic Design in Virtual Reality. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3397537.3398475

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3398475

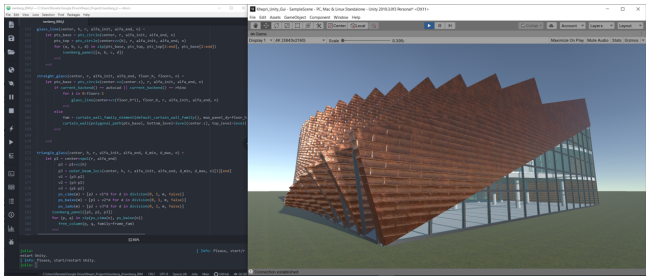

Figure 1: AD workflow showing the adapted model of the Isenberg School of Management Hub: on the left, the algorithmic description, and on the right, the model generated in Unity.

1 INTRODUCTION

Algorithmic Design (AD) has had an increasing impact in the architectural practice, having been adopted by well-known architecture studios such as Foster + Partners, Gehry Partners, and Zaha Hadid Architects. AD defines the creation of architectural designs through algorithmic descriptions [4]. As such, it allows architects to model more complex geometries that would take a considerable amount of time to produce otherwise, automate repetitive and time-consuming tasks, and effortlessly generate diverse design solutions without having to rework the model for every iteration [23]. Figure 1 presents an example of one such process, and Figure 2 shows the final product with four possible variations of the same model generated by a single algorithmic description. The model shown is an adaptation of the Isenberg School of Management Hub, an original project from BIG architects.

1.1 Visualization

Despite the innovation presented by this workflow, the visualization of AD models remains largely dependent on traditional digital design aid tools, such as Computer-Aided Design (CAD) or Building Information Modelling (BIM). The typical interaction with architectural 3D models in these tools entails a mouse-based manipulation of 3D geometry, where the user manipulates the design through a collection of windows showing the geometry from various perspectives. This forces architects not only to combine the separate views to form a mental model of the entire scenario [11], but also to scale it to real size in their imagination. Although architects are known for their 3D visualization capabilities, therefore being more than able to work within these boundaries, their clients, on the other

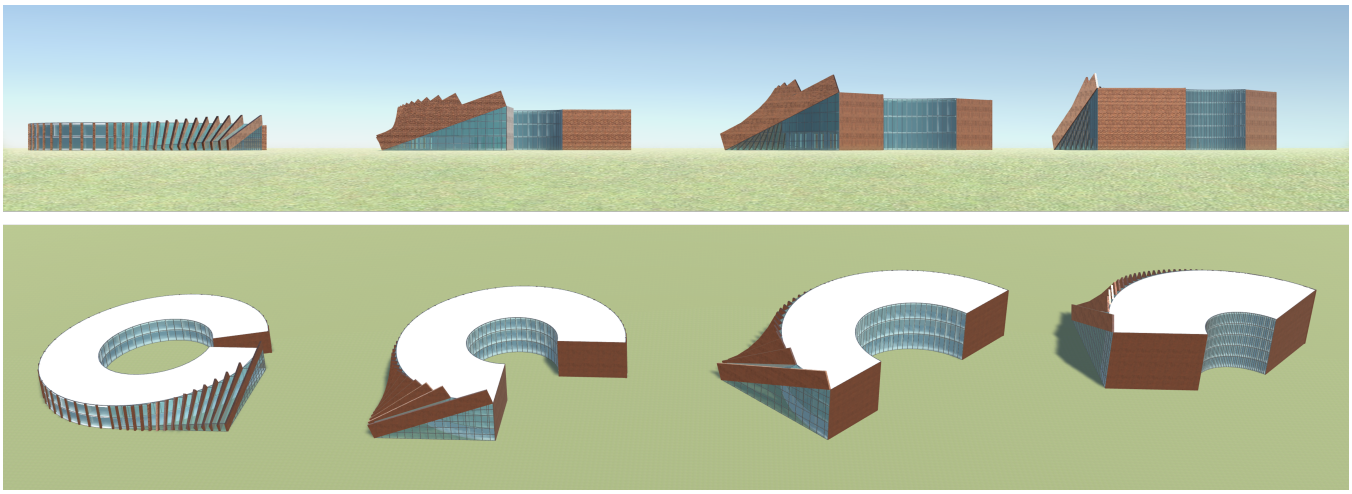

Figure 2: Isenberg School of Management Hub adaptation: four possible variations of the model shown in Unity.

hand, are not. Particularly, the far-fetched design method presented by AD, combined with the design complexity it allows, demand better visualization mechanisms.

To overcome the limitations of the computer screen, traditional architectural design processes heavily rely on the continuous production of physical models. This activity has been at the heart of the architectural practice for centuries, since it provides the dimensionally required to assess the quality of a design solution [9, 21]. Unfortunately, the flexible nature of AD renders it impossible for architects to produce physical models for every instance of the design space, which means we require an alternative.

Virtual Reality (VR) presents itself a fast and cheap substitute to physical model making [21], while guaranteeing not only the tactile interaction offered by its analog counterpart but also the scalability factor. VR has the potential to bring the currently existing AD workflow closer to the traditional model manipulation techniques architects are so familiar with.

1.2 Immediate Feedback

Designing a building through code is not a simple task. There is a disconnection between the idealized geometry and the required programming to generate it, which means architects find it hard to understand if their scripting is progressing in the right direction without constant verification of the algorithm's outcome. For this reason, AD approaches tend to be interactive processes, where the programmer creates geometry incrementally, all the while expecting immediate visual feedback on the impact of the changes the programmer is performing.

Having this in mind, a Live Coding in Virtual Reality (LCVR) approach was developed, in the past, which joined these two realities, AD and VR, in a real-time and interactive design process [5]. LCVR proposes the integration of Live Coding (LC), which is a creativity technique centered upon the writing of interactive programs on the fly [18, 19], in Virtual Environments (VEs), using an AD workflow. The proposed methodology entails the use of an AD tool integrated in a VE, where architects can code their design intents in a flexible and parametric manner. Architects are immersed in the VE, where their designs are being concurrently updated in accordance with the changes made to the algorithmic descriptions. In practice, this constitutes a LC mechanism for architects to edit algorithmic descriptions of their models while immersed in them.

1.3 Interaction Mechanisms

AD assumes that the architect is developing programs in a textual programming interface throughout, ergo, quite a lot of typing is required for this activity. Despite the advantages of immersion, writing code in VR is a challenge. Even the most efficient typing solution is still no match for the typing speed one can achieve on a normal keyboard outside the VE [5], particularly for non-experienced typists, who heavily rely on both visual and haptic feedback. This question is particularly relevant in the present context, as the majority of programming architects are, in fact, non-experienced typists. This suggests the interaction mechanisms at play in the LCVR workflow are still flawed and may deter architects from its use.

This paper proposes to aid the LCVR workflow with Program Comprehension (PC) mechanisms, such as program-model traceability, call chain exploration, and refactoring. PC is in itself an important aid to the AD process alone. Coupled with VR, it has the potential to improve the way architects currently interact with their models. This research also motivates a discussion on different interfaces and code input mechanisms that could offer a more natural experience in the context of VR.

The following sections detail the state of the art on the use of VR in architecture, the implementation details of the LCVR workflow along with the limitations identified in a preliminary study performed with a small number of users, and finally the PC mechanisms explored to improve upon this workflow.

2 VR IN ARCHITECTURE

Throughout the years, multiple authors have studied the benefits and challenges of the use of VR in the Architecture, Engineering,

and Construction (AEC) industries [15, 17]. The immersive properties of VR, such as stereoscopic views and representations at full body scale, facilitate the essential perceptions of solid, void, navigation, proportion, and function [7, 17, 22], as well as the capacity to detect errors or unresolved issues in the design [7]. This is especially important when communicating the design to clients and other team members, whose spatial perception might not be as trained as an architect's.

The development of an architectural project involves the collaboration between several stakeholders. However, arranging meetings with all entities involved is not a trivial task as physical gathering might not always be possible. Collaborative Virtual Environments (CVEs), a computer-based distributed virtual space or a set of places where people can meet and interact with others, with agents or with virtual objects [20], can ease this task by joining remote participants in the same VR experience, e.g., multiple collaborators can be inside a virtual model of a building, simultaneously interacting with the building and with each other. This is fundamental for design studios lacking a collective ideation space and allows design teams to engage in long distance collaborations [8].

Virtual Reality Aided Design (VRAD) can be defined as a CAD technique that uses VR methods [6]. This way, architects can benefit from the visualization aspects of VR without having to break the visualization workflow when they want to make adjustments to the design. Experiments with highly interactive VEs indicated that fun and exploratory factors positively contributed to creative thinking [12], which is a fundamental part of the architectural design process.

3 LIVE CODING IN VIRTUAL REALITY

Considering the advantages VR presents to the architectural practice, a LCVR approach was developed, in the past, which joined AD and VR in a real-time and interactive design process [5].

Figure 3 presents a scheme of the proposed workflow, combining (1) the Traditional Algorithmic Design Coding (TADC) workflow, where architects code their designs still outside the VE, (2) the LCVR workflow, where architects are fully immersed in their creations, and (3) a hybrid approach, which consists in switching back and forth between the two as architects see fit for the design processes. For either approach we can also see the tools used for this implementation, as well as where they intersect.

3.1 LCVR vs TADC

Both the TADC and the LCVR approach imply the use of an Interactive Development Environment (IDE) to input the algorithmic descriptions into the AD tool, which is then responsible for controlling the visualizer. In this case, the combination Julia+Atom+Juno was used for the IDE, while the AD tool chosen was Khepri [13]. The visualizer is Unity, a Game Engine (GE) that not only allows for a fast update of model changes but also for the connection to VR.

In either case, designers conduct an iterative process of programming the algorithmic description followed by the visualization of the generated architectural model, enhancing the project with each iteration. The main difference in the LCVR process is that designers are now inside the VE created by the GE, programming from there. In the VE, the model is being updated just as it would in the GE's normal interface, with the added interactivity to the experience. Designers can access the IDE from the VE, using the virtual desktop application, which allows them to edit the algorithmic descriptions of the models as the buildings change around them.

As for collecting text input for code modification, most strategies for VR are based on either a physical or virtual keyboard. Virtual keyboards are simulated on the VE and key pressing can be achieved by using a controller as a tool, i.e., a drum-like interface [3], or using bare-hand tracking devices with finger tracking. Physical keyboard approaches collect the input from a real keyboard which provides more haptic feedback and a smoother learning curve, leading to better typing efficiencies than alternative approaches, as previously studied in [5].

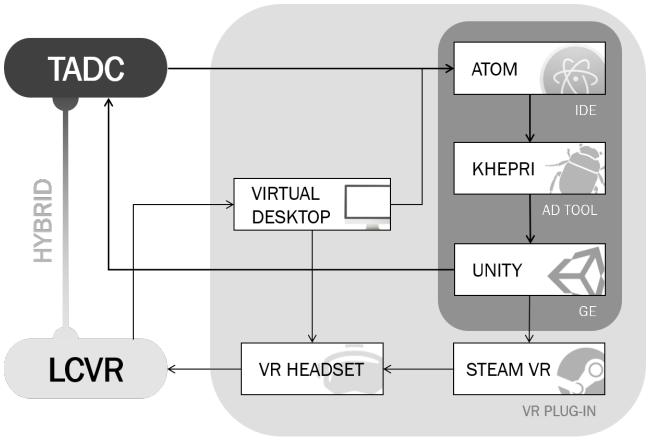

Figure 3: The TADC workflow highlighted in dark grey, the LCVR workflow in light grey, and the hybrid approach connecting them.

3.2 Preliminary Study

The proposition still considers the use of the TADC approach for two simple reasons: first, the physical discomfort caused by the continuous use of a Head-Mounted Display (HMD) and, second, the performance degradation that occurs when transitioning to LCVR. Despite the advantages of immersion, writing code in VR has proven to be a challenge.

A preliminary study was conducted on 10 subjects, 6 of which were architects with programming experience, i.e architectural practitioners trained in AD, and 4 were computer science Msc students. Users were asked to code a very simple exercise using LCVR: a random pagoda city. The exercise was intentionally basic, in order to last no longer than 30 minutes. Figure 4 shows the view our users had if they completed the exercise, on the left, and the user setup, on the right. The experiment ended with a survey, whose answers are summed up in Figure 5.

Users were asked if the LCVR experience was more fun, slower, more productive, and more advantageous than TADC. The one to five scale in the graph corresponds to a Likert scale ranging from full disagreement to full agreement. The voting average for the architectural subjects was a lot more pessimistic than their counterparts. Architects replied with almost full agreement that coding in

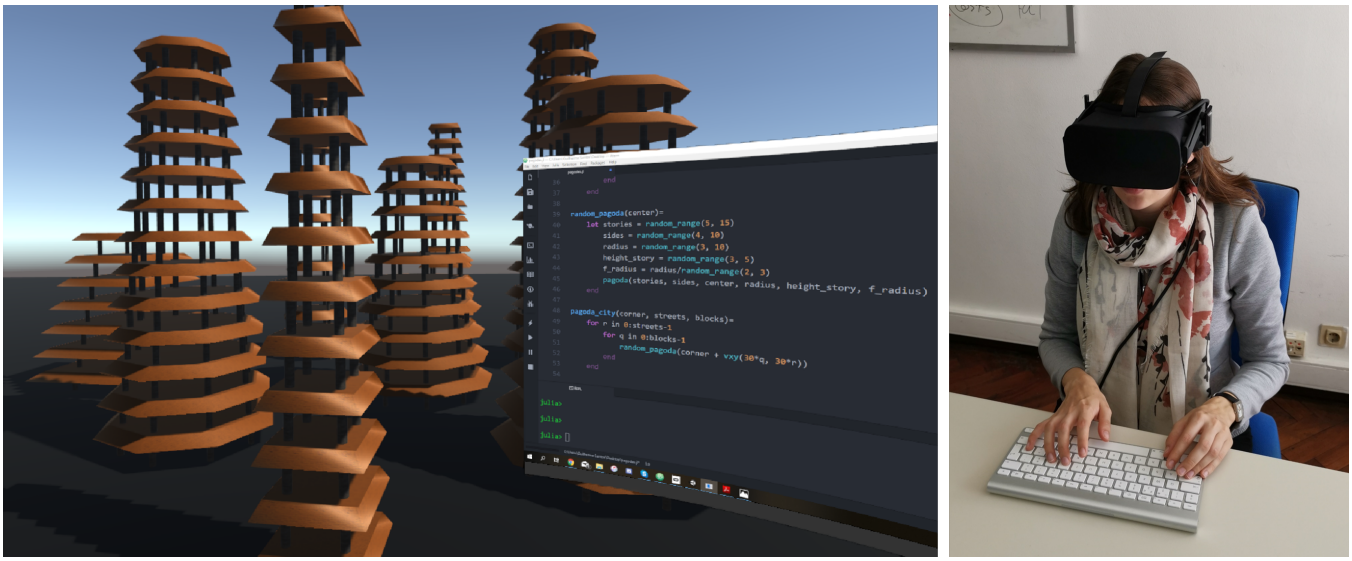

Figure 4: LCVR preliminary user test: random pagoda city exercise.

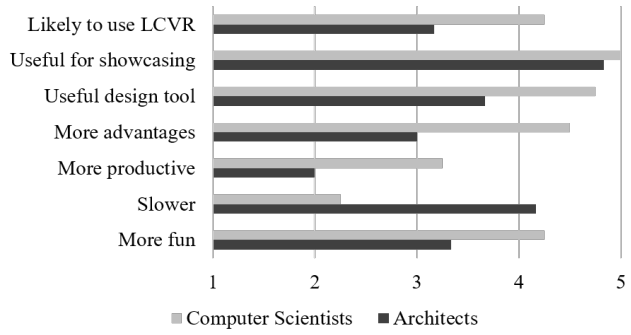

Figure 5: User study survey results: averaged answers for the questions comparing the LCVR to the TADC approach.

LCVR is slower, while computer science students mostly disagreed. The latter also voted for more fun, while architects remained fairly close to neutral on average, on account of the typing difficulty, according to the comments left on the open question. Architects also voted LCVR as less productive and less advantageous than TADC. However, they agreed on the usefulness of the workflow for showcasing, and also for design, although less enthusiastically. Only a few stated they would be likely to use LCVR in their own design process, at least as it is currently presented. Comments were also issued on the added difficulty in reading the program's text in VR for poor-sighted users.

3.3 Limitations

Despite the small number of participants involved, this preliminary study already allowed us to verify a set of issues in the LCVR workflow. The short survey revealed that the group of users with an architectural background, despite their familiarity with programming tools, struggled to use them in the VE. If programming architects found it difficult to manage the workflow, we can only assume less experienced programmers will find even less motivation to engage in LCVR.

Circling back to the two issues presented previously - physical discomfort and typing performance – we believe they will soon be solved. On the one hand, future generations of architects will be more experienced typists on account of the impact computers are having in younger generations by the day. On the other, alternative input mechanisms, such as voice recognition, may evolve to a point where we need not type as much as we currently do. In the meantime, however, and in favor of practitioners currently in the market, we may ease the programming task for architects in VR through other means, namely PC.

4 PROGRAM COMPREHENSION

One of the major difficulties architects find in understanding and modifying AD programs is tied to the correlation between code and model, that is, figuring out which parts of the code are responsible for generating which parts of the model. This process may include scrolling up and down the source code or searching for the names of relevant functions or parameters. This search-and-change process is aggravated if the parcels of code requiring modifications were developed by a third party, which is a rather common scenario in the collaborative work that characterizes the development of architectural projects. Understanding which code fragments generate which parts of the model is not a simple task, particularly, when the user is not familiar with the code, or when the code is complex enough for the developer to get lost in it.

However, architectural programs are built on the type of data that simplifies the establishment of correlations. Traceability allow us to retrieve the relationships between code and generated shapes. This means we could simply identify the shape we wish to

change (by grabbing it or clicking on it) and be guided towards the corresponding code parcel, thus considerably helping the PC task.

Besides the identification of the code one wishes to modify, the modification itself may also be challenging. Writing programs from scratch, even with a well-defined plan, can result in poorly organized code structures. The nature of AD programs, in particular, motivates the creation of messy code as architects typically iterate multiple times over an idea until they reach a shape with which they are satisfied. The situation is aggravated when performing changes in programs originally written by someone else. To help ensure that these changes do not introduce bugs in the AD programs, we propose the use of refactoring. The following sections discuss these PC mechanisms.

4.1 Traceability

Traceability entails the identification of which parts of the model correspond to which parts of the program, and/or vice versa [14]. Traceability reveals the correlation that exists between program and generated output and is crucial to understand, maintain, and debug the program.

Grasshopper[1] and Dynamo,[2] for instance, two popular visual programming environments in architecture, present unidirectional traceability mechanisms that relate code components to model elements. This means users can select any component in the IDE and the shapes generated by that component will automatically be highlighted in the model. Rosetta [14] and Luna Moth [1], two textual programming tools, as well as Khepri, the AD tool used for the LCVR implementation, support bi-directional traceability, which means users may select either code fragments or model parts and the tool will highlight the corresponding part in the opposite end. Khepri's traceability is illustrated in Figure 6.

Traceability aids PC, maintenance, and debugging but it depends on the ability to select model parts which, in the context of VR, can not be done with the typical mouse-based selection mechanisms employed by non-VR applications. The following section provides some insight on how to address this issue.

4.2 Interaction Mechanisms

With the addition of a third dimension to the virtual space, the human-computer interactions used in two dimensional spaces are no longer suitable. Therefore, new interaction techniques for VR have been developed throughout the years, exploring different input modalities.

Some of the most used techniques for geometry selection within a fully immersive VE are based on hand gestures and pointing [2]. Hand gestures provide a direct mapping between physical and virtual hands. The range of selection is limited to the arm's extent when working at full scale, which can restrict the quantity of geometry that can be selected in large models. However, this limitation can be mitigated by rescaling the model, though the perception of the real dimensions of the building can be affected. Hand gestures are also more physically demanding and require more dexterity.

[1]https://www.grasshopper3d.com
[2]https://dynamobim.org

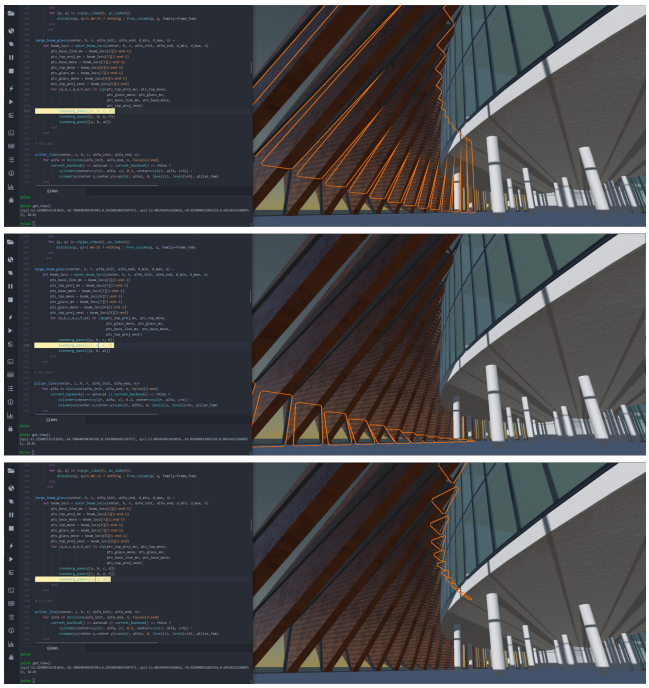

Figure 6: Khepri's traceabilty module working on the Isenberg case study.

Pointing techniques, on the other hand, remove the physical limitations by projecting a ray that intersects the geometry to be selected. The ray can have its origin on the hand or eye, e.g., the ray-casting and occlusion selection techniques, respectively. Despite requiring considerably less hand effort, they do not remove the dexterity issue. The selection of small or distant geometries can still be a challenge, particularly when the user is not able to maintain a steady hand during selection. The action confirmation task is also likely to make a change in the user's hand orientation, which can lead to a wrong selection.

The implementation of the workflow for this paper was conducted in a HTC VIVE headset. We implemented a ray-casting technique, triggered by pressing a button on the controller. Both hands perform the same action, leaving the choice up to the user's dominant hand. The highlights, both on the geometry and the code, provide useful feedback for the interaction.

4.3 Call Chain

Traceability allows us to find the operations that, ultimately, generated the model part we wish to change. However, the part of the code that we must modify is likely to be deeper down the call chain. This means we also require a mechanism to navigate the sequence of calls until the parcel of code responsible for the error is found.

From the AD tools mentioned earlier, only Rosetta [14] and Khepri provide this feature. Figure 7 presents an example: a column of the Isenberg case study model was selected in Unity, and the corresponding function call was highlighted in Atom. However, this call provides little information on the column's conception

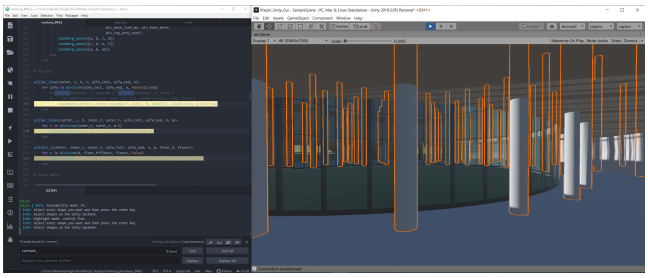

Figure 7: Khepri's traceabilty module showing the call chain of the columns' invocation (intensity of the highlight color increases as we move further away from the first call).

logic. We thus move through the relevant function calls, whose highlight gets progressively stronger as we close in on the end of the call chain.

4.4 Refactoring

While the previously explained features are meant to aid the comprehension task alone, we now introduce one that focuses on the code modification task. This activity is a source of errors and inconsistencies, which users frequently introduce in their program while trying to solve other issues.

Refactoring is commonly defined as the process of improving the structural integrity or performance of existing programs without changing their external behavior [10]. In order to guarantee overall consistency in the changes introduced, most refactoring tools do the code manipulation work themselves. In the case of AD changes, this means that, by using refactoring whenever possible, our users would have to write less code on average than when applying changes manually. More importantly, they would avoid introducing bugs.

Figure 8 presents an example of Khepri's refactoring capabilities. The initial approach to the modeling of the slabs in this case study considered a function that generated a C-shaped slab. The function responsible for generating all building slabs then called the previous function for each building floor. Later on, however, as the model approached the real case study in its details, we decided to introduce an exception: the base slab should have a slightly different shape in order to meet the domino-shaped laminas that compose the building's façade.

Listings 1 and 2 present the two phases resulting from this process. The introduction of the conditional expression in the slab function imposed an additional parameter on that function. Khepri's refactoring mechanisms automatically adjusted all function calls affected by this change, guaranteeing that the program keeps on running as expected despite the modification. The change in the slabs function's for-loop could also be automated by a partial loop unrolling, since all we did was extract and modify the first instance of the loop.

5 DISCUSSION

In order to combine the LCVR approach with the aforementioned features we extended the methodology so that users programming

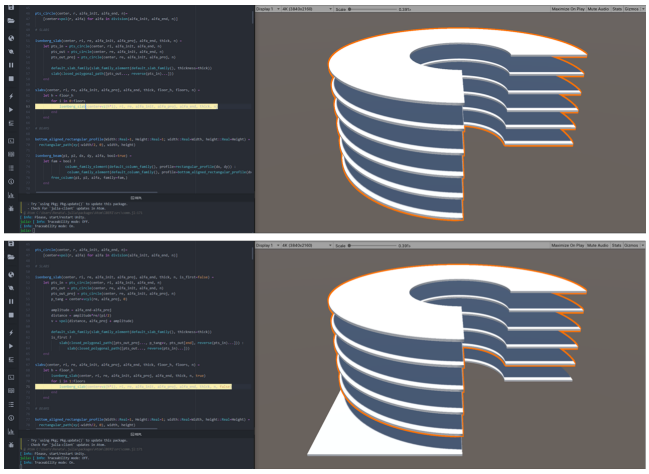

Figure 8: Slab function refactoring.

their AD models in the VE can benefit from PC and program maintenance mechanisms. Traceability allows for a faster and more intuitive way to find program-model correlations. Call chains map this feature to the entire call sequence, which means users can trace their shapes back to the lowest abstraction level, where the changes are most likely to occur. Finally, refactoring mechanisms help architects apply changes to their programs with less typing required and, more importantly, with a consistency guarantee.

Listing 1: Slab functions before refactoring

```
function isenberg_slab(c, r_i, r_e, α0, α_p, α_e, n)
  ps_i = ps_circle(c, r_i, α0, α_e, n)
  ps_e = ps_circle(c, r_e, α0, α_e, n)
  ps_p = ps_circle(c, r_e, α0, α_p, n)
  ps = [ps_e..., reverse(ps_i)...]

  slab(closed_polygonal_path(ps))
end

function slabs(c, r_i, r_e, α0, α_p,α_e, f_h, floors, n)
  for i in 0:floors
    isenberg_slab(c + vz(f_h*i), r_i, r_e, α0, α_p, α_e, n)
  end
end
```

Listing 2: Slab functions after refactoring

```
function isenberg_slab(c, r_i, r_e, α0, α_p, α_e, n, base)
  ps_i = ps_circle(c, r_i, α0, α_e, n)
  s_e = ps_circle(c, r_e, α0, α_e, n)
  ps_p = ps_circle(c, r_e, α0, α_p, n)
  p_t = c + vcyl(r_e, α_p, 0)
  Δα = α_e - α_p
  v = vpol(Δα*2r_e/π, α_p + Δα)
  ps = base ?
    [ps_p..., p_t + v, ps_e[end], reverse(ps_i)...]
    [ps_e..., reverse(ps_i)...]

  slab(closed_polygonal_path(ps))
end

function slabs(c, r_i, r_e, α0, α_p, α_e, f_h, floors, n)
  isenberg_slab(c, r_i, r_e, α0, α_p, α_e, n, true)
  for i in 1:floors
    isenberg_slab(c + vz(f_h*i), r_i, r_e, α0, α_p, α_e, n, false)
  end
end
```

5.1 Live Coding in Virtual Reality

Figure 9 presents an example of the workflow. While immersed in the VE, the architect decides to improve upon the design by changing the model. For this case specifically, by changing the parameters that define the large domino beams of the façade. The chosen object is selected in the model, using the VR controllers, and the corresponding fragment of code is highlighted in the editor. The architect may immediately find the function needing modifications, or instead browse through the remaining highlights until the function is located deeper down the call chain.

In this case, a simple modification was applied to enlarge the domino beams, changing their maximum width from 3m (in the first shot) to 6m (in the last one). In case a more complex change is required, refactoring operations become extremely useful, as they overcome several of the difficulties of making extensive program changes while immersed in the VE.

5.2 Interfaces

As mentioned in Section 3.3, coding textual programs in VR raises several issues, particularly regarding the use of keyboards. The solution presented here relied on PC mechanisms to aid the programming task, namely traceability and refactoring. Another feature worth looking into is autocompletion. Tools like Intellisense[3] for Visual Studio Code offer code completion, assistance and hinting for various programming languages, significantly reducing the typing required. Dynamic completion goes even further, guessing any sort of text completion based on probabilistic models of the user's typical writing patterns. With powerful-enough mechanisms of this nature integrated in the workflow we can envision a scenario where users almost do not need to code, they only need to accept the suggested options.

Considering code input mechanisms, other options are also currently available. Hand-written code recognition, for instance, would eliminate the need to visualize any physical apparatus in the VE. However, the scale of the written characters in VR must be considered. Humans are considerably faster when writing symbols in smaller scales, such as paper sheets, as opposed to whiteboards, for instance. However, handwriting code in VR, as hardware stands today, would have to rely on larger displacements of the writing instruments for the sensors to detect the motion, which might ultimately defeat the performance purpose.

Voice input is another viable option, and possibly one of the most comfortable solutions for the user. Since Tavis Rudd presented his system to dictate code in 2013[4], a wave of vocal programming tools has been flooding the market with better voice coding solutions. Nevertheless, we can foresee a series of obstacles. Primarily, current technology has low voice-recognition accuracy in loud environments, which means expensive equipment might be required when working collaboratively. The same mechanisms may also reveal intrusive for people in the same workspace not directly involved in the activity. Finally, and more specific to the problem at hand, dictating coding commands is not a trivial task and, thus, requires training.

As for solutions that attempt at reducing the amount of coding required, we can imagine an intermediate interface: a solution somewhere between direct manipulation and having the entire code showcased in VR. As a building design evolves, it is expectable, that, in later phases of the design process, the majority of changes that architects want to apply are related to parameter values only (sizes, materials, etc). These sort of changes are not difficult to support in VR in a more user-friendly manner, e.g., with sliders and toggles popping up in the VE upon selection of geometry. In this sense, users would only need to interact with the entire AD program for more profound changes to the structure of the program, witch they may wish to perform outside the VE anyway.

In the end, multimodal approaches may hold the answer, i.e., taking advantage of multiple input modalities simultaneously. Or perhaps we can be as bold as to imagine the integration of brain-computer interfaces, which are becoming ever more sophisticated [16]. We may soon enough arrive at a point where users can just

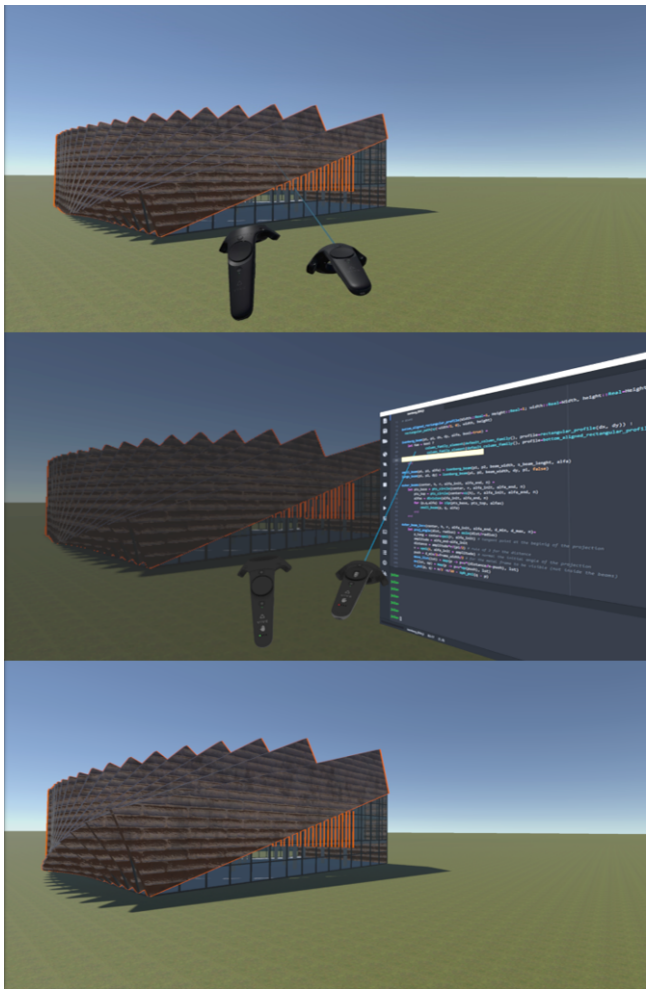

Figure 9: LCVR methodology applied to the Isenberg case study, with traceability mechanisms at play.

[3] https://code.visualstudio.com/docs/editor/intellisense
[4] https://pyvideo.org/pycon-us-2013/using-python-to-code-by-voice

think of the actions they wish to perform and the computer will interpret their electroencephalograph signals and do the rest.

6 CONCLUSION AND FUTURE WORK

Live Coding in Virtual Reality (LCVR) is a methodology for developing Algorithmic Design (AD) programs describing architectural 3D models while immersed in a virtual representation of those models. While doing LCVR, architects are inside the Virtual Environment (VE), changing the algorithmic descriptions of their models and, thus, making the buildings change around them.

In this paper, we proposed to augment the LCVR approach with Program Comprehension (PC) and manipulation mechanisms, namely traceability and refactoring, in order to ease the programming task for architects who wish to live code in Virtual Reality (VR). This not only aids the comprehension task and makes it easier to modify programs without introducing errors but also reduces the typing and scrolling required in the process.

As a result, we expect to remove one of the largest obstacles to the adoption of the LCVR approach in architectural studios, so that architects can truly benefit from the advantages of VR in their AD processes.

The preliminary study presented constitutes a very early evaluation of this research. In the future, we plan on repeating the evaluation on a larger pool of users, now focusing on later stages of the design process where we believe LCVR is most beneficial. This could be done by offering users an already developed AD program and asking them to modify it. The experiment would allow us to assess the value of the PC mechanisms implemented in helping architects understand AD programs developed by others.

ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017.

REFERENCES

[1] Pedro Alfaiate, Inês Caetano, and António Leitão. 2017. Luna Moth: Supporting Creativity in the Cloud. In *Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*. 72–81.

[2] Ferran Argelaguet and Carlos Andujar. 2013. A survey of 3D object selection techniques for virutal environments. *Computers & Graphics* 37, 3 (2013), 121–136. https://doi.org/10.1016/j.cag.2012.12.003

[3] Costas Boletsis and Stian Kongsvik. 2019. Text Input in Virtual Reality: A Preliminary Evaluation of the Drum-Like VR Keyboard. *Technologies* 7, 2 (2019), 31.

[4] Mark Burry. 2011. *Scripting cultures: Architectural design and programming.* John Wiley & Sons.

[5] Renata Castelo-Branco, António Leitão, and Guilherme Santos. 2019. Immersive Algorithmic Design: Live Coding in Virtual Reality. In *Architecture in the Age of the 4th Industrial Revolution: Proceedings of the 37th Education and research in Computer Aided Architectural Design in Europe (eCAADe) Conference*, José Pedro Sousa, Gonçalo Castro Henriques, and João Pedro Xavier (Eds.), Vol. 2. University of Porto, Porto, Portugal, 455 – 464.

[6] Dirk Donath and Holger Regenbrecht. 1995. VRAD (Virtual Reality Aided Design) in the Early Phases of the Architectural Design Process. In *Proceedings of the 6th International Conference on Computer-Aided Architectural Design Futures (CAAD Futures)*. 313–322.

[7] Tomás Dorta, Gôkçe Kinayoglu, and Sana Boudhraâ. 2016. A new representational ecosystem for design teaching in the studio. *Design Studies* 47 (2016), 164 – 186. https://doi.org/10.1016/j.destud.2016.09.003

[8] Tomás Dorta, Annemarie Lesage, Edgar Pérez, and JM Christian Bastien. 2011. Signs of collaborative ideation and the hybrid ideation space. In *Design Creativity*, Toshiharu Taura and Yukari Nagai (Eds.). Springer, 199–206. https://doi.org/10.1007/978-0-85729-224-7_26

[9] N. Dunn. 2014. *Architectural Modelmaking.* Laurence King Publishing. https://books.google.pt/books?id=2yN2oAEACAAJ

[10] Martin Fowler, Kent Beck, Don Roberts, and Erich Gamma. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

[11] Enrico Gobbetti and Riccardo Scateni. 1998. Virtual reality: past, present and future. *Studies in health technology and informatics* 58 (1998), 3–20. https://doi.org/10.3233/978-1-60750-902-8-3

[12] Iris Graessler and Patrick Taplick. 2019. Supporting Creativity with Virtual Reality Technology. *Proceedings of the Design Society: International Conference on Engineering Design* (2019), 2011–2020.

[13] António Leitão, Renata Castelo-Branco, and Guilherme Santos. 2019. Game of Renders: The Use of Game Engines for Architectural Visualization. In *Intelligent & Informed: Proceedings of the 24th Annual Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA) Conference*, Matthias Hank Haeusler, Marc Aurel Schnabel, and Tomohiro Fukuda (Eds.), Vol. 1. Victoria University of Wellington, Wellington, New Zealand, 655 – 664.

[14] António Leitão, Luís Santos, and José Lopes. 2012. Programming Languages for Generative Design: A Comparative Study. *International Journal of Architectural Computing* 10, 1 (2012), 139–162. https://doi.org/10.1260/1478-0771.10.1.139

[15] Julie Milovanovic, Guillaume Moreau, Daniel Siret, and Francis Miguet. 2017. Virtual and Augmented Reality in Architectural Design and Education: An Immersive Multimodal Platform to Support Architectural Pedagogy. In *Proceedings of the 17th International Conference on Computer-Aided Architectural Design Futures (CAAD Futures)*. 513–532.

[16] Luis Fernando Nicolas-Alonso and Jaime Gomez-Gil. 2012. Brain Computer Interfaces, a Review. *Sensors* 12, 2 (2012), 1211–1279. https://doi.org/10.3390/s120201211

[17] Michelle Portman, Asya Natapov, and Dafna Fisher-Gewirtzman. 2015. To go where no man has gone before: Virtual reality in architecture, landscape architecture and environmental planning. *Computers, Environment and Urban Systems* 54 (2015), 376–384.

[18] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *Programming Journal* 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[19] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *Programming Journal* 3, 1 (2018), 3.

[20] Dave Snowdon, Elizabeth F. Churchill, and Alan J. Munro. 2001. *Collaborative Virtual Environments: Digital Spaces and Places for CSCW: An Introduction.* Springer London.

[21] Milena Stavric, Predrag Sidanin, and Bojan Tepavcevic. 2013. *Architectural Scale Models in the Digital Age: design, representation and manufacturing.* Birkhäuser. https://books.google.pt/books?id=Il5SDgAAQBAJ

[22] Xiangyu Wang. 2007. Mutually augmented virtual environments for architectural design and collaboration. In *Proceedings of the Computer-Aided Architectural Design Futures conference (CAAD Futures)*. Springer, 17–29. https://doi.org/10.1007/978-1-4020-6528-6_2

[23] Robert Woodbury. 2010. *Elements of Parametric Design.* Routledge.