A NATURAL LANGUAGE INTERFACE FOR MODELING IN RHINO 3D USING LARGE LANGUAGE MODELS

by

Sharaa Norouzi Talkhounche

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Architecture and
Master of Science in Information Technology

Charlotte

2024

Approved by:

____

Jefferson Ellinger

____

Dr. Razvan Bunescu

____

Rachel Dickey

ABSTRACT

SHARAA NOROUZI TALKHOUNCHE. A Natural Language Interface for Modeling in Rhino 3D Using Large Language Models. (Under the direction of JEFFERSON ELLINGER)

In an increasingly interconnected world, effective communication with digital design software is paramount. This thesis addresses the challenge of enabling engineers, designers, and creators to communicate with Rhino 3D software through an interactive, user-friendly natural language interface.

Many users, including new students and experienced individuals unfamiliar with Rhino, often struggle with using the correct instructions to fully utilize the software's capabilities. This challenge impedes them from leveraging the software's full potential. To address this issue, this thesis presents an integrated system that combines Large Language Models (LLMs) with the Rhino API, facilitating intuitive communication with the software.

The objectives of this thesis included developing this integrated system to make Rhino accessible to users with limited prior knowledge and empowering experienced users to explore advanced capabilities. The methodology employed large language models such as GPT-4, coupled with the Rhino API, and focused on prompt engineering techniques to effectively instruct and interact with LLMs. This process involved crafting precise and context-aware prompts, breaking down tasks into step-by-step instructions, and iteratively refining prompts through interactive dialogues.

The findings of this research highlight the feasibility of the proposed interface in enhancing user interaction with digital design software, promoting accessibility and facilitating a better learning environment in the field of modeling and design. While the system has shown promise in bridging the gap between natural language and complex software commands, it primarily serves as a proof of concept at this stage.

In conclusion, this thesis establishes the foundational work for future enhancements and applications, indicating potential paths for transforming human-computer interaction in parametric design. The focus remains on proving the concept and assessing feasibility, with subsequent research needed to realize its full transformative potential.

DEDICATION

To my family, whose boundless love and unwavering support have been my guiding star. You believed in me, and for that, I am forever grateful.

And to the courageous girls across the globe, especially those from my homeland, Iran. Your strength and resilience inspire us all.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Jefferson Ellinger, for his invaluable guidance and support throughout my research and the completion of my degree. His commitment to fostering my interests and encouraging my unique approach to learning has profoundly shaped my educational experience. I also wish to thank my committee members, Dr. Razvan Bunescu and Rachel Dickey, for their insightful feedback and encouragement, which have greatly enriched my work.

Furthermore, I acknowledge the financial support from a GASP Award provided by the School of Architecture, which was essential for the completion of this program.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER 1: INTRODUCTION 1

1.1. Background 1

1.1.1. Rhino 3D 1

1.1.2. Large Language Models (LLMs) 1

1.2. Literature Review 3

CHAPTER 2: METHODOLOGY 6

2.1. Overview 6

2.2. System Architecture 7

2.2.1. RhinoCommon Framework Integration 7

2.2.2. Python Script and OpenAI API Interaction 7

2.3. Generative Script Using Large Langugae Models 8

2.3.1. Prompt Engineering 8

2.4. LangChain Framework 13

2.4.1. Retrieval Augmented Generation 13

2.4.2. Indexing 14

2.4.3. Retrieval and Generation Chain 14

2.5. Environment Setup 17

2.5.1. Visual Studio and RhinoCommon Framework 17

2.5.2. Anaconda and Python Environment 17

2.5.3. Integration of Components 18

2.5.4. Tooling and Configuration 18

CHAPTER 3: EVALUATION AND RESULTS 19

3.1. Test Cases 19

3.2. Performance Metrics 21

3.3. Case Studies 24

3.3.1. Case Study 1: Basic Geometry Creation 24

3.3.2. Case Study 2 24

3.3.3. Case Study 3 25

CHAPTER 4: DISCUSSIONS AND FUTURE WORK 28

4.1. Challenges 28

4.2. Future Work 29

CHAPTER 5: CONCLUSIONS 30

REFERENCES 31

APPENDIX A: PROMPTS 33

APPENDIX B: SOURCE CODE 38

APPENDIX C: TEST CASES 46

APPENDIX D: GITHUB 49

LIST OF TABLES

TABLE 3.1: Rating Criterea 20

TABLE 3.2: Performance Metrics 22

TABLE 3.3: Case Study 1 24

TABLE 3.4: Case Study 2 24

TABLE 3.5: Case Study 3 25

TABLE C.1: Development Test Cases 46

TABLE C.2: Evaluation Test Cases 48

LIST OF FIGURES

FIGURE 1.1: Rhino 3D Interface 2

FIGURE 2.1: Architecture 7

FIGURE 2.2: Simple Chain 10

FIGURE 2.3: Retrieval Chain 14

FIGURE 3.1: Mean and Standard Deviation 22

FIGURE 3.2: Average Ratings by Category and Model 22

FIGURE 3.3: Count Distribution by Category 23

LIST OF ABBREVIATIONS

CAD An acronym for Computer Aided Design.

CLI An acronym for Command Line Interface.

GPT An acronym for Generative Pre-trained Transformer.

LLM An acronym for Large Language Models.

CHAPTER 1: INTRODUCTION

1.1 Background

1.1.1 Rhino 3D

Rhino 3D is a powerful software application used primarily for parametric modeling, computer-aided design (CAD), and 3D graphics. Developed by Robert McNeel & Associates, Rhino 3D is highly regarded in various industries, including architecture, engineering, product design, and more [1].

One of the notable features of Rhino 3D is its application programming interface (API). The Rhino API allows developers and users to extend the software's functionality by creating custom scripts, plugins, and applications. This API opens up a world of possibilities for automating tasks, creating parametric models, and integrating Rhino 3D with other software tools [2].

The Rhino command line is another crucial aspect of the software. It provides a text-based interface that allows users to interact with Rhino by entering commands and parameters. While experienced users often find the command line efficient for executing precise actions, newcomers and those with limited technical knowledge may struggle with mastering the specific commands required for various tasks [3].

1.1.2 Large Language Models (LLMs)

Large Language Models (LLMs) represent a transformative leap in Natural Language Processing (NLP). These models, characterized by their vast parameter counts, have redefined NLP capabilities. The advent of the Transformer architecture, introduced in "Attention is All You Need" by Vaswani et al. (2017), played a pivotal role in their evolution. LLMs are trained on extensive textual data, enabling them to

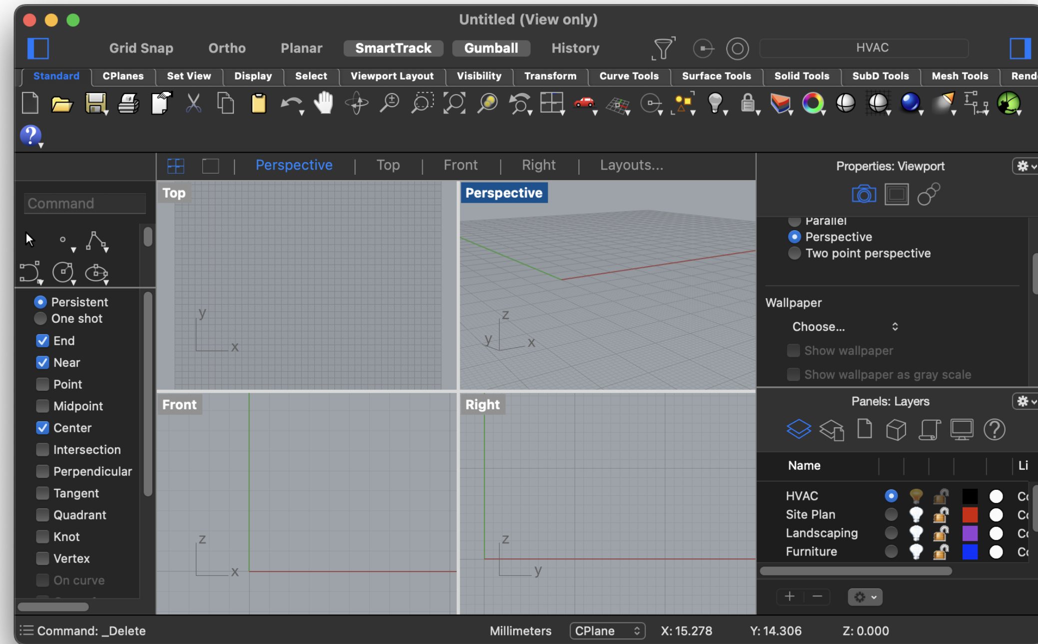


Figure 1.1: Rhino 3D Interface

understand context, semantics, and linguistic nuances. This understanding extends their utility across a spectrum of language tasks, from translation to text generation. Notable examples include BERT, as described in "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" by Devlin et al. (2018), and GPT-3 (Brown et al., 2020), which demonstrated human-level proficiency in various language tasks. The history of LLMs is marked by their ever-increasing scale and performance, underlining their significance in reshaping NLP and advancing AI.

1.1.2.1 GPT-3

GPT-3, short for "Generative Pre-trained Transformer 3," is a cutting-edge language model developed by OpenAI. It's designed to understand and generate human-like text, making it seem almost like it's written by a person. GPT-3 is known for its incredible versatility, with the ability to perform various language tasks, from answering questions to creating content. Its large scale, with 175 billion parameters, gives it an edge in processing and generating text that's impressively coherent and context-aware. GPT-3 has found applications in chatbots, virtual assistants, and content generation, making it a significant advancement in the world of artificial intelligence and natural language processing. [4]

1.1.2.2 GPT-4

Building upon the success of its predecessor, GPT-3, GPT-4 is the next stride in the evolution of AI language models by OpenAI. This advanced multimodal model seamlessly handles both text and images, producing outputs that rival human-level performance. Notably, GPT-4 excels in various professional and academic benchmarks, including a simulated bar exam where it ranks among the top 10

GPT-4, like its siblings, relies on the Transformer architecture and is primarily trained to predict the next token in a text. This foundational understanding of language and context enables it to generate coherent and context-aware responses.

An important aspect of the GPT-4 project is the development of infrastructure and optimization methods that maintain predictability across a wide range of scales. This predictability allows OpenAI to extend the model's capabilities even to models trained with significantly fewer computational resources. GPT-4, like its predecessors, continues to contribute to the advancement of AI and NLP, showcasing the ongoing progress in this field. [4]

1.2 Literature Review

The integration of natural language processing (NLP) with the generation of code represents a compelling research route, serving as a pipeline between the expressiveness of human language and the precision of machine code [5]. Research in this field has unveiled the capacity of extensive codebases to support the automation of code generation and its interpretation via machine learning [6]. The role of semantic parsing is pivotal, demonstrating the feasibility of extracting structured queries from conversational language while underscoring the need for semantic precision [7]. Advances in the field have brought to light methods that derive executable programs from verbal descriptions, highlighting the production of code that is both semantically nuanced and context-sensitive [8]. The advent of syntactic neural models tailored for generating code has significantly contributed to this area, integrating syntactic constraints to improve the generated code's accuracy [5]. Furthermore, efforts to transform natural language inquiries into structured database queries have proven the practicality of NLP in translating informal dialogue into formal language constructs [9]. This body of research collectively illustrates the transformative capability of NLP, facilitating an intuitive and streamlined approach to converting linguistic inputs from humans into formalized code and language structures.

Generative AI is significantly altering the landscape of design software, introducing a new era where algorithmic processes augment human creativity and efficiency. In architectural design, parametric modeling has become a foundation, emphasizing its potential to automate and optimize design processes [10]. Similarly, in the domain of product development, the integration of performance-driven generative design tools has been shown to revolutionize product conception, enabling the creation of more complex, functional, and comperhensive solutions [11]. The concept of computational creativity extends the generative design discourse, proposing frameworks where AI systems exhibit autonomous creative behaviors, thus fostering innovative solutions that go beyond traditional design constraints [12].

The integration of generative AI and Large Language Models (LLMs) in educational platforms facilitates innovative approaches to learning new tools and enhancing user interfaces.. Meanwhile, LLMs, exemplified by OpenAI's GPT models, have revolutionized the way interactive learning systems respond to user queries, providing instant, context-sensitive support that can mimic one-on-one tutoring sessions [13]. These advancements not only make learning more interactive and personalized but also significantly enhance the efficacy of digital educational tools.

CHAPTER 2: METHODOLOGY

2.1 Overview

The application serves as a bridge between Rhino 3D's command-line interface and the user's intentions, facilitating a more intuitive interaction within the 3D design environment. Its main functionality lies in its ability to translate natural language instructions into Rhino 3D's command-line scripts. This process leverages the capabilities of GPT API to interpret the user's natural language input, convert it into a formal command script, and execute the resulting actions within Rhino 3D. The integration of RhinoCommon API plays a critical role here, enabling the application to interact directly with Rhino 3D and enhance user experience by leveraging digital potential.

The practicality of this application is evident through its integration of advanced language processing techniques and the established Rhino3D software framework. The application streamlines command-line operations by converting natural language instructions into Rhino3D's formal command language. This process leverages the significant advancements in Large Language Models (LLMs), crucial for interpreting nuanced human language, and RhinoCommon API, a cornerstone in the Rhino3D community for developing plug-ins and interfaces that enhance functionality and user interaction. This approach not only improves user accessibility and efficiency but also broadens the potential for digital design, making sophisticated 3D modeling tasks more accessible and enhancing usability across various levels of expertise.

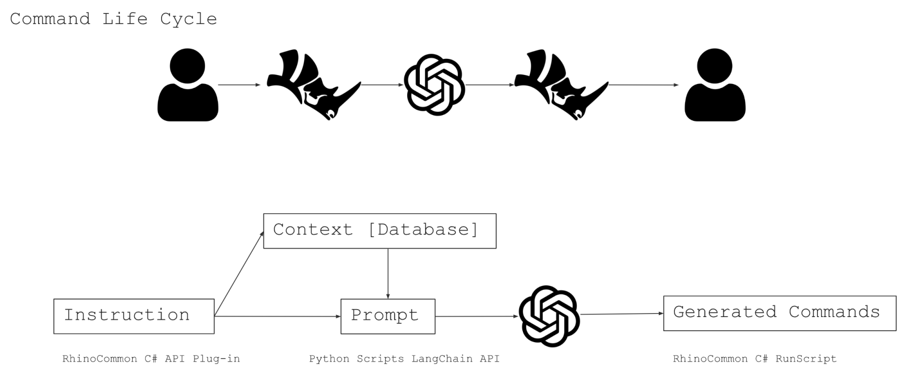


Figure 2.1: Architecture

2.2 System Architecture

The application's architecture is divided into two primary components: the RhinoCommon framework integration and a Python script interfacing with the OpenAI API. Here's how each part functions within the system:

2.2.1 RhinoCommon Framework Integration

The application utilizes the RhinoCommon framework, employing C and .NET to interact with Rhino 3D software. At its core, it will override the RunCommand method, which asks for user instructions and then validates it. Then, it will set up the anaconda environment with the proper packages and installed resources. This setup initiates the main process by using natural language instructions from the user. These instructions are then passed as inputs to a Python script tasked with communicating with the OpenAI API. The standard output from this process will be read and used by RunScript method in order to trigger the Rhino 3D visual output.

2.2.2 Python Script and OpenAI API Interaction

The Python component employs the LangChain framework to orchestrate a sequence of operations that transform user input into a structured prompt. This transformation integrates the user's natural language instructions with a template prompt, optimizing the query for the GPT API.

2.3 Generative Script Using Large Langugae Models

Prompt engineering is a critical component in the interface between natural language processing models and application-specific commands, particularly when integrating AI capabilities like those provided by OpenAI's GPT models into user-focused software tools. Effective prompt engineering involves crafting inputs that leverage the model's pre-trained knowledge to produce outputs that are both accurate and relevant to the task at hand[14]. This process is particularly important in applications like 3D modeling software like our case, where commands must not only be syntactically correct but also contextually appropriate to execute the desired actions within the software environment. Recent studies highlight the importance of fine-tuning the interaction between large language models and end-user applications, emphasizing that even subtle variations in prompt structure can significantly impact the quality of the output[13].

2.3.1 Prompt Engineering

The initial phase involves an in-depth exploration of prompt variations to ensure the optimal way the model translates user instructions. Domain-specific examples from Rhino are used to fine-tune the translation of user intention to commands. [15]

A prompt is combined of different parts helping the model properly understand the request or instructions of the user and helps it to generate the scripts while having unambiguous instructions , Figure 2.2. Prompt Template: The template should mimic the structure of commands expected by the target application (e.g., Rhino 3D). This includes specifying the command syntax and sequence in which parameters should be presented. For example, a template for creating a sphere might structurally require '_Sphere' followed by specifications for location and radius.

To aid the AI in understanding the context of the request, templates should include placeholders for necessary contextual information that the user might provide. This could be details about the environment, previous actions, or specific requirements of the task. Including examples within the template can significantly improve the model's understanding of the desired output format. This is particularly beneficial in few-shot learning scenarios where the model uses the examples to better grasp the task requirements.

2.3.1.1 Chain of Thoughts

This technique involves encouraging the models to generate a sequence of intermediate reasoning steps, which significantly enhances their capability to tackle intricate reasoning challenges. The approach is relatively straightforward: by providing a few examples of chain of thought reasoning as exemplars during the prompting process, LLMs are able to mimic this structured thought process.

Template: The prompt template is crafted to guide the AI in a stepwise manner, ensuring that all necessary components of a command are correctly formulated:

1. Instruction Analysis: The model first breaks down the user's instruction to understand the required actions.

2. Command Identification: It then identifies the RhinoScript command(s) that correspond to the desired action.

3. Parameter Assignment: Parameters are extracted from the user's instruction, ensuring they match the requirements of the RhinoScript commands.

4. Parameter Validation: The template checks that the parameters are in the correct order and correctly formatted for each command.

5. Command Generation: Commands are generated for each step of the instruction, inserting default values for any parameters not specified by the user.

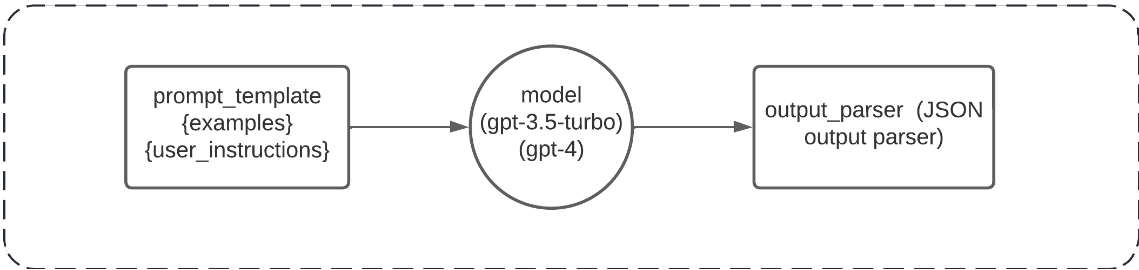


Figure 2.2: Simple Chain

6. JSON Structuring: The final output is structured in a JSON format, with an optional description field for any additional notes about the commands.

Model: It specifically refers to the API used to interact with pre-trained language models like GPT-3 or GPT-4. This API serves as the interface through which the system sends natural language queries and receives generated text.

Output Parser: Language models typically generate text outputs, but in many applications, a more structured form of information is necessary. Output parsers are designed to convert these text outputs into structured formats, enhancing the utility of language model responses. Essential to output parsers are two main methods: "Get format instructions," which provides a string of instructions on how the output should be formatted, and "Parse," which converts a text response from a language model into a structured format. Additionally, there is an optional method, "Parse with prompt," which also takes in the original prompt used to generate the response. This method allows the parser to use the context from the prompt to potentially retry or correct the output, providing a mechanism to refine parsing accuracy and relevance[16].

```
"You are an expert Rhino 3D designer who can work with Rhino 3D api,
    the user will give you instructions about an action in rhino and
```

```
you should give a command of 'RhinoScript' for running in RhinoCommon framework 'RunScript' method, put the command in a json. Take these steps to make sure you generate the proper script:
1. Find the steps needed for the instruction to be executed.
2. Find the proper command for each step
3. Find the proper measurements based on the users instruction and assign it to the parameters.
4. Make sure you are using the correct parameters and correct order of parameters for each command.
5. Generate the command for each step and fill in the unknown values for parameters with default values.
6. structure the commands in a json format.
7. If you have any notes or description about the commands, put it in the 'description' note in the JSON object.
"Examples": [
    {{"user_instruction": "please create a sphere",
      "output": {{"RhinoScript": ["_-Sphere 0,0,0 1"]}}"
    }},
    {{"user_instruction": "Create an ellipse with a major axis of 15 units and a minor axis of 10 units.",
      "output": {{"RhinoScript": ["_-Ellipse 0,0,0 0,15,0 10,0,0"]}}
    }}
    {{"user_instruction": "Lock all objects on the 'Furniture' layer.",,
      "output": {{"RhinoScript": ["_-Layer _Lock \"Furniture\" _Enter]}}
    }}
  ]
This is user's request: << {user_instruction} >>

ouptut format(JSON):
{{ "RhinoScript": ["_-Sphere 0,0,0 1"],
```

```
"description": "It is sphere centering at point 0,0,0 with radius of 1 unit. The steps needed are 1. Using the Cone command, 2. Choosing the center point, 3. Entering the radius. The command used is Sphere and the parameters are the center point and the radius like _-Sphere center radius"
}}
}
```

In this project, few shot learning has been used to show case an example of the desires structure needed by the source code of rhino plug-in in order to be able to use it as an script. Another important factor is to help the model understand the beginning and the end of the user instruction and distinguish it properly from the static part of the prompt. Finally, for tuning the generated results, sometime it is needed to have more examples, specifically to help the model with accuracy of the scripts. In this case, the first few iterations of the generated scripts were unreadable by the Rhino 3D software, because of the inaccuracy in parameters order for most of the commands in Rhino 3D CLI. This happens because the LLM models are usually trained on internet corpus but there is not enough resources for CLI commands in written format.

2.3.1.2 Few Shot Learning

Few-shot learning explores the capability of language models, particularly GPT, to learn from a minimal number of examples, typically fewer than ten. This approach is part of broader efforts to enhance the flexibility and adaptability of machine learning models without the need for extensive training data. Few-shot learning leverages the extensive pre-training of models like GPT on diverse data, allowing them to apply accumulated knowledge to new tasks with only a few illustrative examples. In practice, this means that GPT can perform tasks like translation, question answering, and even specific professional tasks such as medical diagnosis, by seeing just a few examples of the task at hand. This method significantly reduces the time and data requirements typically associated with training models for specialized tasks, marking a significant advancement in the efficiency and practicality of deploying AI in various applications[13].

The use of few-shot learning within this framework is critical, as it tends to yield more accurate and contextually relevant responses compared to zero-shot learning. Few-shot learning is a technique in prompt engineering helping the LLM models to understand the request through examples to enhance the performance and accuracy. Also, when having a structured output, few-shot learning can help to generate the proper structure that can further be parsed into the desired components by the application. The number and variety of these shots play an important role for a more generalized output. It is important that these shots be variant enough to help the model understand different combinations of instructions but also proper to prevent any over-fitting. However, it's noted that the LLM model can still produce hallucinations, generating commands that are not applicable or contain errors, such as non-existent methods or incorrect parameters in the Rhino environment.

Using OpenAI's API, the GPT-4 model was queried with natural language prompts from users. These prompts were then transformed into structured RhinoCommon API commands. These commands are the same commands the user employ while using the Rhino 3D software command line. For example, for drawing a line, a user can either apply the command using the visual interface or the command line interface with the following command. _Line 0,0,0 1,1,1

2.4 LangChain Framework

2.4.1 Retrieval Augmented Generation

Retrieval Augmented Generation is a technique used mainly to help LLM with supplementary data. LLMs can be powerful tools for reasoning, however their knowledge is limited to the data provided through their training, therefore, they can be outdated or in some cases, they do not have access to private data. This technique helps the

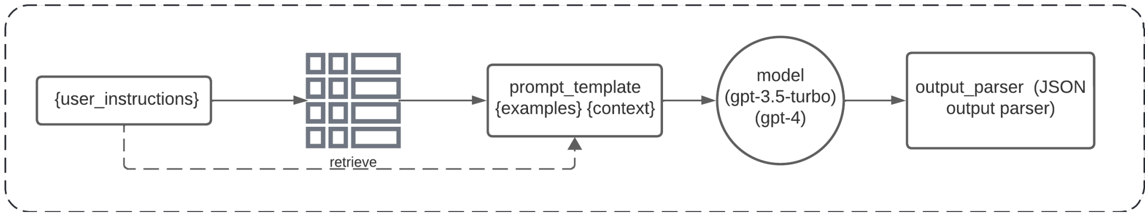


Figure 2.3: Retrieval Chain

model to access the appropriate pieces of information through the prompt to enhance the reasoning. Figure 2.3

2.4.2 Indexing

Indexing is usually an offline process in which the desired data will be processed and stored through a pipeline[16]. In this project, BeautifulSoup package is used for scraping RhinoCommon documentation webpages in a recursive function. These webpages later were loaded, splitted and finally vectorized using an OpenAI Embedding and stored in a ChromaDB database [17].

2.4.3 Retrieval and Generation Chain

By using the user input and proper embeddings, the relevant splits of the document will be retrieved from the vectorstore and used for result generation by the LLMs[16]. This step accesses the ChromaDB vectorsotre structured repository of command-specific documentation, detailed steps, and parameter data. By fetching this information, the system contextualizes the user's natural language instructions. The retrieved data is then integrated with the user's instructions to generate a comprehensive prompt. This augmented prompt is fed into the model to generate the desired command. This integration of command-specific information enriches the model's understanding, enabling it to produce better results and reducing the likelihood of generating irrelevant or incorrect commands, commonly referred to as 'hallucinations.' This method not only enhances accuracy but also leverages the model's capabilities to interpret and respond to complex design requirements effectively.

In this project, embeddings are utilized to facilitate the retrieval of documents that are most relevant to the outputs generated by the initial context question. The context question specifically asks for the optimal Rhino 3D commands and associated shapes, based on user instructions. The responses from this query are then used to search through a vectorstore a collection of document embeddings that represent a variety of potential data sources, such as HTML documents. These embeddings effectively map textual content to a high-dimensional space, where semantic similarities translate to spatial proximity. By querying this space with the embeddings of the AI-generated answers, the system can efficiently retrieve documents that are contextually related to the desired commands and shapes. These documents serve as a contextual backdrop, providing the AI with additional relevant information that enhances its ability to generate accurate and applicable Rhino 3D commands.

```
    db = Chroma(
    persist_directory=source_dir + "/chroma",
    embedding_function=OpenAIEmbeddings(model="text-embedding-ada-002"))
```

The main block orchestrates the flow of processing user instructions using an AI model. It includes setting up AI models and chains for document retrieval and question answering.

```
if __name__ == '__main__':
    user_instruction = sys.argv[1]
    model = ChatOpenAI(temperature=0.0, model="gpt-4")
```

The temperature parameter in an AI model's configuration controls the randomness of the output. A lower temperature results in more deterministic and predictable outputs, whereas a higher temperature allows for greater variability and creativity in the responses. For this project, the temperature is set to 0.0 in the model configuration to ensure that the outputs are as deterministic as possible. This setting is particularly important in the context of generating commands, where robustness and specificity are crucial. A deterministic approach ensures that the model's responses are consistent and reliably aligned with the structured data retrieved from the document embeddings. This reliability is vital for maintaining the integrity of the design process and ensuring that the commands generated by the AI are directly executable within the Rhino 3D environment without additional modifications or clarifications.

```
response_schemas = [
ResponseSchema(name="RhinoScript",
    description="The runnable script for Rhino 3D CLI"),

ResponseSchema(name="description",
    description="Description of the script and parameters")]
output_parser =
    StructuredOutputParser.from_response_schemas(response_schemas)
```

The script configures output parsers and response schemas to organize the AI's responses into structured formats, which is essential for accurately extracting the RhinoScript necessary for the RhinoCommon plugin. Large Language Models like GPT typically generate human-like text, which may not be immediately interpretable by computers. Although the output format is predetermined in the prompt template, responses - especially in GPT-4 - might still include extra explanations. Utilizing an output parser ensures that the outputs are properly structured, enabling efficient script extraction for further use.

```
context_prompt = PromptTemplate(
        template=command_question_template, ...)
retriever = db.as_retriever(search_kwargs={'k': 1})
context_chain = context_prompt | model | StrOutputParser()

docs = {
    "context": RunnableLambda(lambda x: retriever.
```

```
    get_relevant_documents(x))
}
inputs = {**user_input, **docs}
retrieval_chain =
     context_chain | inputs | command_prompt  | model | output_parser
result = retrieval_chain_.invoke({'input': user_instruction})
```

2.5 Environment Setup

The development environment for the application is structured to support the seamless integration of .NET C and Python components, facilitating the interaction between the Rhino 3D software and OpenAIś API. Here is how the environment is set up and the tools involved:

2.5.1 Visual Studio and RhinoCommon Framework

The application's Rhino 3D-related component is developed using the RhinoCommon framework within the Visual Studio IDE. Visual Studio provides a robust environment for .NET C development, offering comprehensive support for RhinoCommon SDK, which is crucial for creating and debugging the plugin for Rhino 3D. This setup allows for the development of script-running commands and the integration of user interfaces to capture natural language instructions.

2.5.2 Anaconda and Python Environment

For the Python component, the Anaconda distribution is utilized to manage the Python environment. Anaconda simplifies the process of setting up and managing various Python versions and libraries, making it an ideal choice for configuring the environment required for OpenAI API interaction. Within this environment, the LangChain library and framework are installed to facilitate the construction of structured prompts and handle the communication with the OpenAI API.

2.5.3 Integration of Components

The integration between the C# and Python components is crucial for the application's functionality. The C# code within the RhinoCommon framework triggers the Python script, which in turn processes the user's natural language input, interacts with the OpenAI API, and returns the command script to be executed in Rhino 3D.

2.5.4 Tooling and Configuration

- Visual Studio: Used for C development and RhinoCommon plugin integration.
- Anaconda: Manages the Python environment, ensuring that the necessary versions and libraries, including LangChain, are correctly configured for optimal interaction with the OpenAI API.

CHAPTER 3: EVALUATION AND RESULTS

This chapter outlines the evaluation and results of the application designed to convert natural language instructions into Rhino 3D commands using LLMs. The focus of the evaluation is on the application's effectiveness in accurately interpreting user inputs and generating corresponding Rhino 3D scripts. The testing was conducted through a series of test cases covering basic geometry creation, object editing, view adjustments, and annotations, reflecting the application's scope. These cases helped assess both the functionality and the quality of the AI-generated outputs. This section details the testing approach, the execution of the test cases, the analysis of the results, and the implications for further development.

3.1 Test Cases

A comprehensive set of test cases was developed to cover key functionalities of Rhino 3D, including basic geometry creation, object editing, view and display settings, and annotations. These cases were specifically designed to assess both the technical execution and the interpretative accuracy of the application.

Qualitative rating assessments are a crucial tool in research to evaluate complex phenomena that are not easily quantifiable. Qualitative ratings allow to assess dimensions of a subject that are subjective and nuanced, often based on criteria that do not lend themselves to straightforward numeric scoring. This method involves developing a set of criteria that reflect the key aspects of the phenomenon being studied and then assigning ratings based on the judgment of experienced evaluators. The qualitative ratings can provide a deeper insight into the effectiveness, usability, and overall impact of an intervention or system, offering a richer and more holistic view

Table 3.1: Rating Criterea

Rating 1	Correct Command Choice
Rating 2	Accuracy of Parameters
Rating 3	Parameter Order
Rating 4	Execution Success
Rating 5	Overall Effectiveness

than quantitative metrics alone[18].

This rating system, ranging from 1 to 5, critically evaluates several key aspects of each generated script. Each generated script performance is rated during the testing phase based on the following rating system.

1. Rating 1: The script fails to recognize the proper command.

2. Rating 2: The script recognizes the proper command for the instruction but not the proper parameters.

3. Rating 3: The generated script is using the proper command and parameters but in an incorrect order or format and execution fails.

4. Rating 4: The script is largely effective with correct commands and parameters, though minor improvements are possible.

5. Rating 5: The script perfectly executes the intended command with optimal parameters and excellent adherence to user instructions, fully satisfying the design requirement.

In the testing framework, two distinct types of test cases were employed: development test cases and evaluation test cases. Development test cases were a smaller, focused set of scenarios used primarily during the initial phases of the application's creation. These cases, numbering 11 , Table C.1, were instrumental in fine-tuning the application's functionality, specifically in optimizing the prompt engineering process to enhance the AI's ability to interpret and execute natural language instructions accurately. They served as a foundational tool for iterative refinement, allowing for adjustments in the application's design and its interaction with the GPT API based on observed performance and output quality. Evaluation test cases, on the other hand, were used to assess the application performance across a wider spectrum of Rhino 3D functionalities. These test cases, numbering 48, Table C.2, were designed to cover a variety of user instructions, testing the application ability to handle different commands and its accuracy in translating them into Rhino 3D actions. Through these evaluation test cases, the effectiveness and adaptability were examined, offering insights into its practical application and highlighting areas for potential improvement. The testing process was iterative, allowing for the refinement of both the application and the testing suite. Initial rounds of testing informed prompt adjustments and application enhancements to improve performance and accuracy.

Where possible, automation was employed to streamline the testing process, especially for straightforward or repetitive test cases. Manual oversight remained crucial, however, to interpret results, especially in cases involving a more complex or subjective assessments of output quality.

3.2 Performance Metrics

For each model iteration, the average execution rating is calculated along with the standard deviation to provide insight into the variability of the model's performance. These statistics help identify how consistently each model performs across different test scenarios.

Mean Scores: The average execution rating for each model across all test cases, providing a snapshot of overall performance.

Standard Deviation: Measures the amount of variation or dispersion from the average, offering insights into the consistency of each model's performance.

In the evaluation data, GPT-4 without contextual enhancements demonstrated a higher incidence of failures, notably generating incorrect commands or parameters

Table 3.2: Performance Metrics

	gpt-3.5-turbo	gpt-4	RAG gpt-4
Rating 0	8	10	4
Rating 1	2	6	6
Rating 2	7	5	4
Rating 3	9	8	5
Rating 4	1	2	4
Rating 5	21	17	25
Mean	3.167	2.770	3.541
Standard Deviation	1.905	1.991	1.801

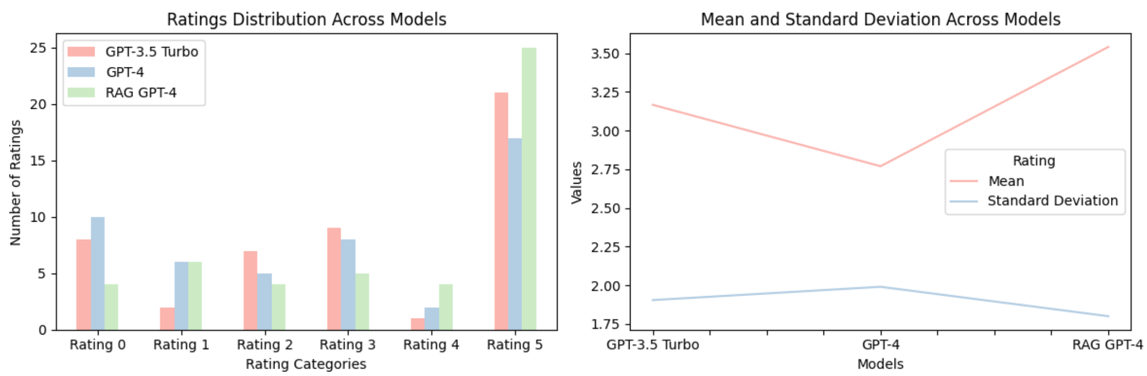


Figure 3.1: Mean and Standard Deviation

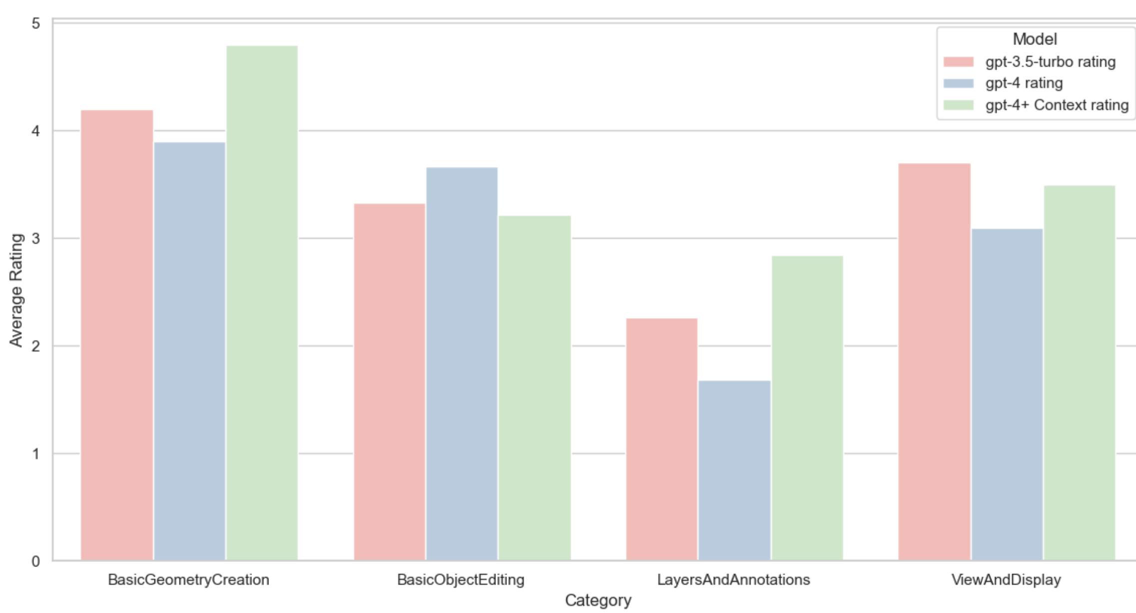


Figure 3.2: Average Ratings by Category and Model

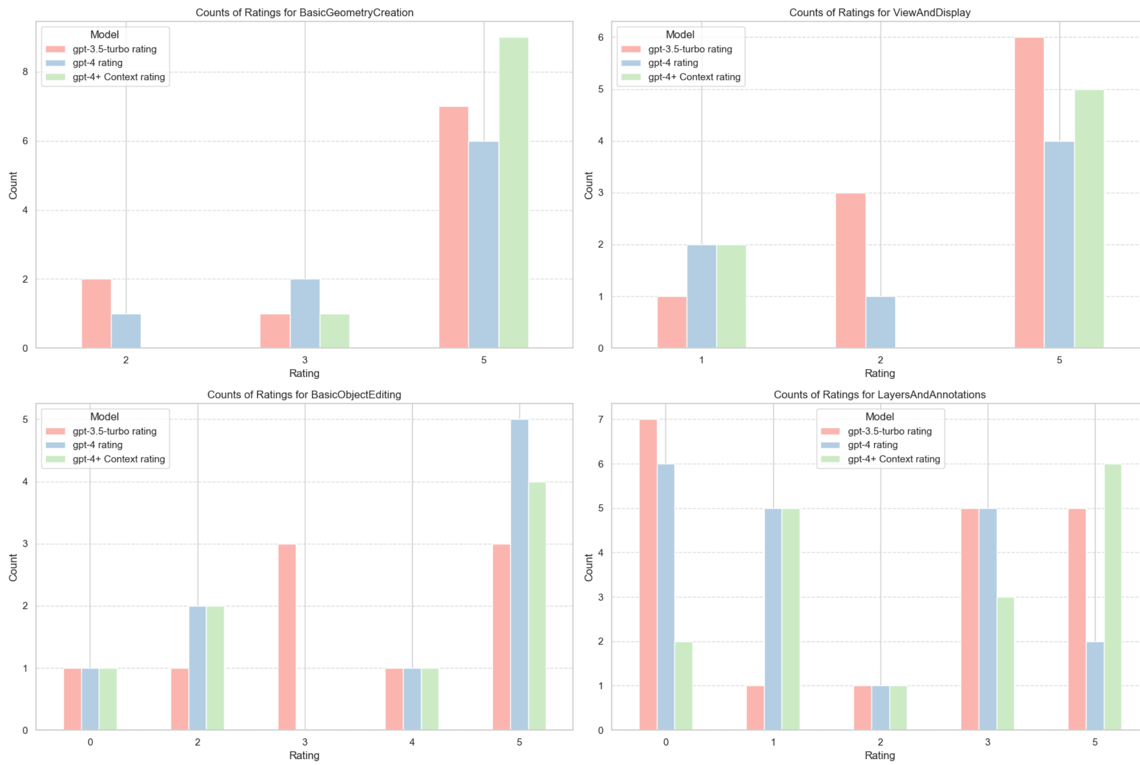


Figure 3.3: Count Distribution by Category

that do not exist within the Rhino 3D command set. This is by an increased number of cases rated as "0," indicating significant errors in output. The data suggests that by incorporating contextual enhancements into GPT-4, there was a noticeable performance improvement. The addition of context helped reduce the occurrences of hallucinated commands and parameters, aligning the model's outputs more closely with the actual functionalities of Rhino 3D. This improvement is reflected in the reduced number of failure cases and an overall higher average rating for the model with contextual data, indicating a clear benefit from integrating specific, relevant information into the model's processing framework. This suggests that the model's understanding and accuracy are enhanced by context, which helps it generate more reliable and appropriate commands for Rhino 3D. It is also possible to identify which categories experience more frequent failures. Notably, the LayerAndAnnotation instructions show a lower average rating, largely due to the dependency of annotation

Table 3.3: Case Study 1

GPT-3.5 Turbo	GPT-4	RAG GPT-4
_-Line 2,3,0 5,8,0	_-Line 2,3,0 5,8,0	_-Line 2,3,0 5,8,0

Table 3.4: Case Study 2

GPT-3.5 Turbo	GPT-4	RAG GPT-4
_-TextObject Midpoint _Pause _Enter	_-SelLine _Enter	_-SelLine _Enter

commands on measurement commands, which will be explored further.

3.3 Case Studies

3.3.1 Case Study 1: Basic Geometry Creation

Test Scenario: "Draw a straight line from point A to point B." , Table 3.3

Instructions Used: "Draw a straight line from (2,3,0) to (5,8,0)."

Results:

GPT-3.5 Turbo: Successfully generated the correct command with a high rating of 5.

GPT-4: Similar performance to GPT-3.5, also achieving a perfect score.

GPT-4 with Context: Maintained a perfect score, indicating consistent accuracy across all models for basic tasks.

3.3.2 Case Study 2

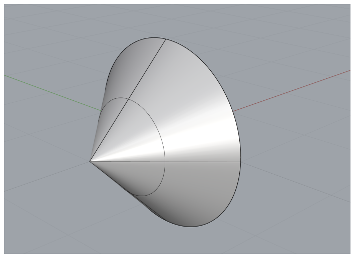
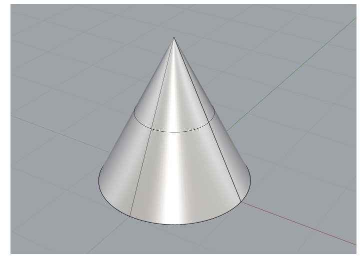
Test Scenario: "Label the midpoint of the selected line with text 'Midpoint'." Table 3.4

Instructions Used: "Label the midpoint of the selected line with text 'Midpoint'."

Results:

This case consistently underperformed across all iterations, primarily due to implementation constraints and the limitation to processing only single commands. A significant contributing factor to the failure is Rhino 3D's command line limitation, which struggles with chaining the outputs of commands. Although the GPT-4 models generated a sequence of commands that theoretically addressed the task, the imple-

Table 3.5: Case Study 3

GPT-3.5 Turbo	GPT-4	RAG GPT-4
_-Cone 0,0,0 1 0’ 	_-Cone 0,0,0 1,0,0 0,2,0 2 NONE	_-Cone 0,0,0 0,0,1 1 

mentation only utilized the first command. Further testing revealed that chaining commands, especially when an output value from one command needs to be input into the next, is not consistently effective in Rhino 3D. This limitation highlights a critical area where the interface between the GPT models and Rhino 3D needs enhancement to handle complex command sequences effectively.

3.3.3 Case Study 3

Test Scenario: Implicit Object Creation This case study examines the ability of GPT models to interpret and execute an instruction that implicitly describes a common object without explicit design parameters.

Instruction Used: "A birthday hat!" Table 3.5

Results: In this case, we also investigate the chain of thought based on the prompts and chain architecture. The description in the output as shown earlier in the schema of the output is used to explain the chain of thought of the model and the way the script is created. Furthurmore, the verbosity of the chained model can help understanding each step within the chain and investigate the context extraction and how it helps the generation of the script.

GPT-3.5 Turbo:

’description’: ’It is a cone centering at point 0,0,0 with radius 1 unit and height 0. The steps needed are 1.Using the Cone command, 2. Choosing the center point, 3. Entering the radius, 4. Entering the height. The command used is Cone and the parameters are the center point, radius, and height like _-Cone center radius height'

GPT-4:

'description': 'This script creates a birthday hat by creating two cones. The first cone is created with the base at the origin and the apex at (0,2,0) with a radius of 2. The second cone is created with the base at the origin and the apex at (0,2,0) with a radius of 2. The commands used are Cone and the parameters are the base point, the apex point and the radius like _-Cone base apex radius'

RAG GPT-4:

Chain 1:

The top 2 best RhinoScript "command line" commands that can be used to create a birthday hat in three Command keys are: ExtrudeCrv: This command can be used to extrude a curve to create the main body of the hat. FilletEdge: This command can be used to round off the edges of the hat to give it a more realistic look. The shape that better describes the request in one word is "cone" Retriever Documents:

```
page_content='Cone | Rhino 3-D modeling  Cone
...
Cone
The  Cone  command draws a cone.
Steps 1.  Pick  the center and radius of the base.
Note : See the  Circle  command for option descriptions.
2. Pick  the end of the cone (apex).
Command-line options
Direction constraint options Direction constraints restrict the
    direction of the circle. None
The center can be anywhere in 3-D space.
The second point can be placed anywhere using  elevator mode,
    object snaps  or other  modeling aids.
Vertical
Draws an object perpendicular to the construction plane.
Pick  the center and a radius or diameter.
```

```
AroundCurve
Draws a circle perpendicular to a curve.' metadata={'source': '/Cone
    ', 'title': 'Cone | Rhino 3-D modeling'}
```

'description': This script creates a cone (birthday hat) with the base centered at point 0,0,0, a base radius of 1 unit, and an apex at point 0,0,1. The steps needed are 1. Using the Cone command, 2. Choosing the center point and radius of the base, 3. Choosing the apex of the cone. The command used is Cone and the parameters are the center point, radius of the base, and apex like _-Cone center radius apex Please note that the actual size and position of the birthday hat may need to be adjusted based on the specific requirements of your 3D model."

CHAPTER 4: DISCUSSIONS AND FUTURE WORK

4.1 Challenges

Challenges in Integrating Natural Language Processing with Rhino 3D The integration of GPT-4 with Rhino 3D's command-line interface highlighted several key challenges that must be addressed to fully harness the potential of natural language processing in 3D modeling environments. The primary limitations pertain to the command-line scripting capabilities of Rhino 3D, where object selection tools lack the sophistication needed for precise manipulations without direct user input. This fundamental limitation complicates the automation of tasks that require detailed object interactions, thereby impacting the system's usability and effectiveness.

Additionally, the project encountered significant difficulties with sequencing actions and understanding the spatial orientation of objects. The CLI's limited ability to interpret and execute sequences of commands based on varying views restricts its capacity to handle complex modeling tasks. These challenges are further exacerbated by the Rhino 3D environment's inherent complexities, requiring a deep understanding of 3D modeling concepts and interactions.

Performance issues also exist, particularly related to the speed and consistency of the GPT-4 model's responses. The model often demonstrated slow response times, especially when handling multiple chained commands, leading to operational delays. Moreover, inconsistencies in the model's outputsâlikely due to ongoing modifications in the GPT-4 implementationâresulted in outputs that varied significantly between sessions, undermining the reliability and predictability of the system.

4.2 Future Work

Fine-Tuning Based on CLI Logs: Analyzing CLI logs to identify common errors and inefficiencies could provide crucial insights that help refine the GPT model's training. This fine-tuning process would enhance the model's accuracy in generating Rhino 3D commands and its capacity to handle more complex scenarios.

Integration of Voice Commands: Implementing voice command capabilities could offer a more intuitive and efficient method for user interaction with Rhino 3D. Voice commands would simplify the interface and could potentially accelerate operations, making the system more accessible and reducing the learning curve for new users.

Development of a Memory Component: Introducing a memory component that retains information from previous interactions could significantly improve the system's functionality. Such a component would allow the system to reference earlier commands and parameter values more effectively, facilitating the handling of complex design tasks that require modifications and iterations over time.

CHAPTER 5: CONCLUSIONS

This thesis explored integrating LLMs in Rhino 3D's command-line interface to translate instructions in natural language into executable CAD commands to provide a more accessible route to 3D modeling to users with varying levels of experience. The project was focused on the potential of AI to help new users by providing a more fluent interface that reduces the typical learning curve that characterizes highly specialized design software. While the model was able to handle basic commands and simple geometrical reasoning quite well, the study revealed shortcomings in applying advanced design reasoning, which is a common limitation for NLP models.

In spite of these limitations, the study showed that it was feasible to bring natural language processing into CAD tools, which should make digital design tools more intuitive and easier to learn, and potentially impact the way users interact with software. The study also showed the practical limitations of the current technologies, pointing to a need for fine-tuning both the NLP models and the CAD software interfaces to achieve these benefits. For instance, tasks that require advanced reasoning and understanding of spatial relationships pose a particular obstacle for the NLP models and should be a focus for future improvements.

With attention to these spaces, the integration of AI into professional design tools like Rhino 3D could lead to significant changes in digital design practice and allow for faster learning and more efficiency. This has the potential to open the gateway to more accessible, user-friendly computational design tools â and is part of a wider trend of technological developments in the creative professions that ease access and use. These findings demonstrate the transformative power of AI design, both as a tool for beginners and as a complex problem with many miles to go.

REFERENCES

[1] Robert McNeel & Associates, "Rhino 3d - nurbs modeling for windows," 2023.

[2] Robert McNeel & Associates, "Rhino developer docs," 2023.

[3] Robert McNeel & Associates, "Rhino command line," 2023.

[4] OpenAI, "Openai gpt-4," 2023.

[5] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," 2017.

[6] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," 2018.

[7] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on Freebase from question-answer pairs," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (D. Yarowsky, T. Baldwin, A. Korhonen, K. Livescu, and S. Bethard, eds.), (Seattle, Washington, USA), pp. 1533–1544, Association for Computational Linguistics, Oct. 2013.

[8] M. Raza, S. Gulwani, and N. Milic-Frayling, "Compositional program synthesis from natural language and examples," in *International Joint Conference on Artificial Intelligence*, 2015.

[9] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," 2017.

[10] P. Janssen and R. Stouffs, "Types of parametric modelling," in *Proceedings of the 20th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)*, pp. 157–166, 2015.

[11] K. Shea, R. Aish, and M. Gourtovaia, "Towards integrated performance-driven generative design tools," *Automation in Construction*, vol. 14, no. 2, pp. 253–264, 2005.

[12] J. Mccormack, A. Dorin, and T. Innocent, "Generative design: a paradigm for design research," 2005.

[13] T. B. Brown and et al., "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[14] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[16] "Question answering langchain documentation," 2023. Accessed: 2023-04-17.

[17] J. Webster, "Chatting to a website with langchain, openai, and chromadb," 2023. Accessed: 2024-04-24.

[18] J. A. Maxwell, *Qualitative research design: An interactive approach*. Sage publications, 2012.

APPENDIX A: PROMPTS

```
You are an expert rhino3d designer who can work with rhino3d api,
the user will give you instructions about an action in rhino and you should give a list of 'RhinoScrip' for
running in RhinoCommon framework 'RunScript' method, list the commands in a json.
RhinoScripts are like this <<
'_Sphere 0,0,0 1' for creating a sphere or
'_ArrayPolar _Copy=_Yes _DeleteInput=_No _AngleBetweenCopies=90 _Center=0,0,0 _Count=4 _DeleteInput=No' for array the object
>>
This is users request: << {user_instruction} >>
"""
```

Listing A.1: Prompt 1

```

You are an expert rhino3d designer who can work with rhino3d api,
the user will give you instructions about an action in rhino and you should give a list of 'RhinoScript' for
running in RhinoCommon framework 'RunScript' method, list the commands in a json.
Take these steps to make sure you generate the proper script.
1. Find the steps needed for the instruction to be executed.
2. Find the proper command for each step
3. Make sure you are using the correct parameters and their orders for each command.
4. Genereate the command for each step and fill in the unknown values for parameters with default values.
5. structure the commands in a json format.
The following examples are here to help you with the structure of the output:
1. user instruction: "please create a sphere" >> output: "{{"
```

```
RhinoScript" ["_-Sphere 0,0,0 1"]}}"
2. user instruction: "I want an array of the last object created" >> output: "{{"RhinoScript": [ "_-SelLast", "_-ArrayLinear _Count=5 _Direction=1,0,0 _Distance=10 _DeleteInput=No"]}}"
3. user instruction: "Create an ellipse with a major axis of 15 units and a minor axis of 10 units." >> output: "{{"RhinoScript": ["_-Ellipse 0,0,0 0,15,0 10,0,0"]}}"
The output should follow the structure above.
This is user's request: << {user_instruction} >>
Please make sure it is a valid syntax and works if I feed it to the rhino command line!
```

Listing A.2: Prompt 2

```
You are an expert rhino3d designer who can work with rhino3d api,
the user will give you instructions about an action in rhino and you should give a command of 'RhinoScript' for
running in RhinoCommon framework 'RunScript' method, put the command in a json.
Take these steps to make sure you generate the proper script.
1. Find the steps needed for the instruction to be executed.
2. Find the proper command for each step
3. Find the proper measurements based on the users instruction and assign it to the parameters.
4. Make sure you are using the correct parameters and correct order of parameters for each command.
5. Genereate the command for each step and fill in the unknown values for parameters with default values.
6. structure the commands in a json format.
7. If you have any notes or description about the commands, put it in the 'description' note in the json.
8. Reply in a JSON format.
The response format should follow the following JSON format. It has to be a json without any notes. If any notes are needed they
```

```
   should be put in the 'description' object of the JSON.
Examples:
These are a few examples of the user instructions and the proper
   RhinoScript output:
{{
  "Examples": [
    {{
      "user_instruction": "please create a sphere",
      "output": {{"RhinoScript": ["_-Sphere 0,0,0 1"]}}"
    }},
    {{
      "user_instruction": "Create an ellipse with a major axis of 15
    units and a minor axis of 10 units.",
      "output": {{"RhinoScript": ["_-Ellipse 0,0,0 0,15,0 10,0,0"]}}
    }}
    {{
      "user_instruction": "Lock all objects on the 'Furniture' layer
   .",,
      "output": {{"RhinoScript": ["_-Layer _Lock Furniture]}}
    }}
    ]
}}

This is user's request: << {user_instruction} >>
Please make sure it is a valid syntax and works if I feed it to the
   Rhino Command Line Interface.
ouptut format(JSON):
{{ "RhinoScript": ["_-Sphere 0,0,0 1"],
 "description": "It is an sphere centering at point 0,0,0 with
   radius of 1 unit. The steps needed are 1.Using the Cone command,
   2. Choosing the center point, 3. Entering the radius. The command
    used is Sphere and the parameters are the center point and the
   radius like _-Sphere center radius"
```

```
}}
```

Listing A.3: Prompt 3

```
You are an expert Rhino 3D designer who can work with Rhino 3D API,
the user will give you instructions about an action in Rhino and you should give a command of 'RhinoScript' for
running in the RhinoCommon framework 'RunScript' method, put the command in a JSON.
This is the probable related context for the request.
{context}
Take these steps to make sure you generate the proper script.
1. Find the steps needed for the instruction to be executed.
2. Use the context to figure out the parameters and steps of the command needed.
3. Find the proper measurements based on the user's instructions and assign them to the parameters.
4. Make sure you are using the correct parameters and the correct order of parameters for each command.
5. Generate the command for each step and fill in the unknown values for parameters with default values.
6. structure the commands in a JSON format.
7. If you have any notes or description about the commands, put it in the 'description' note in the json.
The response format should follow the following JSON format. It has to be a json without any notes. If any notes are needed they should be put in the 'description' object of the JSON.
{{ "RhinoScript": ["_-Sphere 0,0,0 1"], "description": "It is a sphere centering at point 0,0,0 with radius of 1 unit. The steps needed are: 1.Using the Cone command, 2. Choosing the center point, 3. Entering the radius. The command used is Sphere and the parameters are the center point and the radius like _-Sphere center radius"}}
These are a few examples of the user instructions and the proper
```

```
    RhinoScript output:
{{
  "Examples": [
    {{
      "user_instruction": "please create a sphere",
      "output": {{"RhinoScript": ["_-Sphere 0,0,0 1"]}}"
    }},
    {{
      "user_instruction": "Lock all objects on the 'Furniture' layer
    .",
      "output": {{"RhinoScript": ["_-Layer _Lock \"Furniture\"
    _Enter]}}
    }},
    {{
      "user_instruction": "Create an ellipse with a major axis of 15
     units and a minor axis of 10 units.",
      "output": {{"RhinoScript": ["_-Ellipse 0,0,0 0,15,0 10,0,0"]}}
    }}
    ]
}}
This is user's request: << {input} >>
Please make sure it is a valid syntax and works if I feed it to the
    Rhino Command Line Interface.
ouptut format(JSON):
{{ "RhinoScript": ["_-Sphere 0,0,0 1"],
 "description": "It is an sphere centering at point 0,0,0 with
    radius of 1 unit. The steps needed are 1.Using the Cone command,
    2. Choosing the center point, 3. Entering the radius. The command
     used is Sphere and the parameters are the center point and the
    radius like _-Sphere center radius"
}}
```

Listing A.4: Prompt 4

APPENDIX B: SOURCE CODE

```
using System;
using System.Collections.Generic;
using Rhino;
using Rhino.Commands;
using Rhino.Geometry;
using Rhino.Input;
using Rhino.Input.Custom;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Drawing;
namespace OpenAiCommandHelper
{
    [Rhino.Commands.CommandStyle(Rhino.Commands.Style.ScriptRunner)]
    public class OpenAiCommandHelperSeries : Command
    {
        public OpenAiCommandHelperSeries()
        {
            // Rhino only creates one instance of each command class defined in a
            // plug-in, so it is safe to store a refence in a static property.
            Instance = this;
        }

        ///<summary>The only instance of this command.</summary>
        public static OpenAiCommandHelperSeries Instance { get; private set; }

        ///<returns>The command name as it appears on the Rhino command line.</returns>
```

```
        public override string EnglishName => "OpenAiCommandHelperCommand";
        protected override Result RunCommand(RhinoDoc doc, RunMode mode)
        {
            string userInstructions = GetUserInstructions();
            var tokens = userInstructions.Split(new char[] { ' ', '\t', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);
            RhinoApp.WriteLine($"{tokens.Length}");

            if (string.IsNullOrWhiteSpace(userInstructions) || tokens.Length < 2)
            {
                RhinoApp.WriteLine("No user instructions provided. Aborting.");
                return Result.Failure;
            }
            var ScriptResult = CallPython(userInstructions);
            try
            {
                RhinoApp.WriteLine("Generated Result!: ");
                RhinoApp.WriteLine(ScriptResult);
                var res = RhinoApp.RunScript(ScriptResult, true);
                RhinoApp.WriteLine(res.ToString());
                CreateTextInTopView(doc, userInstructions);
            }
            catch (Exception e)
            {
                CreateTextInTopView(doc, "\n Error! \n Generated Command: " +ScriptResult + "\n Error: " +e.Message);
                RhinoApp.WriteLine(e.Message);
            }
            return Result.Success;
```

```
        }
        private string GetUserInstructions()
        {
    // Implement your logic to obtain user instructions as a string
    // You can use Rhino's GetString or any other method to collect user input
    // For example:
            string prompt = "Enter your instructions in double quotes: ";
            GetString getStr = new Rhino.Input.Custom.GetString();
            getStr.SetCommandPrompt(prompt);

            if (getStr.GetLiteralString() == GetResult.String)
                return getStr.StringResult();

            return null; // Return null if user input is canceled
        }


        private string CallPython(string userInstruction)
        {
            const string cmd = "bash";
            const string activateConda =
                "source ./anaconda3/bin/activate";
            const string activateVenv = "conda activate openaienv";
            string scriptPath = "./call_openai.py";
            var pythonCommand =
            $"python \"{scriptPath}\" \"{userInstruction}\"";
            try {
                var startInfo = new ProcessStartInfo
                {
                    RedirectStandardOutput = true,
                    RedirectStandardInput = true,
```

```
                    RedirectStandardError = true,
                    UseShellExecute = false,
                    CreateNoWindow = true,
                    //Arguments = args,
                    FileName = cmd,
                    //WorkingDirectory = workingDirectory
                };
                RhinoApp.WriteLine("trying to run python script!");
                var process = Process.Start(startInfo);
                if (process == null)
                    RhinoApp.WriteLine("Could not start process");
                var sw = process.StandardInput;
                if (sw == null)
                    RhinoApp.WriteLine("Could read the input");
                if (sw.BaseStream.CanWrite)
                {
                    sw.WriteLine(activateConda);
                    sw.WriteLine(activateVenv);
                    RhinoApp.WriteLine("source activated!");
                    sw.WriteLine(pythonCommand);
                    sw.Flush();
                    sw.Close();
                }
                else
                    RhinoApp.WriteLine("Could read the input");
                var sb = new StringBuilder();
                while (!process.HasExited)
                    sb.Append(process.StandardOutput.ReadToEnd());
                var error = process.StandardError.ReadToEnd();
                if (!string.IsNullOrEmpty(error))
                {
                    RhinoApp.WriteLine($"Error: \n{error}");
                    throw new Exception($"{error}");
```

```
                }
                return sb.ToString();
            } catch (Exception e)
            {
                RhinoApp.WriteLine(e.Message);
                return null;
            }
        }
    }
}
```

Listing B.1: RhinoCommon Plug-in

```
import openai
from dotenv import load_dotenv, find_dotenv
import os
import sys
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain.output_parsers import ResponseSchema,
    StructuredOutputParser
from langchain.schema import StrOutputParser
from langchain.vectorstores.chroma import Chroma
from langchain.schema.vectorstore import VectorStoreRetriever
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import JsonOutputParser

# Specify the path to your .env file
env_file_path = '.env'  # Update this with the correct path
source_path = __file__

# Get the directory of the current file
source_dir = os.path.dirname(source_path)

from dotenv import load_dotenv, find_dotenv
```

```
_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']
db = Chroma(
    persist_directory=source_dir + "/chroma",
    embedding_function=OpenAIEmbeddings(model="text-embedding-ada
   -002"),
)

def load_prompt(prompt_file):
        with open(prompt_file, 'r') as file:
                prompt = file.read().replace('\n', '')
                return prompt

def no_context(user_instruction, model):
    parser = JsonOutputParser()
    template_string = load_prompt(source_dir+'/Project/prompts/3.txt
   ')
    prompt = PromptTemplate(
    template=template_string,
    input_variables=["user_instructions"],
    partial_variables={"format_instructions": parser.
   get_format_instructions()},
)
    chain = prompt | model | parser
    result = chain.invoke({'user_instruction':user_instruction})
    print(result['RhinoScript'][0])

def with_context(user_instruction, model):
        parser = JsonOutputParser()
        response_schemas = [
            ResponseSchema(name="RhinoScript", description="the
   runnable script for Rhino3d CLI"),
            ResponseSchema(
```

```
                name="description",
                description="The description for the script and the parameters",
            ),
        ]
        output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
        template_string = load_prompt(source_dir+'/Project/prompts/4.txt')

        command_question_template = """You are an expert rhino3d designer who can work with rhino3d  command line api.
            Based on the user instruction, what is the top 2 best RhinoScript "command line" command that can be used to complete this request in three Command keys?
            And what shape better describes the request in one word?
            {input}
        """
        context_prompt = PromptTemplate(
        template=command_question_template,
        input_variables=["context", "input"],
        partial_variables={"format_instructions": parser.get_format_instructions()},
        )
        retriever: VectorStoreRetriever = db.as_retriever(search_kwargs={'k': 3})
        context_chain = context_prompt | model | StrOutputParser()
        context_result = context_chain.invoke({'input': user_instruction})
        docs = {"context": retriever.get_relevant_documents(context_result)}
        command_prompt = PromptTemplate(
        template=template_string,
```

```
        input_variables=["input"],
        partial_variables={"format_instructions": parser.get_format_instructions()},
        )
        retrieval_chain_2 = command_prompt | model | output_parser
        result_2 =retrieval_chain_2.invoke({'input': user_instruction, 'context': docs})
        print(result_2['RhinoScript'][0])

if __name__ == '__main__':
    try:
        user_instruction = sys.argv[1]
        llm_model = "gpt-3.5-turbo" if len(sys.argv) < 3 else sys.argv[2]
        model = ChatOpenAI(temperature=0.0, model=llm_model)
        with_context(user_instruction, model)

    except Exception as e:
        print('ERROR', e)
```

Listing B.2: NLP Python

APPENDIX C: TEST CASES

1	"Draw a vertical line from (0,0,0) to (0,0,10)."
2	"Create an ellipse with a major axis of 15 units and a minor axis of 10 units."
3	"Switch the top viewport to wireframe mode."
4	"Duplicate the selected object and translate it 10 units in the X-axis and 5 units in the Y-axis."
5	"Extrude the selected curve 15 units along the global Y-axis."
6	"hide all objects on a specific layer named 'Temp Layer'."
7	"Sweep a selected profile curve along a helix-shaped rail."
8	"Loft between these three curves."
9	"Revolve this curve around the Z-axis."
10	"Measure the radius of this sphere."
11	"Taper a box shape by a specified angle along the Z-axis."

Table C.1: Development Test Cases

BasicGeometryCreation

1 "Draw a straight line from (2,3,0) to (5,8,0)."

2 "Create a circle centered at (4,4,0) with a radius of 3 units."

3 "Construct a regular hexagon with each side measuring 4 units."

4 "Place a point at (7,7,7)."

5 "Sketch a curve passing through (0,0,0), (3,3,3), and (6,0,0)."

6 "Generate an equilateral triangle with a side length of 5 units at (2,2,0)."

7 "Create a rectangular plane with dimensions 4x6 units at the origin."

8 "Draw an arc with a radius of 3 units from angle 0 to 90 degrees."

9 "Create a polyline connecting points (0,0,0), (2,2,2), and (4,4,4)."

ViewAndDisplay

10 "Switch the current viewport to wireframe mode."

11 "Display all layers that are currently hidden."

12 "Zoom extents to fit all objects within the current viewport."

13 "Set the display mode to shaded for the current view."

14 "Activate the perspective view as the current viewport."

15 "Highlight all objects on the layer named 'Site Plan'."

16 "Switch to rendered view mode in the active viewport."

17 "Turn on the grid display for the top view."

18 "Set the view to show object edges as dashed lines."

19 "Enable the display of surface control points."

BasicObjectEditing

20 "Mirror the selected object across the global Y-axis."

21 "Copy and move the selected geometry 5 units along the Z-axis."

22 "Split a solid object using the selected cutting plane."

23 "Offset a selected polyline inward by 1 unit",

24 "Delete the currently selected free-form curve."

25	"Create a blend between two open curves."
26	"Scale the selected object down by 50%."
27	"Rotate the selected object 90 degrees around the global X-axis."
28	"Move the selected objects to layer 'Draft'."
	LayersAndAnnotation
29	"Move the selected objects to layer 'Draft'."
30	"Lock the layer named 'Annotations'."
31	"Create a new layer named Electrical with a red color."
32	"Assign the selected objects to the 'Mechanical layer'."
33	"Change the line type of the selected curves to 'Hidden'."
34	"Set the print width of the selected objects to 0.25 mm."
35	"Isolate all objects on the 'Landscaping' layer."
36	"Rename the 'OldLayer' to 'Renovations'."
37	"Change the object color of selected items to blue."
38	"Lock all objects on the 'Furniture' layer."
39	"Turn off the visibility of the 'HVAC' layer."
40	"Dimension the distance between two selected points."
41	"Label the midpoint of the selected line with text 'Midpoint'."
42	"Show the area of the selected closed polyline."
43	"Place a dimension annotating the radius of the selected circle."
44	"Create a leader pointing to the selected object with text 'Main Entrance'."
45	"Measure and display the angle between two intersecting lines."
46	"Add a text block at point (5,5,0) saying 'Start Point'."
47	"Annotate the length of the selected curve."
48	"Place a linear dimension along the selected edge."

Table C.2: Evaluation Test Cases

APPENDIX D: GITHUB

openai_command_helper_plugin