BIM-LLM: A Virtual Assistant for Architectural Design in a VR Environment

Ander Salaberria[1], Oier Ijurco[1], Markel Ferro[1], Jiayuan Wang[1], Iñigo Vilá Muñoz[1], Roberto de Ioris[2], Jeremy Barnes[1], Oier Lopez de Lacalle[1]

[1]HiTZ Center, University of the Basque Country (UPV/EHU), [2]Vection Technologies
Correspondence: ander.salaberria@ehu.eus

Abstract

Architectural design relies on 3D modeling procedures, generally carried out in Building Information Modeling (BIM) formats. In this setting, architects and designers collaborate on building designs, iterating over many possible versions until a final design is agreed upon. However, this iteration is complicated by the fact that any changes need to be implemented manually, lengthening the design process and making it difficult to quickly prototype changes.

In order to improve this, we propose BIM-LLM, a virtual assistant that allows for quickly prototyping changes in a virtual reality (VR) environment. This framework allows users to make changes directly in the VR environemnt via voice commands, which are mapped to executable Python code grounded in the user's environmental context, specifically within a 3D BIM environment. This approach allows for real-time, contextualized guidance through natural language dialogue, allowing for a faster and more natural prototyping experience.

Our demo's video is available at this link and the code is publicly available here.

1 Introduction

The integration of Natural Language Processing (NLP) approaches within Virtual Reality (VR) systems has been an active research direction for many years now (Everett et al., 1997, 1999; Giunchi et al., 2024). However, the more recent developments in Large Language Models (LLMs) (Radford and Narasimhan, 2018; Brown et al., 2020) and their ability to generate and understand more complex language has lead to an increased interest in exploiting LLMs inside of VR environments, either as virtual assistants (Wu et al., 2023) or as virtual tutors (Ward et al., 2025).

One such domain that would benefit from the integration of LLM-based assistants is architectural design review. Architects and structural engineers design 3D representations of buildings and other assets in Building Information Modeling (BIM) format. These BIMs are digital files that contain detailed information about buildings or infrastructure, which can be shared, accessed, or connected across networks to help architects, structural engineers, or contractors make informed decisions about constructed facilities and assets (Sacks et al., 2019).

However, interacting with BIM environments usually demands specialized knowledge and familiarity with sophisticated software, making it challenging for non-experts and hindering real-time collaboration during design reviews. Additionally, conventional interfaces often fail to offer the intuitive, context-aware interactions needed for effectively navigating or adjusting intricate architectural components in 3D spaces. In such cases, incorporating an LLM assistant that is able to answer queries and make changes directly in the VR scenario would therefore allow for quicker prototyping and streamline the design process.

As a first step in this direction, we propose **BIM-LLM**, the first LLM assistant that allows a BIM user to make changes directly in a VR environment via voice commands. This system enables an LLM to reason over spatial relationships and perform multi-step operations, such as object manipulation, changes to object visibility, and camera control all based on natural language input from the user.

In order to enable an LLM assistant to make changes directly in the VR environment, we develop a custom Unreal Engine sandbox, accompanied by a Python API with predefined functions for interacting with assets within the building, such as hiding/revealing doors, changing a wall's color, or rotating stairs. We use a speech-to-text system to capture the voice commands and incorporate an LLM module to convert the command to executable code within the API. After execution, the changes are made visible directly in the VR sandbox, allowing the user to more quickly prototype changes, even if they are not a proficient BIM user.

We furthermore develop a multi-step human evaluation protocol within the VR environment. The results of our evaluation indicate that LLMs are able to generate code to modify the environment correctly, specially as models get bigger. However, they still face challenges navigating the complicated nature of BIM environments.

2 Related Work

NLP approaches in VR are often dedicated to enabling VR tutors (García-Méndez et al., 2024; Konenkov et al., 2024). Ward et al. (2025), for example, develop a VR tutor for learning Irish, while Aguirre et al. (2025) develop a VR tutor for VR health and safety training.

Another common application of NLP in VR is the development of VR assistants. Wu et al. (2023) introduce a video-grounded task-oriented dialogue dataset that captures real-world AI-assisted user scenarios in VR, while Prange et al. (2017) develop a multimodal dialogue system to help doctors make decisions about patient therapy.

Finally, there are also a few efforts to develop avatar-based VR chatbots, either via spoken dialogue (Yamazaki et al., 2023) or sign language (Quandt et al., 2022).

While these VR tutors and assistants represent an interesting step in integrating natural language in VR, they are not actually able to make any changes to the VR environment itself.

BIM manipulation with LLM integration Most previous work on integrating LLMs into BIM manipulation has focused on querying the BIM for information (Zheng and Fischer, 2023; Li et al., 2025). Editing or prototyping changes to the BIM file, however, has not been widely explored. Jang and Lee (2024) propose a pipeline to convert BIM files to XML and then allow users to collaborate with an LLM to make changes in the XML files, before being converted back to BIM. However, these changes are not made via voice commands and are not directly visible to the user in a VR environment.

Instruction-Following Multimodal Agents Recent advances have demonstrated that modern agents can effectively follow instructions in complex multimodal scenarios. A prevalent paradigm in recent years involves leveraging external APIs to augment and extend the agent's action space (Shen et al., 2023). These APIs are often integrated with external models to handle more sophisticated tasks (Yang et al., 2023). Our work is closely related to systems such as ViperGPT (Surís et al., 2023) and VisProg (Gupta and Kembhavi, 2023), which frame visual question answering as a complex task requiring both visual perception and reasoning. These systems utilize code-generation models to orchestrate vision-and-language models into executable subroutines. Specifically, a large language model (LLM) generates a structured plan in the form of code, which specifies a sequence of tools to invoke. The outputs of these tool executions are then aggregated to produce a final response to the query.

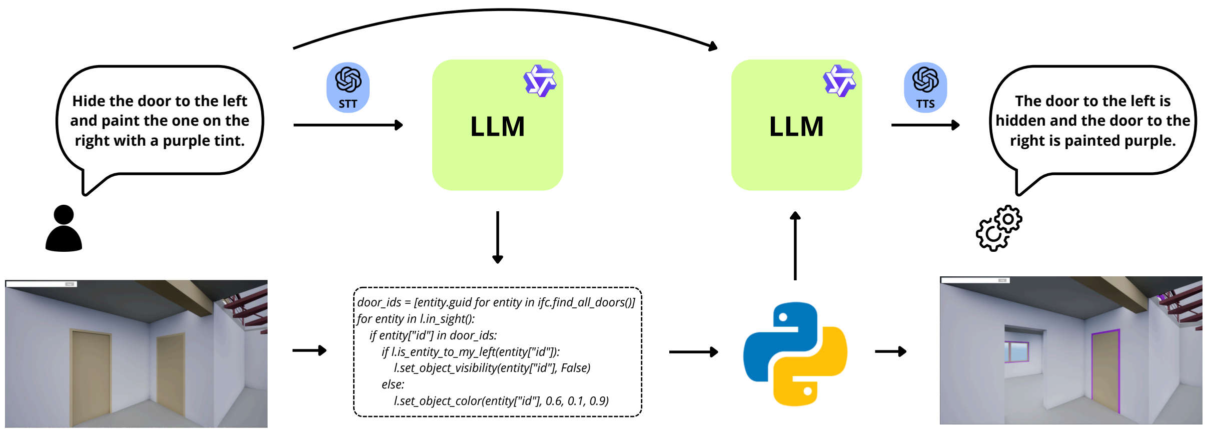
3 System Architecture

BIM-LLM contains **five main components** that together form the pipeline shown in Figure 1. The scene is rendered in a custom Unreal Engine sandbox that loads a BIM file. This sandbox communicates with a Python API that is able to apply changes to the sandbox via predefined functions. These functions allow us to identify objects, interact with them (change color, move, hide, show, etc.), handle the camera, and add/transform/delete props.

The system takes a voice command from the user, which is converted via a speech-to-text system. The resulting text is fed to the LLM module, along with the sandbox's configuration and the custom API, whose objective is to create Python code to fulfill the command. The code is executed and the resulting changes are visualized in the sandbox. Additionally, the output of the program, alongside the initial command, is passed to the LLM module to generate a textual response, which is returned as the output after being vocalized by the text-to-speech module. In this way, the output to the user is completely disentangled from the execution process, avoiding errors and allowing the output to report on runtime errors.

4 Adaptation of the LLM

We employ prompt engineering techniques to adapt the LLM to this specific task-oriented environment. The LLM serves two distinct roles within our system: 1) as a code generator, which is conditioned on the user's input query and aims to make changes in the BIM; and 2) as a feedback generator that provides feedback to the user once the code is exe-


Hide the door to the left and paint the one on the right with a purple tint.
STT
LLM
LLM
TTS
The door to the left is hidden and the door to the right is painted purple.

```
door_ids = [entity.guid for entity in ifc.find_all_doors()]
for entity in l.in_sight():
    if entity["id"] in door_ids:
        if l.is_entity_to_my_left(entity["id"]):
            l.set_object_visibility(entity["id"], False)
        else:
            l.set_object_color(entity["id"], 0.6, 0.1, 0.9)
```

Figure 1: BIM-LLM pipeline. Given the scene's configuration and an instruction, the LLM generates the code to modify the scene. This code is then executed, and the output is passed to the LLM along with the instruction to generate a response.

cuted.

Each role necessitates a distinct prompt design, as the complexity and requirements of the tasks differ. In particular, the code generation task is more demanding and therefore incorporates richer contextual information in the prompt. Specifically, we include:

- **Task definition:** A description of the task at hand, specifying what information is available and the output format.
- **API documentation:** The simplified documentation of the API, which defines the available Python functions and classes.
- **Initial State of the BIM:** The initial configuration of the environment where the generated code will be executed.
- **Few-shot examples:** A set of seven query-code pair examples, which demonstrate correct API usage and guide the model toward generating code in the desired format.
- **User query:** The specific input provided by the user.

In contrast, the feedback generation prompt requires less contextualization, as the task is comparatively straightforward. This prompt includes:

- **Task definition:** As above, specifying the nature of the task and available inputs.
- **User query:** The query that led to the code execution.
- **Execution outcome:** An indicator of whether the generated code executed successfully.

Both prompts can be found in Appendix A.

5 Demonstrator evaluation

The following link contains a video showcasing a short summary of the BIM-LLM system with a couple of running examples: https://youtu.be/AaOZ3rstU0Y

To assess the capabilities of the proposed system, we run an extensive manual evaluation of the system's performance. This section is dedicated to this evaluation and its analysis.

5.1 Evaluation Settings

The evaluation of BIM-LLM focuses on assessing the system's ability to accurately interpret and execute natural language commands via code generation, as well as to reflect changes in a VR environment using an API. The evaluation was designed to measure both the technical accuracy of the generated code and the semantic correctness of the generated responses.

Models. We evaluate four variants of Qwen2.5 instruction-tuned language models (Qwen et al., 2025) to explore the effect of model scale on performance, as higher capacity usually increases model performance at the expense of execution-time and higher infrastructure needs. These include the 1.5B, 7B, 32B and 72B parameter versions.

Dataset. The evaluation dataset consists of 60 unique instances, tailored to test the different functionalities that the current Python API offers. These

Model	Error Type ↓		Accuracy ↑
	Runtime	Semantic	
Qwen2.5-1.5B	40.0	**43.3**	17.7
Qwen2.5-7B	26.7	49.6	23.7
Qwen2.5-32B	20.0	44.3	35.7
Qwen2.5-72B	**11.7**	50.6	**37.7**

Table 1: Percentage of instances that were correctly completed. Incorrect instances are separated between the ones that failed to finish the execution (runtime) and the ones with incorrect outcomes (semantic).

Model	Error Type ↓			
	No Change	Incorrect	Collateral	Other
Qwen2.5-1.5B	53.3	**14.3**	5.7	9.0
Qwen2.5-7B	45.7	24.7	2.3	**4.3**
Qwen2.5-32B	29.7	27.0	2.3	5.3
Qwen2.5-72B	**21.0**	35.7	**1.0**	**4.3**

Table 2: Distribution of type of errors made by each model due to either runtime or semantic errors. Percentages are computed considering all instances to maintain consistency across models.

include queries that affect visibility, coloring, transformation, and removal of BIM entities, as well as camera transformations and prop additions. Moreover, these instances can go from basic atomic instructions (e.g., "Remove the left door.") to more complex and even composite queries (e.g., "Look at the window and color the wall around it in white."), testing the LLM's capability of reasoning via code generation to their limits. These instances are evenly split between two different BIM environments of varying complexity: a house and a school. The former is a simple building containing around 200 entities, whereas the latter is composed of almost 6,000, allowing us to measure how the complexity of the BIM can affect the completion of the instructions.

Evaluation metrics. In this work, we focus on human evaluation. We automatically detect runtime errors in the generated code, but we require manual annotation to distinguish between correct and incorrect outcomes of the queries, as well as to classify between 4 error types. These error categories reflect different model behaviors after executing code conditioned on an instruction:

- *No Change*: the model failed to produce any change.
- *Incorrect*: the model edited the target object(s) incorrectly or left some without editing.
- *Collateral*: the model edited the target object(s) along with unrelated ones.
- *Other*: the model altered unrelated objects, but not the specified one(s).

In total, six human annotators took part in the annotation process. Each of the first four reviewed the generations for one specific model, while the remaining two annotated half of the annotations (with balanced sampling among different model generations) to assess consistency between evaluators. In other words, we evaluated 240 unique code generations, of which 120 were evaluated twice to compute human inter-annotator agreement.

5.2 Results

Table 1 shows the results of the evaluated models and reveals a clear trend: accuracy improves with model size. Notably, our largest model achieves a 20-point increase in accuracy compared to the smallest one, reaching up to 37.7% of accuracy. On the contrary, smaller models frequently generate code that violates the API's specifications, leading to a higher number of runtime errors (40% of the cases for the smallest Qwen2.5). In contrast to accuracy, the rate of semantic errors, where the code executes without crashing but fails to produce the correct outcome, does not decrease with model size. This is primarily because larger models generate a larger amount of executable code, thereby increasing the potential for such errors to surface.

A special mention is deserved to the 32B version, which is capable of reducing both errors at the same time, meaning that the improvement is directly reflected in the accuracy.

Table 2 presents the distribution of four types of errors made by the evaluated models (see Section 5.1 for the categories of the error types). We observe a clear trend across model sizes. The frequency of the error *No Change* decreases as the model size increases, from 53.3% in the 1.5B model to just 21.0% in the 72B model. These figures correlate with the results shown in Table 1, in which larger models are less likely to commit runtime errors, and therefore more likely to create code that makes changes.

Similarly, *Collateral* and *Other* errors, reflecting misdirected edits, decrease with model size, suggesting that larger models develop a more accurate understanding of the target entities to be edited.

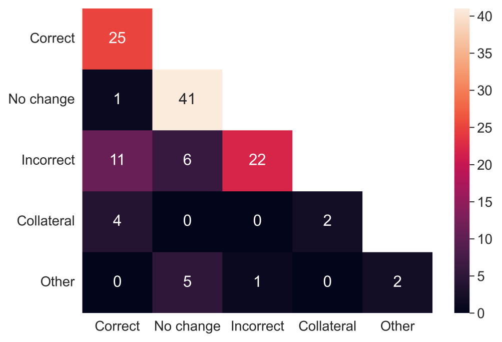


Figure 2: Heatmap of inter-annotator agreement. We aggregate the responses of all annotation pairs to check cases in which annotators tend to disagree.

Model	Accuracy ↑	
	House	School
Qwen2.5-1.5B	17.7	17.7
Qwen2.5-7B	31.0	15.3
Qwen2.5-32B	38.0	**33.0**
Qwen2.5-72B	**42.0**	**33.0**

Table 3: Accuracy of the models by BIM file.

Notably, the 72B model exhibits the lowest number of both *Collateral* and *Other* errors, further highlighting the benefits of increased model capacity for precise code editing. However, the rate of *Incorrect* edits increases with model size, reaching 35.7% in the 72B model. This suggests that larger models are better at identifying the correct target objects, but as a result of this, they are also more prone to making inappropriate modifications once the target is located.

Regarding feedback generation, evaluators found that all models produced feedback reasonably aligned with the applied changes, with accuracy scores ranging between 50 and 60. No clear correlation was observed between feedback quality and model size. This outcome is expected, as the models lack access to the 3D environment's state and must rely solely on the success or failure of executing the generated code, without grounding their feedback in the actual visual or spatial context.

5.3 Human Inter-Annotator Agreement

Figure 2 presents the distribution of aggregated label assignments of all pairs of evaluators to code execution outcomes. The strong concentration of instances along the diagonal indicates a high level of agreement between annotators. Overall, the annotations yield a Cohen's Kappa of 0.67, which corresponds to a substantial agreement according to the interpretation scale proposed by Landis and Koch (1977).

As shown in Figure 2, most disagreements arise in cases where the system interacts exclusively with entities mentioned in the query, leading to diverging labels between *correct* and *incorrect*. This is expected, given the inherent ambiguity in query interpretation for this task. To a lesser extent, disagreements also occur when an unintended object is inadvertently modified by the model (*correct* vs. *collateral*), or when evaluators overlook subtle or unexpected changes, resulting in confusion between other and no change labels (*other* vs. *no change*).

An unexpected source of disagreement was observed in six cases, where one evaluator labeled the outcome as *no change*, while the other identified at least one modification to the target object and labeled it as *incorrect*. This discrepancy suggests that while annotators generally agree on clear-cut cases, disagreements often stem from ambiguity in query interpretation and subtle changes in the scene. We hypothesize that such disagreements primarily arise from differences in how annotators interpret the intent and specificity of the query, particularly in cases where the action involves multiple entities or results in minimal visible impact.

5.4 Analysis

We conducted an analysis to identify the main limitations and strengths of our system. More specifically, we focused on measuring the impact of BIM complexity in correlation with model sizes.

The effects of BIM complexity. To analyze the effectiveness of the models across BIM scenarios of varying complexity, we compare performance between the simpler House environment and the more complex School file. As shown in Table 3, model accuracy is significantly lower in the complex scenario, and the improvements gained from larger models are less consistent. Evaluation is also more challenging in complex settings, with inter-annotator agreement (Cohen's Kappa) decreasing from 0.713 for House to 0.605 for School.

Code quality correlates with model size. Table 4 demonstrates the relationship between model size and code quality through two code snippets generated by 7B and 72B models in response to the query "Show the hidden beam and remove the others."

```python
# Get the list of hidden objects
hidden_objects = l.get_hidden_objects()

# Check if there are any hidden beams
beams_to_show = [obj for obj in  ... ]

# Show the hidden beam
if beams_to_show:
    beam_id = beams_to_show[0]
    l.set_object_visibility(beam_id, True)

# Remove other hidden beams
for beam_id in beams_to_show[1:]:
    l.destroy_object(beam_id)
```

```python
# Get the list of all beams
beam_ids = [b.guid for b in ifc.find_all_beams()]

# Get the list of hidden objects
hidden_objects = l.get_hidden_objects()

# Show the hidden beam and remove the others
for beam_id in beam_ids:
    if beam_id in hidden_objects:
        l.set_object_visibility(beam_id, True)
    else:
        l.destroy_object(beam_id)
```

Table 4: Code generations for the 7B and 72B models given the following query: "Show the hidden beam and remove the others."

The task assumes that all beams in the scene are visible except one. The 7B model fails to reason correctly, incorrectly identifying multiple hidden beams and subsequently removing them. In contrast, the 72B model exhibits more robust reasoning by first retrieving the list of all hidden beams and only removing those that were previously visible, thereby achieving the correct outcome upon execution. Notably, the 1.5B model produces code that fails to execute due to runtime errors, further underscoring the importance of model size in computationally demanding environments.

5.5 Challenges

LLMs continue to face significant challenges in this domain. Common issues include failing to identify all objects requiring modification or incompletely implementing requested changes.

Another major challenge stems from BIM file quality. Even when models generate syntactically correct code, incorrectly labeled objects prevent successful command execution. This issue is exacerbated by building complexity, which exhibits a negative correlation with system performance.

6 Conclusion

In this paper, we have presented BIM-LLM, an interactive voice-controlled assistant for architectural design review in VR environments. Our system converts voice commands into executable Python code, enabling users to rapidly prototype changes to BIM files without requiring domain expertise.

Manual evaluation demonstrates that model size is crucial for generating robust code with superior reasoning capabilities. Inter-annotator agreement reached substantial levels, and we identified the most common sources of disagreement, which are concentrated in different interpretations of the query.

Future work will explore vision-language models that incorporate visual information from the rendered environment to determine whether current models can leverage visual and spatial context to respond more accurately to instructions. Additionally, we plan to investigate automated evaluation methods to reduce the manual annotation burden. Similarly, we plan to investigate another line of research that would find ways to adapt code generation to environments with low annotated data.

Overall, we believe that the system demonstrates the capabilities and limitations of the current models. In the future, it could serve as a benchmark to evaluate different capabilities required for any agent operating in realistic environments.

Limitations

To enable new commands, both the custom Unreal Engine sandbox and Python API require adaptation, and the corresponding Python functions must be predefined. This process implies some manual labor when users wish to incorporate new functionalities.

The code-generation LLM may also hallucinate plausible but non-existent Python functions that are not executable within the API. While this issue can be mitigated by filtering incorrect function names, it remains a current limitation of the system.

Acknowledgments

This work was developed under the grant 101135724 (LUMINOUS) of the EU Horizon Europe Framework.

References

Maia Aguirre, Ariane Méndez, Aitor García-Pablos, Montse Cuadros, Arantza del Pozo, Oier Lopez de Lacalle, Ander Salaberria, Jeremy Barnes, Pablo Martínez, and Muhammad Zeshan Afzal. 2025. Conversational tutoring in VR training: The role of game context and state variables. In *Proceedings of the 15th International Workshop on Spoken Dialogue Systems Technology*, pages 215–224, Bilbao, Spain. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Stephanie Everett, Kenneth Wauchope, and Manuel A. Pérez-Quiñones. 1999. Creating natural language interfaces to vr systems. *Virtual Reality*, 4:103–113.

Stephanie S. Everett, Kenneth Wauchope, and Manuel A. Pfirez. 1997. A spoken language interface to a virtual reality system (video). In *Fifth Conference on Applied Natural Language Processing: Descriptions of System Demonstrations and Videos*, pages 36–37, Washington, DC, USA. Association for Computational Linguistics.

Silvia García-Méndez, Francisco de Arriba-Pérez, and María del Carmen Somoza-López. 2024. A review on the use of large language models as virtual tutors. *Science & Education*, pages 1–16.

Daniele Giunchi, Nels Numan, Elia Gatti, and Anthony Steed. 2024. Dreamcodevr: Towards democratizing behavior design in virtual reality with speech-driven programming. *2024 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*, pages 579–589.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962.

Suhyung Jang and Ghang Lee. 2024. Interactive design by integrating a large pre-trained language model and building information modeling. In *Computing in Civil Engineering 2023*, Computing in Civil Engineering 2023: Visualization, Information Modeling, and Simulation - Selected Papers from the ASCE International Conference on Computing in Civil Engineering 2023, pages 291–299, United States. American Society of Civil Engineers (ASCE). Publisher Copyright: © 2024 Computing in Civil Engineering 2023: Visualization, Information Modeling, and Simulation - Selected Papers from the ASCE International Conference on Computing in Civil Engineering 2023. All rights reserved.; ASCE International Conference on Computing in Civil Engineering 2023: Visualization, Information Modeling, and Simulation, i3CE 2023 ; Conference date: 25-06-2023 Through 28-06-2023.

Mikhail Konenkov, Artem Lykov, Daria Trinitatova, and Dzmitry Tsetserukou. 2024. VR-GPT: Visual Language Model for Intelligent Virtual Reality Applications. *Preprint*, arXiv:2405.11537.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Ang Li, Peter Kok-Yiu Wong, Xingyu Tao, Jun Ma, and Jack C.P. Cheng. 2025. An interactive system for 3d spatial relationship query by integrating tree-based element indexing and llm-based agent. *Advanced Engineering Informatics*, 66:103375.

Alexander Prange, Margarita Chikobava, Peter Poller, Michael Barz, and Daniel Sonntag. 2017. A multimodal dialogue system for medical decision support inside virtual reality. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, pages 23–26, Saarbrücken, Germany. Association for Computational Linguistics.

Lorna Quandt, Jason Lamberton, Carly Leannah, Athena Willis, and Melissa Malzkuhn. 2022. Signing avatars in a new dimension: Challenges and opportunities in virtual reality. In *Proceedings of the 7th International Workshop on Sign Language Translation and Avatar Technology: The Junction of the Visual and the Textual: Challenges and Perspectives*, pages 85–90, Marseille, France. European Language Resources Association.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pretraining.

Rafael Sacks, Tanya Bloch, Meir Katz, and Raz Yosef. 2019. *Automating Design Review with Artificial Intelligence and BIM: State of the Art and Research Framework*, pages 353–360.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *Advances in Neural Information Processing Systems*, volume 36, pages 38154–38180. Curran Associates, Inc.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 11888–11898.

Monica Ward, Liang Xu, and Elaine Uí Dhonnchadha. 2025. A pragmatic approach to using artificial intelligence and virtual reality in digital game-based language learning. In *Proceedings of the 5th Celtic Language Technology Workshop*, pages 27–34, Abu Dhabi [Virtual Workshop]. International Committee on Computational Linguistics.

Te-Lin Wu, Satwik Kottur, Andrea Madotto, Mahmoud Azab, Pedro Rodriguez, Babak Damavandi, Nanyun Peng, and Seungwhan Moon. 2023. SIMMC-VR: A task-oriented multimodal dialog dataset with situated and immersive VR streams. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6273–6291, Toronto, Canada. Association for Computational Linguistics.

Takato Yamazaki, Tomoya Mizumoto, Katsumasa Yoshikawa, Masaya Ohagi, Toshiki Kawamoto, and Toshinori Sato. 2023. An open-domain avatar chatbot by exploiting a large language model. In *Proceedings of the 24th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 428–432, Prague, Czechia. Association for Computational Linguistics.

Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. 2023. Mm-react: Prompting chatgpt for multimodal reasoning and action. *Preprint*, arXiv:2303.11381.

Junwen Zheng and Martin Fischer. 2023. Dynamic prompt-based virtual assistant framework for bim information search. *Automation in Construction*, 155:105067.

A Prompts

A.1 Code generator prompt

Your task is to fulfill a query given by the user in a 3D environment. The query will involve getting some information about a specific entity (or a group of them) or changing some of its/their features.

To get the query done, you will need to write Python code. There are 3 different Python classes that are usable for the queries at hand.
* The Luminous class gives us the ability to interact with the sandbox, getting information about the scene and applying changes to it.
- Function names with the 'object' substring affect just BIM entities (e.g. walls, doors), whereas the ones with 'prop' affect only props (e.g chairs)

* The IFC class is useful to determine the type of an object/entity of the scene, that is, defining whether the object is a wall, a window...
* The Entity class defines each object's name, type and id.

Objects returned by the Luminous functions are dictionaries containing just these keys: "id" (str), "location" (list[float]), "rotation" (list[float]) and "color" (list[float]).

Apart from that, you can add props and transform them. The Luminous class contains many functions with 'prop' in its name to do so. The list of the available props are the following:
* Barn Lamp: "data/props/AnisotropyBarnLamp.glb"
* Boom box: "data/props/BoomBox.glb"
* Purple chair: "data/props/ChairDamaskPurplegold.glb"
* Plant: "data/props/DiffuseTransmissionPlant.glb"
* Velvet sofa: "data/props/GlamVelvetSofa.glb"
* Iridescence lamp: "data/props/IridescenceLamp.glb"
* Punctual lamp: "data/props/LightsPunctualLamp.glb"
* Sheen chair: "data/props/SheenChair.glb"
* Leather sofa: "data/props/SheenWoodLeatherSofa.glb"
* Pouf: "data/props/SpecularSilkPouf.glb"
* Sunglasses: "data/props/SunglassesKhronos.glb"
* Toy car: "data/props/ToyCar.glb"
* Water bottle: "data/props/WaterBottle.glb"

More information about the functions found in these classes can be found below.

{api_documentation}

The following buildings can be loaded. but load them only when prompted to do so:

* House: "data/ifc/AC20-FZK-Haus.ifc"

* School: "data/ifc/Technical_school-current_m.ifc"
* Office: "data/ifc/Office Building.ifc"

In the code you generate, you will consider that the following variables are already instantiated:

```python
l = Luminous()
ifc = IFC(l.load_ifc("data/ifc/AC20-FZK-Haus.ifc"))
```

Moreover, you must not give any explanation outside the code.

A.2 Feedback generator prompt

A user made an initial query. If its outcome is positive, state the changes made in the scene. Otherwise, mention that there was an error and the query could not be fulfilled. Answer using only ONE sentence.

User query: {query}
Outcome: {outcome}