MoGraphGPT: Creating Interactive Scenes Using Modular LLM and Graphical Control

Hui Ye, Chufeng Xiao, Jiaye Leng, Pengfei Xu, and Hongbo Fu

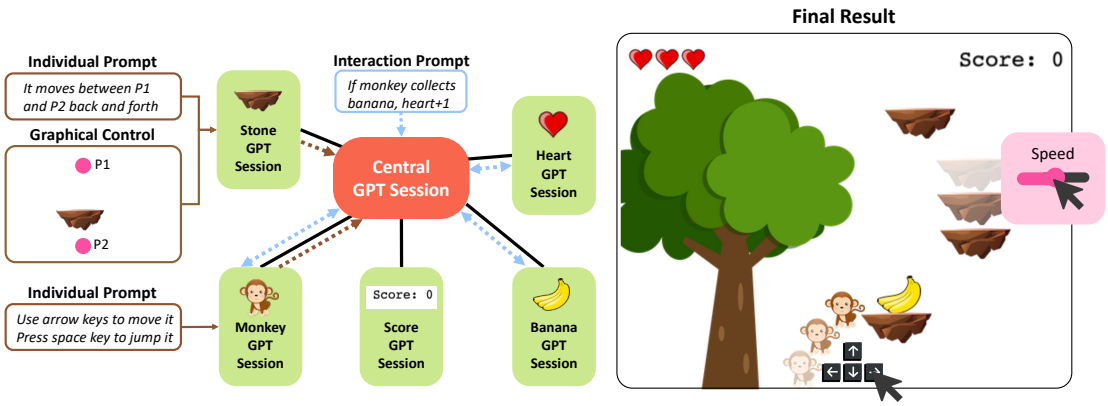


Fig. 1: We introduce *MoGraphGPT* to facilitate the easy creation of interactive scenes using a graphical user interface powered with modular LLMs. When users input text description for individual element behavior (e.g., control the monkey movement using keys), the central session will let the individual session generate code directly. If users specify graphical information in the canvas (e.g., the platform moves between the two points), it will be translated into coordinates together with text input for generation. For interactions among several elements, the central session will analyze and decompose the interaction requirement into multiple sub-tasks, and then dispatch a specific sub-task of code generation or updates to the corresponding individual sessions, and coordinate the implementations. The generated behavior or interaction parameters will be used to automatically generate sliders via an LLM to precisely control the effects (e.g., platform moving speed).

Abstract—Creating interactive scenes often involves complex programming tasks. Although large language models (LLMs) like ChatGPT can generate code from natural language, their output is often error-prone, particularly when scripting interactions among multiple elements. The linear conversational structure limits the editing of individual elements, and the lack of graphical and precise control complicates visual integration. To address these issues, we integrate a context-aware modularization technique that processes textual descriptions for individual elements through separate LLM modules, with a central module managing interactions among elements. It defines a top-down structure to manage interactions, ensuring clear update logic and facilitating efficient collaboration while allowing for independent updates for each element. We design a graphical user interface, *MoGraphGPT*, which combines modular LLMs with enhanced graphical control to generate codes for 2D interactive scenes. It enables direct integration of graphical information and offers quick, precise control through automatically generated sliders. A comparative study with Cursor Composer shows *MoGraphGPT* significantly improves easiness, controllability, and performance in creating 2D interactive scenes with multiple visual elements in a coding-free manner. An ablation study validates the effectiveness of modularization, and an open-ended study demonstrates the usability and expressiveness of *MoGraphGPT*.

Index Terms—Code Generation, Modularization, Large Language Models, ChatGPT, Graphical Control, Interactive Scenes.

1 Introduction

Interactive scenes combine artistic visuals with interactive elements, immersing users in richly crafted environments. They are widely used in video games, educational tools, and interactive demonstrations. Creating interactive scenes often requires coding with engines like Unity or frameworks like Phaser, requiring skills in scripting interactivity, managing animations, and debugging. This complexity is essential for delivering engaging user experiences but poses significant

Corresponding author: Hongbo Fu.
Hui Ye is with the Department of Interactive Media, School of Communication, Hong Kong Baptist University, and Division of Arts and Machine Creativity, Hong Kong University of Science and Technology. E-mail: huiyehy@hkbu.edu.hk.
Chufeng Xiao is with the Hong Kong University of Science and Technology. Email: chufengxiao@outlook.com.
Jiaye Leng is with the School of Creative Media, City University of Hong Kong. E-mail: jiayeleng2-c@my.cityu.edu.hk.
Pengfei Xu is with the College of Computer Science and Software Engineering, Shenzhen University. E-mail: xupengfei.cg@gmail.com.
Hongbo Fu is with the Division of Arts and Machine Creativity, Hong Kong University of Science and Technology. E-mail: fuplus@gmail.com.

challenges for beginners [1], [2].

Recently, Large Language Models (LLMs) like ChatGPT have emerged as powerful tools, enabling users to generate code from natural text. They can assist users in writing scripts [3], creating simple animations [4], and even debugging [5], significantly reducing the time and effort required for coding tasks [6]. Some ad-hoc tools for code generation and refinement like GitHub Copilot and Claude Artifacts have also been proposed to provide real-time suggestions.

However, directly using LLMs to generate code for interactive scenes may have four main issues: (1) **code quality** – LLM-based code generation approaches [7]–[9] may produce incomplete or incorrect code sometimes. They often require a clear context to generate relevant code, which may be challenging in complex projects. Generating interactive scenes usually requires scripting how elements in a scene interact with each other and how users interact with the scene. The logic and relationship can become complex, especially when multiple elements are involved. The generated code tends to be error-prone, so users need to provide extra text input to refine it. (2) **lack of editing independency** – the linear conversational nature of LLMs limits independent editing of individual elements, especially for non-expert users [10]. The models may forget previous interactions, leading to disjointed responses, especially when the conversation is long and involves many scene elements. The modular code generation and refinement in emerging AI tools often require understanding project and code structures for hard modularization, otherwise the updated results may intertwine codes across other files to produce unintended modifications. (3) **lack of graphical control** – it is difficult to directly integrate graphical information into the text input. Users need to manually estimate the exact graphical information (e.g., position, size, path, region) and translate it into textual input, which is not intuitive and direct [11]. (4) **lack of precise control** – fine modifications on the generated effects (e.g., effect parameters) require users to adjust text prompts iteratively [12], which usually traverse users between excessive and inadequate controlling.

Through a content analysis of online tutorial videos and a review of existing code generation tools, we identified significant challenges in creating 2D dynamic scenes. To address this, we developed *MoGraphGPT* (Figure 1), a no-code interactive system leveraging modular LLMs and graphical controls. We propose a context-aware modularization technique that assigns dedicated LLMs to individual elements. A central LLM then analyzes the overall logic and dispatches update instructions to coordinate these elements. To bridge the context gap, individual modules distill variable and function details as context for the central LLM, ensuring accurate interaction code generation. For graphical controls, our *MoGraphGPT* supports four types of graphical proxies, i.e., points, lines, curves, and regions, which can be defined via direct pointing or drawing and explicitly referenced in text prompts. Furthermore, the system can automatically generate sliders given the generated code, allowing users to interactively and precisely fine-tune effect parameters.

We define "interactive scenes" in this paper as 2D dynamic visual content composed of: single or multiple elements with dynamic and interactive behavior driven by rigid animation, user input (keyboard, mouse), element interactions (collision, triggering), or environment interactions (boundary conditions). This scope ranges from simple animated demos to more complex, fully-fledged 2D games and other interactive applications.

This work makes the following main contributions:

- A content analysis of video tutorials on creating interactive scenes using ChatGPT and existing AI tools for code generation.
- The integration of context-aware modularization for controlling independent code generation for individual elements in individual LLM modules, as well as interaction generation and overall management for all elements based on element context in the central module.
- A graphical interface allowing users to create interactive scenes by inputting text prompts to modular LLMs, integrating graphical information into prompts directly, and iteratively adjusting generation results and parameters using additional text inputs and sliders precisely.
- Three evaluations that compare the performance of *MoGraphGPT* and Cursor Composer and validate the system usability, controllability, and expressiveness.

2 Related Work

2.1 LLM-based Code Generation and Improvement

LLM-based code generation leverages advanced language models [8], [9] to translate natural language prompts into executable code. Although these models are powerful at streamlining software development and enhancing productivity, many issues arise from the generated code, including compilation and syntax errors, wrong outputs, maintainability problems, not following standard coding practice, needing refactoring, security smells, etc. [6], [13]. In particular, ChatGPT struggles to generate code for new and unseen problems [6]. Lengthy prompts may negatively affect code generation [6], [13]. These issues could be due to the increased code complexity for more complex problems, making it harder for LLM models to generate a correct and complete solution [13].

Chain-of-Thought [14], [15] makes LLMs more controllable for complex tasks. For coding, decomposition strategies include interactive decomposition [**?**], block-based hierarchical structure [16], and node-based diagrams [17]. CoLadder [18] enables flexible task decomposition. These works focus on general programming tasks involving static code generation. Modularized LLM generation mechanisms like Tree-of-Thoughts [**?**], [19], [20] focus on general thought exploration. We target interactive scene creation, integrating textual and graphical inputs with LLM code generation for intuitive element behavior and interaction control. Agentic workflows [21], [22] achieve agent-level modularization across multiple roles (Table 1). In contrast, our context-aware top-down structure enables a central LLM module to manage interactions among independent elements, facilitating clearer update logic and independent module updates. Kim et al. [23] explore modularized configuration generation for writing tasks. We extend this to interactive scene creation where elements act as objects, addressing interaction challenges by encapsulating code within each element's module while invoking interaction logic centrally.

TABLE 1: The comparison among the closely related works.

	Task	Method	Multiple Element Interaction	Code Modularization	Graphical Control	Precise Control
MoGraphGPT	Interactive scene creation	LLM-based	Yes	Yes	Yes	Yes
DrawTalking [24]	Interactive scene creation	Rule-based	Yes	No	Yes	No
Spellburst [25]	Generative art creation	LLM-based	No	Yes	No	Yes
DirectGPT [11]	Text/code/vector images edition	LLM-based	No	No	Yes	No
ChatDev [21]	General software development	LLM-based	Yes	Yes	No	No
Cursor [26]	General programming	LLM-based	Yes	Yes	No	No

2.2 LLM-powered Tools for Visual Design and Development

LLMs have been employed to enhance visual design and development processes [27], [28] and generate 2D static visual designs [29], [30]. Designing dynamic 2D visuals with LLMs presents greater challenges, as animations require consideration of timing, motion, and interaction [31]. Keyframer [32] leverages LLMs to create animations by generating keyframes from textual descriptions. LogoMotion [**?**] develops an LLM-based system that automatically generates content-aware logo animations by synthesizing code from visual layouts. Spellburst [25] introduces a node-based interface for exploring creative coding through natural language prompts for interactive visual design with precise parameter control (Table 1). While these works enhance LLMs in animation and dynamic visual design, they do not address interactions between multiple elements, overlooking independent control challenges. v0 [33] generates interactive and dynamic effects but lacks clear control for individual elements and graphical information.

Researchers alsp explored the potential of LLMs in game content design, including investigating the use of video game description language [34], automated level design and generation from text [**?**]. They aim to enhance the game content design process, making it more accessible for creators to design game experiences. Differently, we provide an LLM-based solution, allowing users to interactively create complete games without coding. Some studies further explore the potential of LLMs in application development, including using communicative agents to support software development [21] and providing an AI-augmented system for kids' autonomous visual programming learning [35]. Compared to them, our work focuses on a novel system for creating 2D interactive scenes.

2.3 Supporting Creating 2D Interactive Scenes

The traditional practice of creating 2D interactive scenes requires artists to create frame-by-frame animations and integrate them with programming to enable user interactions and responses. To simplify the production process [36], previous works introduce sketch-based interactions to animate virtual elements [37], manipulate kinetic textures [38], and use filters for dynamic illustrations [39]. Besides dynamic effects, Kitty [40] enhances the sketching interface for interactive illustrations, which involve more user interactions. These works are powerful for creating diverse, fascinating, dynamic effects, but expect users to have drawing skills for specifying animation effects. Recently, several tools employ visual programming interfaces using blocks [41], nodegraphs and flowcharts [2] to build interactive scenes. For example, as a pioneering platform, Scratch [42] enables users to create animations and games through a block-based coding environment. Our graphical interface is largely inspired by Scratch's interface. However, instead of explicit coding in Scratch, our system uses LLMs to generate code given natural language text inputs, aiming to lower the barriers to creating interactive scenes.

DrawTalking [24] is closely related to our work and enables the creation of interactive worlds using sketching and speaking. The main difference between DrawTalking and our work is that we introduce LLMs to generate codes for scenes automatically (Table 1), while their dependency-tree structure is difficult to replace directly with an LLM model, a point confirmed by the authors of DrawTalking. The rule-based interactions in DrawTalking can involve multiple effects, but lack flexibility and generalization for scripting customized effects. In addition, the natural language input in DrawTalking needs to conform to the rules, while ours allows users to describe desired scenes freely due to the understanding capability of ChatGPT.

2.4 Interacting with LLM from Input and Output

Recently, researchers have explored interacting with LLMs' inputs and outputs to enable greater controllability. For example, DirectGPT [11] offers an intuitive interface for users to engage directly with LLMs, making it easier to input prompts with direct manipulation. It focuses on a task different from ours, i.e., editing text, code, and vector images, so it does not address dynamic interactions among multiple elements or allow for precise adjustment of editing parameters (Table 1). Graphologue [43] enhances this interaction by allowing users to explore LLM responses through interactive diagrams, which help visualize the relationships and structures within generated content, thus clarifying complex ideas. Meanwhile, Visual ChatGPT [44] combines conversational capabilities with visual foundation models, enabling users to talk, draw, and edit visuals simultaneously, creating a richer and more dynamic engagement experience. ChainForge [45] serves as a visual toolkit for prompt engineering and hypothesis testing, enhancing the overall interaction with LLMs. Compared with these works, we use graphical control and direct manipulation to specify both inputs and outputs. For input, we integrate graphical information (by specifying or drawing visual proxies) into text prompts. For output, we allow users to quickly refine the effect parameters via sliders and value inputs.

3 Formative Steps

To better understand the challenges of utilizing LLMs to create interactive scenes, we employed a mixed-method

approach: (1) content analysis on video tutorials about using ChatGPT to generate codes for building interactive scenes; (2) further analysis of existing AI coding tools.

3.1 Content Analysis on Video Tutorials about Creating Interactive Scenes Using ChatGPT

Due to the lack of well-developed workflows and criteria for creating interactive scenes using LLMs, it is not easy to understand the existing challenges according to experts' experiences. So we resort to online videos – many users uploaded video tutorials documenting their practical experience in using LLMs to create games or make interactive demos for popular video platforms (e.g., "Can AI code Flappy Bird? Watch ChatGPT try"[3]). We want to distill valuable insights from their videos.

Corpus and Methodology. We searched for the videos on popular video websites (e.g., YouTube, Vimeo, TikTok) using keywords such as "GPT for interactive scenes", "GPT for games", "GPT for animations", and "GPT for dynamic effects". Through the first round of searching, we collected 208 video clips. After a thorough filtering process, we retained 56 videos that specifically guided or demonstrated how to use GPT to build a complete interactive or dynamic scene. All the videos are in English and feature a single speaker. The styles include full-screen screencasts, screen recordings with annotations, tutorial formats, and picture-in-picture screencasts. The duration of the videos ranges from 4 minutes 13 seconds to 26 minutes 48 seconds. The programming languages covered in these videos include C#, JavaScript, Python, GML, Lua, and Scratch pseudocode. Two of our authors employed the open-coding approach [46] to analyze the content of the selected videos. We began by watching each video multiple times to understand its content comprehensively. In the initial coding phase, we identified explicit difficulties mentioned in the videos, as well as challenges reflected in the creation process. We focused on common issues faced by those users when using GPT for creating interactive scenes, their strategies to overcome these difficulties, and the problems that persisted even after applying these methods. We developed a coding framework that categorized the identified difficulties and strategies into several themes. We then compared individual results and summarized the findings into overarching themes. The coding process was iterative, allowing for refinement as new insights emerged.

Findings. We distilled three main issues as follows.

Independent generation and refinement (29/56). Three types of strategies are mainly used for generating codes: (1) describing all the elements in the scene at one time and then iteratively adjusting the results; (2) describing each element and element interactions step by step and manually adjusting the code snippet of different elements; (3) describing the scene and asking ChatGPT to implement a basic version as the start, and then adding more features iteratively. The first strategy does not require much programming understanding, but it may produce incorrect results. Refining one element would sometimes affect other elements. For example, in reproducing the Super Mario game, adding a moving feature to one platform might also make another static platform move. This is due to ambiguous references and a limited understanding of ChatGPT. The second and third strategies require coding skills to some degree. Sometimes GPT generates incorrect results, so users need to use their prior knowledge to correct them. The dependent results will also require manual adjustment and differentiation.

Graphical control (42/56). To enable GPT to generate graphical scene interfaces or demos, there are three types of strategies to employ: (1) let GPT generate graphic effects using simple descriptions and then adjusting them using text prompts or manual coding refinement; (2) prepare a 2D snapshot of a desired graphical scene with accurate graphical information of each element; (3) copy and paste the generated code to game engines like Unity and manipulate elements. The first strategy often produces random results – even different for every trial. Users need to use wording like "make the rectangle larger" and "move it to the top" to adjust the accurate properties. If users would like to make a random game for fun, this strategy could be acceptable. Otherwise, they have to bear a tedious adjustment process. The second strategy requires users to make a lot of preparation effort and then translate this graphical information into text. Some information, like user-defined curved paths, is very difficult to describe in text. So, they only use a rough representation of the results they create. In the third strategy, users need a good understanding of both coding and game engines. For those without coding skills, filling the gap between the generated code and the manipulation in game engines requires GPT to guide users step by step to find the correspondence.

Precise refinement (27/56). Once the scene code is generated, users may refine the specific effects, such as the moving speed and the rotation radius of elements. In ChatGPT, they input the text again using comparative expressions (e.g., make Mario jump lower each time the space key is pressed, let the star move slower). However, they usually do not have an intuitive understanding of the magnitude of the parameters. So, they need to refine it back and forth to achieve the desired effect.

In summary, the analysis identified three main challenges in creating interactive scenes using LLMs: the difficulty of independent generation and refinement of code, the lack of graphical control requiring extensive manual adjustments, and the need for precise parameter adjustments due to users' limited understanding of effect magnitudes.

3.2 Further Analysis on Existing AI Coding Tools

Methodology. We collected and reviewed six common AI coding tools (i.e., GitHub Copilot, Cursor, Tabnine, Codeium, Replit Ghostwriter, and Amazon CodeWhisperer). For the analysis of these tools, we mainly focus on the three distilled issues in Section 3.1.

Findings. These tools provide functions for generating code snippets or blocks, either independently or considering the entire file context. However, implementing individual elements in an interactive scene, such as those in game development, often requires creating entire classes.

The challenge lies in constructing these classes and maintaining the interactions among them. While these tools can generate basic class structures, they typically lack support

3. https://www.youtube.com/watch?v=8y7GRYaYYQg

for managing complex interactions, such as communication between a player character class and an enemy class. Therefore, users often need to manually refine and adjust the generated code to ensure that the components work together effectively, demanding a solid understanding of object-oriented programming principles. Some tools, like Cursor [26], support modular code generation and refinement by selecting project files as context to constrain the generation. However, it performs modular code generation in a soft manner. In other words, it treats them as contexts, which can lead to unintended modifications across other files. Such confusion can intertwine interaction code and individual behavior code, causing inconsistent updates and chaotic modifications. For users seeking more controllable hard modularization, a solid understanding of the entire project structure and code framework is essential with Cursor, creating a significant barrier to entry.

Since these tools are primarily designed for general programming tasks, none of them supports graphical control of the scripted elements, making it difficult to create interactive scenes directly and intuitively. This limitation forces users to rely solely on text-based refinement, which can be cumbersome when managing complex visual components. Without a graphical interface, users cannot easily manipulate or visualize elements in real-time, leading to a tedious trial-and-error process. The precise control also relies heavily on text-based adjustment. While these tools can provide suggestions for modifying element properties, they lack the ability to manipulate graphical elements directly. This means users must translate their visual intentions into code, which can be a time-consuming process, especially for intricate designs. They may spend significant time fine-tuning parameters through trial and error rather than simply dragging and dropping elements or adjusting them visually.

3.3 Design Considerations

Based on the above findings, we envision that an ideal LLM-based tool for creating interactive scenes should consider:

D1. Independent code generation and control on elements: refining individual elements does not affect others.

D2. Context-aware code generation: independency will not lose context and coordination.

D3. Graphical control: integrating graphical information directly into text prompts.

D4. Easy and precise parameter control: direct manipulation of effect parameters.

4 MoGraphGPT System

According to the design considerations, we first integrate a context-aware *modularization* technique (D1, D2) to help generate code for individual elements and interactions for multiple elements (Section 4.1). We further design and develop a graphical interface (Figure 1), *MoGraphGPT*, combining modular LLMs with graphical control for users to create 2D interactive scenes (Section 4.2). It enables direct integration of graphical information (D3) and offers quick, precise control through automatically generated sliders (D4).

In our system, a 2D interactive scene is organized around elements and their interaction patterns on a canvas. We support both individual elements and grouped elements, where the former correspond to single visual assets or primitives (e.g., a character, a platform), whereas grouped elements denote higher-level constructs that combine multiple sub-elements into one logical unit (e.g., a character composed of body parts, or a platform assembly). For each element or group, the system maintains its current spatial state (such as position, scale, and orientation) and its logical relations to other elements, and supports four interaction patterns among them according to the degree of dependence between elements: (1) independent behaviors: elements driven solely by their own states, (2) unidirectional dependencies: one element depends on the other's state but not vice versa, (3) bidirectional coupling: two elements influence each other, (4) multi-element coordination: multiple elements require synchronized or coordinated behavior. These four patterns can form scenes with different complexities. Building on this representation, our context-aware modularization technique (Section 4.1) generates and refines behaviors for individual elements and multi-element interactions, while the graphical interface (Section 4.2) exposes these elements on a canvas where users can specify intents and adjust effect parameters without writing code.

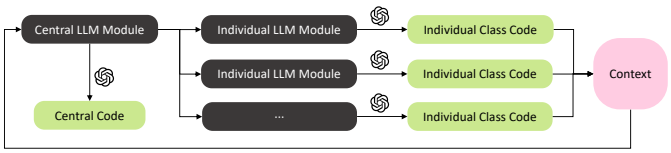


Fig. 2: The framework of our context-aware LLM modularization technique. The central LLM module generates and maintains the central code. It manages individual LLM modules to generate individual class codes. The contextual information is extracted from individual codes and input to the central LLM module for reference.

4.1 Context-aware LLM Modularization

Our context-aware *modularization* technique (Figure 2) opens modular LLMs for individual elements and uses a central LLM module to manage interactions and relationships among elements. It employs a hierarchical structure where the central module oversees coordination, while the individual modules operate independently. This design ensures a clear and cohesive update logic. We employ ChatGPT-4o Mini as our LLM model in our implementation.

Individual LLM Modules for Individual Elements. Each individual element in a 2D interactive scene is associated with its own LLM module. These individual modules are used to generate and maintain class codes for their respective elements. For example, when creating a Super Mario platform game, the Mario element has its own LLM module (Figure 2), which generates a class named Mario for its own properties (e.g., sizes) and behaviors (e.g., using arrow keys to control its movement) from the text input. To modify Mario's properties and behaviors with additional text prompts, the Mario module will locate the previously generated Mario class and update its corresponding code accordingly. This approach allows each element to operate independently, enabling users to customize and enhance

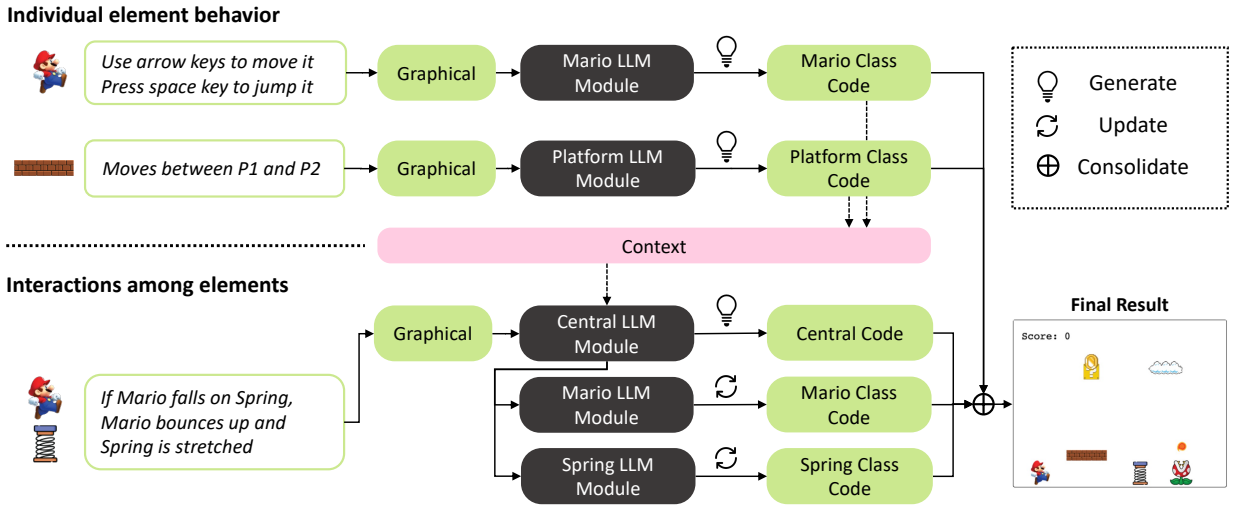


Fig. 3: *MoGraphGPT* workflow. When users input text prompts for individual elements, our system integrates graphical information into prompts and sends them to individual modules to generate class codes (Top). For interactions (Bottom), prompts with the integrated graphical information go to the central LLM, which creates the central code. It then notifies individual LLM modules to update their codes with new variables and functions. Changes are reflected in real-time, and the central and individual codes together form the final result.

each element without disrupting other elements with rapid iteration and testing.

Central LLM Module. The central LLM module serves as the orchestrator of interactions and relationships among elements (Figure 2). It is responsible for instantiating classes from individual modules, coordinating their communication, and managing interactions among elements, thus ensuring that they work together cohesively within the interactive scene. For example, in the Super Mario platform game (Figure 3), the central module generates codes for instantiating all elements and scripting interactions among them (e.g., when Mario falls on the spring, he bounces up, and the spring stretches). Importantly, when generating interaction code, it may involve variables and behaviors specific to individual elements. To prevent interference between elements, we instruct it to define the code of variables and functions for each element within their respective classes (e.g., Mario bouncing code in Mario class, spring stretching code in spring class) and to call these functions in the central module. This approach allows the central module to directly invoke functions from individual modules while keeping their definitions separate. As a result, modifications to individual elements will not impact the interaction code, maintaining the integrity and functionality of the overall system.

Contextual Communication between Modules. Independent code generation will lead to a lack of contextual information. To address this issue, we design a contextual communication mechanism (Figure 2) between the central module and individual modules. Each time the code for an individual element is generated, we guide LLM to also provide a summarized overview of the class, including the class name, variables (name, initial value, and short description), and functions (name, argument, return value, and short description). Please refer to the supplementary materials for more details. Such information is then compiled into a context information repository. When generating the code from the central module, this context is referenced, enabling the central module to maintain an understanding of the overall state of all elements in the scene. It can directly access the variables and call the functions defined in the element classes. If a user revises any element class, both its code and context information will be updated accordingly. Additionally, if the central module modifies or updates variables and behaviors for elements, this will also be reflected in the contextual information. This dynamic update ensures that the central module remains aware of all changes, promoting a more responsive and flexible operation.

Every user input is fed into the system, where the central module analyzes, decides, and allocates which modules need to generate or update codes and whether the shared context requires modification. Each affected module is immediately updated to reflect the latest text or graphical input, overwriting previous states so that users can freely experiment and revise without dealing with complex conflict states. Based on this well-constructed modularization and context, our central module can also handle complex coupled logics by decoupling interactions into scalable and simple components that can be more easily understood by the LLM. In practice, complex behaviors that conceptually mix independent, unidirectional, bidirectional, and multi-element coordination patterns are decomposed into multiple minor sub-logics, each involving the minimum number of interactive elements. It analyzes the prompt, identifies these minor sub-logics, and processes them individually, enabling the system to resolve intricate logic chains while maintaining scalability and interpretability.

4.2 Graphical Interface

In *MoGraphGPT* UI (Figure 5), context-aware LLM modularization and graphical control are seamlessly integrated to facilitate the creation of 2D interactive scenes from natural language inputs and graphical specifications. Our target

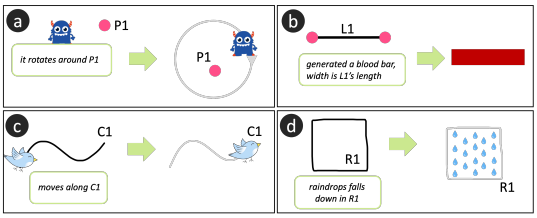


Fig. 4: We allow users to specify four types of graphical inputs: (a) point, (b) line, (c) curve, and (d) region. Users can refer to their names in the text prompts. The green text box represents the user's textual description bound to each element. The green arrow illustrates the before-and-after specification process and its resulting output. The black lines/curves/regions denote the graphical inputs as specified by the user, while the grey lines or boundaries illustrate the actual paths or resulting region outlines.

Fig. 5: *MoGraphGPT* UI. The Element Pane contains buttons and preview images for all the created elements in the scene. The Canvas Area displays all elements that users can manipulate directly. Once users press the "Generate" button, the result is generated or updated in the Result Area. The Effect Control Pane displays the automatically generated parameter values and sliders for precise control.

users are those with no or limited programming skills. To simplify the user experience and avoid overwhelming newcomers, we do not reveal the generated code in the UI. Instead, we focus entirely on prompts and graphical elements, encouraging users to engage with this tool for scene creation rather than transitioning to full programming.

Element Creation. Users have the flexibility to upload, draw, and request our system to generate elements for them. Users can press the "Upload" button to upload an image element and draw elements with 2D sketches on the canvas area. If users have not prepared any images, they can create an empty asset by pressing the "Add" button and then input text descriptions to ask the associated individual module to generate an element for them, such as texts, graphics, and particle effects. Besides single elements, users can also create element groups in two ways: (1) pressing the "Add Group" button, uploading an element image, and adding a text description for generating a group, or (2) uploading an image element and providing a prompt that explicitly indicates a group (e.g., containing the word "group").

Once an element is created, it is displayed in the canvas area (Figure 5) and rendered in the result area (Figure 5). The system opens a dedicated ChatGPT session for that element in the left text pane (Figure 5). Subsequently, when users select an existing element (by pressing its associated button or clicking on it in the canvas), the system automatically switches the GPT session to the selected element's session. The first element in the element pane serves as a central proxy, representing the central session. By clicking on this proxy, users can access the central session in the left pane.

Graphical Control. Once an element is created, the user can move, rotate, and scale it on the canvas. These graphical properties are updated in real time in the generated code, as displayed in the result area. Since describing graphical properties in natural language can be challenging, we introduce a drawing mode allowing users to specify four types of graphical inputs: point, line, curve, and region (Figure 4). They can switch to a certain mode and draw on the canvas. After completing their drawings, each input is labeled with an index, designated as P_i, L_i, C_i, R_i, respectively. They will be represented by their corresponding coordinate information (i.e., the coordinates for points, sequences of coordinates for lines and curves, and boundary curves for regions). These graphical representations are then directly integrated into the prompt as part of the user input for further generation or update. Users can then reference these labels explicitly in their text input, facilitating explicit communication of their graphical specifications.

Text Input. We allow users to input any text to describe the interactive scenes. To generate the properties and behaviors of individual elements, users enter text in the module of each element by selecting the element button, and press the "Generate" button. Then the code for the element is generated and rendered in the result area. For interactions among multiple elements, users input text in the central module by selecting the "Multiple" button, and press the "Generate" button to send their request. Since each element has its own ChatGPT session, users do not need to mention the element names explicitly in the individual sessions. Instead, users can use pronouns such as "it", "each of them", or "all of them" to refer to specific elements.

Precise Refinement. To further support fine-grained control of LLM-generated behaviors, we introduce automatically generated UI controls [47]. After the code for each element is produced, the updated element context is passed to a separate LLM using a dedicated Slider Prompt template which guides the LLM to identify user-adjustable variables and produce appropriate slider ranges. The system then generates sliders and number input fields in the effect control pane (Figure 5), enabling users to quickly and precisely adjust parameters (e.g., movement speed, shake amplitude) without repeatedly refining text prompts. Users can also directly specify slider ranges via a prompt, either before or after the sliders appear, to ensure suitable UI settings. Please refer to the supplementary materials (Part 1.4) for the detailed Slider Prompt.

Result Testing. Users can watch and test the created results in the result area (Figure 5) at any time during the creation process. Any change, including text revision, slider revision, or element manipulation in the canvas area, will lead to instant updates in the result area.

5 Implementation

MoGraphGPT is built using JavaScript for the front-end client, and our back-end uses OpenAI's ChatGPT API, specifically using the GPT-4o Mini model.

5.1 Prompt Design

Code Template Generation. We first ask GPT to generate a template class code with a basic setting according to the JavaScript framework that users will use. The framework is automatically determined for users when they input general descriptions in the central module. For example, if the user inputs "I want to make a platform game", the Phaser framework will be selected; if he/she inputs "make a creative coding project", the p5.js framework will be initialized. When a new element is created, an initial class template code will be added to the scene folder, and the central.js is also automatically initialized with the elements instantiated.

Code Generation for Individual Elements and Multiple Elements. To guide GPT to generate output in a controlled manner, the text prompt follows a structured template format. We designed three template prompts to implement our modularization workflow: (1) Central Prompt, invoked by the central module, takes as inputs the *Current Central-module Code*, the *Context* (contextual pseudocode for each individual module), and *User Prompt*. It enables the central module to orchestrate updates by generating minimally modified main code and task-specific instructions for each relevant individual module; (2) Central2Local Prompt, used by individual modules when receiving the dispatched signal from the central module, gets the explicit update instructions *Sub-effect*, the target *Pseudocode*, and the *Current Code* of the individual module to produce targeted code updating and context synchronization; (3) Local Prompt, applied when only one individual module is involved, takes the *Current Code* and *User Prompt* to update the code and context. Please refer to supplementary materials (Part 1.1-1.3) for complete prompt templates and representative examples illustrating this contextual communication process.

5.2 Code Integration

Once we receive the response, for individual modules, we extract the code, insert it into the element classes, and insert the context into the context information repository. When updating an element later, the newly generated code will replace the original one, and the newly updated context will be compared to the original ones and then updated. For the central module, we copy the generated code, replace the original, and insert the newly defined variables and functions into the relevant individual element class code. The context information is also compared and updated correspondingly. The insert positions are determined by the pre-defined or guided-maintained flags (e.g., "//variable start", "//variable end", "//function start", "//function end") in the code.

6 Evaluations Overview

To evaluate whether *MoGraphGPT* effectively addresses challenges and understand how users leverage our system, we formulated four questions to be answered in the evaluation:

RQ1: How does context-aware modularization enable independent element control and coordinated multi-element interactions?

RQ2: How does integrated graphical control improve user specification of spatial properties and reduce the friction of text-based spatial descriptions?

RQ3: How do automatically generated sliders enable precise parameter adjustment compared to iterative text-based refinement?

RQ4: How do the three features collectively support the end-to-end workflow and good performance of creating interactive scenes?

We conducted a comprehensive evaluation: (1) a comparative study examining system usability, controllability, and performance with technical evaluation, (2) an ablation study validating modularization's impact on interaction management and element independence, and (3) an open-ended study exploring system expressiveness and applicability across diverse creative scenarios.

7 Comparative Study

7.1 Baseline Setup

We designed a within-subject study to compare *MoGraphGPT* with a state-of-the-art tool, Cursor Composer [26], with the GPT-4o Mini model served as our baseline tool. It supports both full and partial code generation and refinement from text input, applicable to one or multiple files. However, it lacks graphical control features. To ensure a fair comparison, we prepared a basic code template identical to that of the *MoGraphGPT* system, featuring a central JavaScript file along with separate JavaScript files for individual elements. Participants could select files as context to enhance modular code refinement. To observe the results in real time, we launched a live server that rendered the outputs immediately. Participants were instructed to focus solely on text input, context selection, and result rendering, without visibility into underlying code.

7.2 Participants and Apparatus

We recruited 16 participants (7 females, 9 males; aged 22-34, M=28, SD=3.32) from our university network. They consisted of 5 students, 6 staff members, 4 designers, and 1 artist. They came from diverse backgrounds, including Computer Science and HCI (U2, U5-6, U12-15), Design and Arts (U4, U8, U11, U14, U16), and other fields such as Business and Science (U1, U3, U7, U9-10). While most were ordinary users, five participants (U5, U11, U13-14, U16) were experts who routinely create interactive scenes in their professional work. We assessed their experience on a 5-point scale (1=None to 5=Strong). Regarding coding experience, eight participants had little to no experience (Scores 1-2), two were intermediate (Score 3), and six had advanced skills (Scores 4-5). For ChatGPT proficiency, eight users rated themselves as beginners (Scores 1-2), two as intermediate (Score 3), and six as advanced (Scores 4-5). Their usage of ChatGPT primarily involved information search, writing assistance, and code generation. The study was conducted

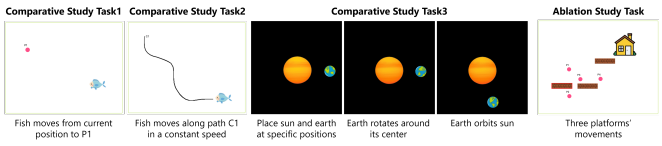


Fig. 6: Tasks in the comparative study and ablation study.

on a laptop running both systems, and participants could use a keyboard, touchpad, and mouse for input.

7.3 Tasks

We designed three tasks for users to reproduce the effects using *MoGraphGPT* and Cursor Composer. Please refer to (Figure 6-Left) for detailed effects. These three tasks are common in 2D interactive scenes and involve typical features such as the behavior of single elements and interactions between two elements, as well as spatial properties like positions, translations, rotations, paths, and speed. Users were required to create results similar to the given effect example video. In particular, the following features should be similar to the target effects as much as possible: (Task1) the positions of the starting point and the ending point in the canvas; (Task2) the moving path and the speed; (Task3) the positions of the sun and the earth, and the rotation speeds of the earth. To focus on the core task and lower the entry barrier, we provided all visual element images for participants in both systems during the study.

7.4 Procedure

We gave participants a 15-minute introduction to the tasks and two systems and allowed them to try the systems freely. Then, we showed them both an image and a video for the target effect of each task, and they could further see the image and video during the whole study process. To avoid the learning effects of our system, we asked each participant to first use *MoGraphGPT* and then Cursor Composer to reproduce the target effect for each task. Once the participants considered that their created target effect had been reproduced successfully, it was double-checked by two of our authors. If both of us reached a consensus, it was considered a complete result. He or she could move to the next step. If the participant tried over 10 minutes for similar text prompts but the system still did not provide a clear result, or the participant thought the effect was very difficult to achieve and he/she did not have any idea for it, it could move to the next step. After completing all the tasks, they were asked to fill in the questionnaire on a 5-point Likert scale. The questions are elaborated in Figure 7. We then conducted semi-structured interviews with them to collect their feedback, including their perspectives on the differences between *MoGraphGPT* and Cursor Composer, as well as our observations during the study. We recorded the time spent on each task, text prompts, operations, results, and generated codes. The whole process was audio-recorded and later transcribed with their agreement by filling out an informed consent form. Each participant received a 13-USD gift card for about one-hour participation.

TABLE 2: The performance of MoGraphGPT and Cursor across three metrics averaged over 16 participants. Note that the time, prompt number, and prompt length are averaged across participants for the total of the three tasks.

	Time (s)	Prompt N	Prompt L
Ours	350.50	4.56	24.69
Cursor	1359.69	16.44	240.50
Reduction	73.87%	68.48%	88.98 %

7.5 Data Analysis

The questionnaire includes personal information background questions, subjective ratings (Closeness to target, Graphical control, Precise refinement, Effect independency, Effect consistency, Effect clearness, Easy to specify action, Mental demand for formulating prompts), and selected questions from NASA-TLX (Figure 7). Following the measures of DirectGPT [11], we design our objective metrics consisting of time taken, the number of prompts, and prompt word counts to measure the system's effectiveness. We first ran the Shapiro-Wilk test to determine whether two sets of data follow a normal distribution ($p < 0.05$); otherwise, we used the Independent Samples t-test to measure whether there is a significant difference. If at least one group does not conform to a normal distribution, we performed an Aligned Rank Transform on both sets of data and then applied the Independent Samples t-test. To evaluate the quality of generated code, we employ an LLM-as-a-Judge approach for semantic assessment and a JavaScript compiler for executability evaluation.

7.6 Results

Completion. All participants successfully completed each task in our system within 6 minutes. However, when using Cursor, U1 (Task 1) and U2, U4, U11, and U15 (Task 2) failed to achieve satisfactory results even after exceeding 10 minutes of creation time. They found Cursor was hard to handle graphical information, such as specific positions and curved paths. Despite attempts to change descriptions or correct responses, Cursor often retained the original results or produced undesirable effects. Table 2 compares the performance of *MoGraphGPT* with Cursor across three metrics averaged over the 16 participants for the three tasks: total completion time (in seconds), prompt number, and prompt length (in words) for all the tasks. We also calculated the reduction rates for these metrics by averaging individual improvements of all the participants with *MoGraphGPT* compared to Cursor.

Time. We found the average time spent on each task using *MoGraphGPT* is significantly lower than Cursor: Task1 (normal distribution, $p < 0.001$): M: 58.81s, SD: 19.21s and M: 296.19s, SD: 148.17s for *MoGraphGPT* and Cursor, respectively; Task2 (not normal distribution, $p < 0.001$): M: 123.31s, SD: 74.22s for *MoGraphGPT* and M: 519.13s, SD: 168.35s for Cursor; Task3 (normal distribution, $p < 0.01$): M: 171.38s, SD: 81.55s for *MoGraphGPT* and M: 606.88s, SD: 264.65s for Cursor. Our graphical specification feature enables participants to quickly define positions, moving paths,

4. The time, prompt number, and prompt length are averaged across participants for the total of the three tasks.

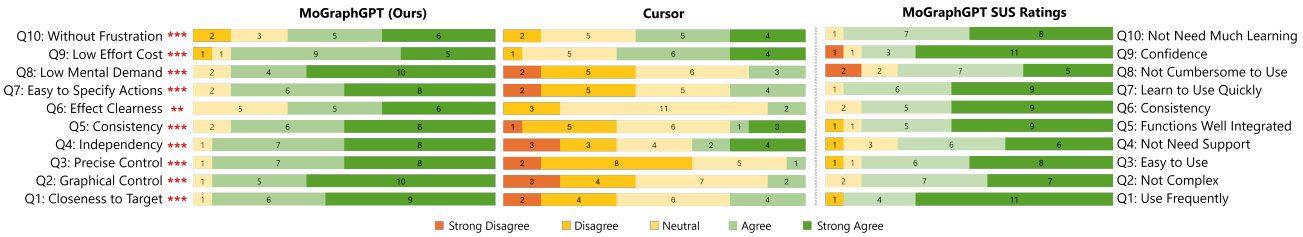


Fig. 7: Left: Subjective ratings on *MoGraphGPT* and Cursor Composer. "***" and "**" indicate a significant difference with $p < 0.001$ and $p < 0.01$ respectively. Right: SUS ratings on *MoGraphGPT* in the open-ended study; for consistent visualization, negatively worded items (even-numbered) were reverse-coded so higher scores uniformly indicate better usability (see supplementary materials for original questions and raw scores).

and both absolute and relative positions quickly (RQ2). Participants (U1, U4, U6-7, U9-10, U12-14, U16) were able to adjust motion parameters, such as speed, using sliders and numerical inputs with precision (RQ3). In contrast, participants using Cursor spent considerable time formulating prompts to integrate graphical information. Some (U1, U4-6, U8, U14, U16) struggled with precise descriptions for several minutes, especially when Cursor continued to produce undesired results. Participants (U1-3, U5, U7-11, U13-15) often had to try multiple word variations to adjust motion parameters. In Cursor, adding context does not guarantee independent control, often requiring participants to attempt multiple times and impose additional constraints (RQ1). In *MoGraphGPT*, individual and interaction behaviors can be created in the earth and central sessions in just one trial. This is because our modular structure provides a clear division when updating interactions–defining function code for elements within their individual classes and invoking these functions in the central module. In contrast, Cursor's context offers only soft constraints, causing interaction code and individual behavior code to intertwine, leading to inconsistent updates and chaotic modifications. **Prompt numbers and length.** Our system results in significantly fewer and shorter prompts, as confirmed by Independent Samples t-tests ($p < 0.01$). In Task1 and Task2, participants accurately referred to elements and graphical proxies (e.g., "P1", "C1") in their prompts, enabling them to quickly and directly map their spatial intentions to scene structures without excessive guesswork. Moreover, graphical control substantially reduced the friction and ambiguity found in text-based specifications, enabling users to express spatial properties with greater clarity and confidence (RQ2).

In contrast, participants must repeatedly enter requests, and LLMs often misunderstand the intent or spatial information in Cursor. They described the relative positions to the canvas (e.g., left-top, right-bottom, U1-3, U5, U7-9, U11-13), estimated specific coordinates (U4, U6, U10, U14-15), and refined results using reference objects and iterations (e.g., set it higher to the corner, make them much closer, U1-2, U4-6, U8-9, U11-13, U16). They adjusted coordinates through multiple iterations and articulated the curved motion path with terms like "curves with two circles" (U2), "waved curves" (U7, U11, U14), "tilde" (U1, U3, U9), and "two connected arcs" (U15). They further specified the shape with phrases such as "make it more curved" (U2, U10) and "let it curve to the left-bottom and then top-right" (U3).

In Task3, even for the second step, Cursor often failed to achieve the target effects on the first trial. Success typically came only after participants added the earth file as context and retried multiple times, which troubles the participants a lot. Some participants (U1-2, U5, U8-10, U12-14, U16) directly inputted text for the third step by instructing the earth to orbit the sun, resulting in the earth orbiting but losing its self-rotation effect. This necessitated rewriting the prompt to include this effect, such as "let the earth rotate around the sun while also rotating around itself." With *MoGraphGPT*, participants could input the self-rotation in the earth module and the orbiting effect in the central module independently (RQ3), requiring less prompting effort.

Subjective ratings and qualitative feedback. Participants overwhelmingly found our system more intuitive, easy to use, and effective than Cursor, preferring *MoGraphGPT* with Wilcoxon signed-rank tests validating the significance on all measures ($p < 0.01$, see (Figure 7). Participants (U2, U4-6, U7, U9, U12-14) highlighted the modularization of our system as a key advantage over Cursor (Q4 & Q5). For example, U4 noted, "the refinement effect for a single element [in our system] is clear (Q6), while in the other tool [Cursor], the modular refinement is ambiguous due to a lack of clear distinction between different elements." Participants with strong coding experience (U1-2, U5-6, U12, U15), including regular Cursor users, noted that Cursor often failed to produce the desired overall behavior and required repetitive corrective prompts. In contrast, they appreciated *MoGraphGPT*'s modular setup for its flexibility and controllable local updates (RQ1). It empowered them to refine individual elements independently and coordinate interactions efficiently in a more satisfying and manageable workflow (RQ4). The graphical UI allowed for rapid, intuitive manipulation of visual elements (Q2), surprising many users with its speed (Q7). Participants who tried redrawing curves in Task 2 (U2, U8-9) still found the process easy and welcomed experimentation. Some participants (U1-2, U7, U13-15) highlighted difficulties with Cursor relying on text for shape creation, resulting in frustration and ambiguity (RQ2). The precise control over effect parameters (Q3) in *MoGraphGPT* was praised by participants. They highlighted its intuitiveness and effectiveness, even for the interaction effects with multiple elements (e.g., orbit radius, orbit speed). In contrast, they found that using text to adjust

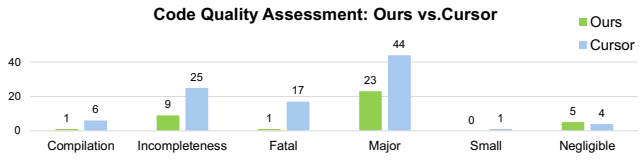


Fig. 8: Code quality assessment comparing *MoGraphGPT* and Cursor. The error counts of different levels are aggregated across all participants and tasks. Lower values indicate higher code quality.

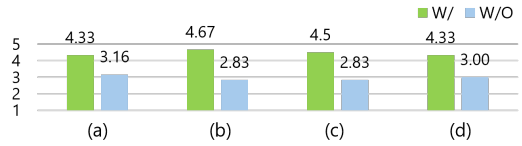


Fig. 9: Subjective ratings between with and without modularization on (a) Result Completeness, (b) Close to Target,(c) Refinement Not Affect Previous Steps, (d) Refinement Not Affect other Elements. The higher the score, the better.

parameters in Cursor "does not have a reference" (U7) and required balancing between excessive and inadequate control (U2, U9, U12).

Code Quality Evaluation. Following CodeJudge [48], we adopt the LLM-as-a-Judge paradigm, leveraging large language models' reasoning capabilities to evaluate generated code quality. We employ Claude 4.5 Sonnet as the evaluator, given its superior code comprehension and reasoning performance. Similar to CodeJudge [48], we design two prompt categories to assess codes generated by *MoGraphGPT* and Cursor: (1) a summary prompt evaluating functional incompleteness, and (2) a taxonomy prompt providing fine-grained assessments across four error severity levels: Fatal, Major, Minor, and Negligible. To ensure executability beyond semantic correctness, we evaluate compilation errors by simulating a browser environment using Selenium and a standard JavaScript compiler. As shown in Figure 8, *MoGraphGPT* produces significantly fewer errors than Cursor across all critical severity levels (Compilation to Major errors) that would prevent successful execution. For minor errors (Small and Negligible), both systems perform comparably. These results demonstrate that our context-aware LLM modularization effectively enhances code quality and reliability. For detailed prompts and implementation, refer to the supplementary material (Part 6).

8 Ablation Study

To assess how modularization affects both independent element management and coordinated multi-element interactions (RQ1), we conducted an ablation study.

8.1 Setup and Participants

We prepared two system versions: with modularization (our original system, w/ system) and without (w/o system). In w/o system, we keep all other functions but remove the "Multiple" button. When users input text and graphical information, selecting any element and generating the output produces no distinction among elements. On the backend, the system processes and combines all user inputs describing both elements and their interactions into a single file for code generation. Six participants (U11-16), also part of the comparative study, participated in this study.

8.2 Task and Procedure

We designed a reproduction task to ask participants to create the interactive scene using two system versions. The target scene (Figure 6-Right) included: (1) a vertical moving platform, (2) a horizontal moving platform, (3) a platform rotating around a house, and (4) a shaking effect on the vertical platform, covering individual behaviors, inter-element interactions, and single-element refinement. We provided all visual element images in both systems. Participants first watched a demo video of the scene and recreated it as accurately as possible. Version order was counterbalanced. Once participants completed their creation of the target effect, two authors independently verified it. A result was considered complete only when both reached consensus. After using both versions, each participant completed a 5-point Likert questionnaire on Result Completeness, Closeness to Target, Refinement Not Affect Previous Steps, and Refinement Not Affect Other Elements.

8.3 Data Analysis and Results

For each participant and system version, we recorded the creation time, the number of prompts, the prompt word counts, and a video of the creation process and the final result. All participants successfully completed the task and were satisfied with their results using w/ system with 6 minutes (mean creation time: 233.83s, SD: 76.83s). Figure 9 shows the subjective ratings on the different dimensions. For w/o system, three participants (U12-15) reported that they did not achieve the desired target effect even after more than 12 minutes of work (mean creation time: 750.00s, SD: 142.92s). This significant increase in creation time for w/o system was mainly due to the participants spending a lot of time for refinement, and it took a longer time for each generation and refinement. W/ system enables individual generation and refinement on each element, which did not affect others or require global checks and updates. Even for interactions among elements, only the relevant elements needed to be checked and updated, based on the current context. By contrast, w/o system generated the entire scene as a single file, so every refinement triggered global checks and updates, greatly increasing the workload.

On average, participants used 4.5 prompts (SD: 1.05) with w/ system, compared to 7.83 prompts (SD: 1.94) with w/o system. The average prompt length was 39.5 words (SD: 14.10) for w/ system and 80.93 words (SD: 17.83) for w/o system. With modularization, each participant achieved a clear and effective outcome for each input. Refining a single element (i.e., adding shaking effect to a platform) produced cumulative results on the element's behavior as intended, without unintended side effects on the overall scene (RQ1). Participants also used sliders to control fine-grained movement and rotation speeds. However,

when participants (U10, U13-14) added a shaking effect to one platform, the same effect propagated to other platforms as well. Similarly, specifying a rotation effect for a single platform sometimes unintentionally triggered rotations in others (U11, U13-14, U16) (RQ1). Participants spent considerable time recovering their previous effects when such interference occurred, and the process became confusing after multiple iterations. Later, even when they wanted to remove an effect, it also affected other effects (U10, U12, U15).

9 Open-Ended Study

To further evaluate the usability and expressiveness of *MoGraphGPT*, we invited participants to an open-ended study in which they could freely create their own desired 2D interactive scenes. We recruited 12 participants (aged 24-33, M: 28.75, SD: 2.70, 5 females and 7 males, P1-12), and all participated in our comparative study.

Procedure. Before the study, the participants were asked to think about interactive scenes that they would like to create. This process mainly allowed us to prepare the images for those elements requiring image uploading from our devices. During their creation using *MoGraphGPT*, we stayed next to them, answering questions and providing verbal guidance whenever they had doubts. After they finished the creation, they played or showed the created results for us to demonstrate the final scenes.

Results. The participants created 18 results in total. Figure 10 shows parts of result snapshots, including 6 games (1 two-player game (Figure 10(a)) and 5 single-player games (Figure 10(b)-(e), (j)), 1 interactive animation demo (Figure 10(f)), 1 website ad interaction design demo (Figure 10(g)), 1 dynamic illustration for an academic paper (Figure 10(h)), and 1 interactive artwork (Figure 10(i)). Each result contains 4-10 elements and various types of single-element behavior and interactions among multiple elements. Each result was completed in 10-30 minutes, including the creation and testing time. They included different user interactions, such as following the mouse, using the arrow keys to control moving directions, using other keys to control moving speed, and using the mouse click to trigger dynamic effects. Multiple graphical controls were used, including user-drawn points for specifying target positions, lines for defining curves for moving paths, and regions for active effects. For example, the participants were able to define specific points and draw curves, simulating the movement of the sun or a bird along a designated path (Figure 10(f)(j)). They could also create defined areas, such as a region within a tree hollow where squirrels could move freely, with nuts appearing randomly in that region (Figure 10(b)), enhancing the scene's liveliness. Additionally, users set up clickable regions (Figure 10(g)(j)), such as an orange that displayed an image of the fruit when clicked, fostering engagement and interaction. Expert participants (P3, P7, P9-10, P12) pointed out that, compared to their daily used tools (e.g., Unity, Unreal, Construct), our tool helped them quickly build interactive scenes without coding. It significantly enhanced creative freedom (P7, P12), allowing them to focus more on scene design and interaction without being constrained by technical implementation (P9, P12). While their commonly used platforms offer strong scripting and plugin ecosystems, changes to one aspect may require attention to dependencies or scripts elsewhere (P3, P9, P12), making independent element control more complex. They liked our accessible workflow, where the modular framework ensures structure and consistency (P3), while precise control via natural language or graphical sliders delivers both ease of use and professional-grade accuracy (P7, P12). They especially thought this could streamline collaboration, particularly in team discussions and interdisciplinary projects (P7, P10), helping to reduce the time spent on prototyping and revising interactive scenes. Many of the results produced by participants are readily applicable in their professional practice, demonstrating that our system supports not only basic tasks but also advanced, realistic creation workflows for experienced users.

The SUS score rated by the participants is 83 on a 100-point scale, indicating good usability. The distribution of the SUS score for each question is shown in Figure 7-Right. We observed that both individuals with programming backgrounds and those without found it easy to use our system to create their desired outcomes. The participants without programming experience (P2, P5-6, P11) expressed a strong need for a tool that does not require coding, since the cost of learning programming is quite high. Our system offers an intuitive solution, enabling them to design animations or games of varying complexity for use in their learning and daily life (P2, P6). These users particularly noted that, beyond entertainment, our tool could also be employed to create diagrams and dynamic effects for academic papers (Figure 10 (h), P3, P6). Participants with programming experience found the system highly useful (P3, P8). P3, with 8 years of programming experience, noted that traditional methods would take at least two hours, but with our system, she completed a game in 15 minutes. This saved significant time and eliminated the need to learn new languages or frameworks, as she could directly generate game code through natural language and graphic controls. Participants (P1, P4) mentioned that the system required no close technical guidance; a simple introduction was sufficient to get started quickly. P1 particularly valued the "modular architecture", which enables users to navigate complex object relationships more effectively, stating: "even for programming simple games, this system is much easier to use than ChatGPT."

However, participants also highlighted some limitations, including limited graphic controls (P5-6) restricted to points, straight lines, curves, and user-drawn regions. They desired additional shapes like circles, rectangles, and triangles, plus automatic background image segmentation. Some participants (P11) expressed concern that the strictly modular framework might constrain broader applications, particularly for specialized or cross-cutting interactions spanning multiple modules. This echoes observations that overly rigid modular architectures can hinder system extension to new use cases.

10 Discussions and Future Work

10.1 Modularization with LLMs: Design and Trade-offs

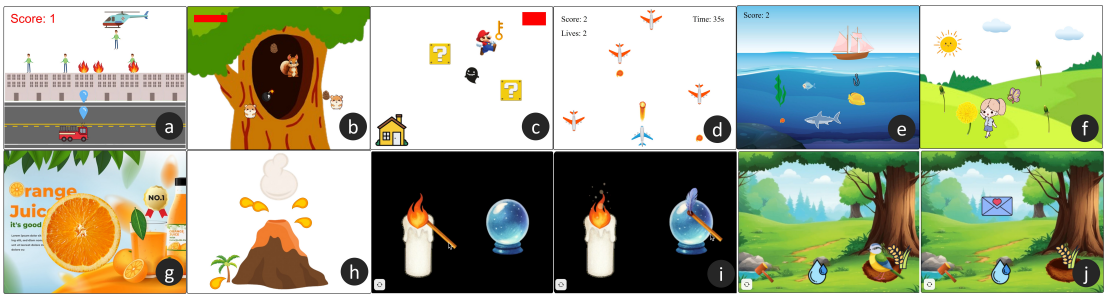


Fig. 10: A gallery of selected results in the open-ended study. (a) Two-player Rescue Game (P2). (b) Squirrel Guard Game (P1). (c) Scavenger Hunt Game (P4). (d) Airplane War Game (P4). (e) Sea Fishing Game (P6). (f) Interactive Animation Demo (P3). (g) Website Ad Design Demo (P5). (h) Dynamic Illustration for Academic Paper (P6). (i) Interactive Artwork (P7). (j) Interactive Storytelling Game (P11). Please refer to supplementary materials and video for detailed descriptions.

We tackled the challenge of interdependent effects on multiple elements within conversational LLMs through our element-level modularization technique. While our primary focus is on creating 2D interactive scenes, this issue is prevalent across many other LLM applications. Our approach presents a valuable solution applicable to broader LLM-based frameworks (e.g., Google Gemini, Claude, LLaMA), extending beyond 2D visual creation to encompass a broader range of interactive scenarios. Our modularization technique encourages users to provide descriptions for individual elements separately. While this promotes clarity and specificity, it can hinder seamless interaction to some extent, as users must manually delineate each element's behaviors, in contrast to the original ChatGPT interface, where elements can be combined more fluidly, albeit often with undesired outcomes. Looking ahead, we plan to develop an advanced input parsing method that allows users to describe elements without strict separation. Our system will intelligently partition descriptions of individual and multiple elements, routing them to distinct modules as necessary. This aims to balance user control with interaction fluidity, improving the overall user experience.

10.2 Positioning Against Professional Engines

Several expert study participants compared *MoGraphGPT* with professional engines such as Unity and Unreal, which commonly adopt an Inspector/Component paradigm. These engines provide a bottom-up workflow that enables fine-grained control but introduces nontrivial architectural overhead when managing interactions among many elements, particularly in early prototyping. Our context-aware modularization follows a different top-down workflow. Instead of manually wiring components, users describe desired behaviors and interactions at the level of semantic intent, while individual LLM modules maintain element-level classes and the central module orchestrates interactions using the shared context. This automatically separates individual behaviors from interaction logic and localizes updates, reducing the chance that refining one element unintentionally affects others. From a professional usage perspective, *MoGraphGPT* is thus positioned as a complementary, no-code front end for rapid interaction sketching rather than a replacement for professional engines, and the scenes produced by expert participants suggest potential as semantically structured prototypes that could be exported into standard Inspector/Component workflows for further refinement. In the future, we will validate these differences by conducting comparative studies between *MoGraphGPT* and professional tools and by evaluating the resulting scenes using our LLM-as-a-judge protocol.

10.3 Limitation of Context-awareness

Our modular system significantly reduces context burden compared to the non-modular baseline (2K-5K vs. 20K-40K tokens). Although the employed LLM (GPT-4o Mini) supports a 128K context, observed consumption ranges from 0.5K to 10K tokens per turn, depending on task complexity. This suggests that *MoGraphGPT* is optimal for projects with up to 30 elements within 5 chat turns. Beyond this scale, accumulated context may trigger model limits, potentially causing truncated responses or performance degradation. To address these challenges, future work should develop context management strategies that summarize key information and discard irrelevant details. Dynamic context windowing can allocate tokens to important exchanges, while chunking complex tasks helps preserve context in long interactions. Enhancing context-awareness in these ways aims to improve LLM robustness and coherence in complex scenarios.

10.4 Consolidation and Conflict Resolution

Our current consolidation module mainly maintains consistency between the central control code and the element modules, rather than introducing new behaviors. In our study scenes, the generated central control code remained relatively small (a handful of interaction handlers and cross-element calls), so consolidation operated on modestly sized code bases (up to 30 interaction elements due to context limitations of the consolidation module as discussed in Section 10.3) and could align references and interfaces without adding extra structural complexity. Typical conflicts involved duplicated variable names, mismatched function signatures after refinement, or interaction logic referring to outdated element contexts, which the module resolved by renaming, updating calls, and synchronizing with the latest summaries.

Across our comparative and ablation studies, consolidation successfully produced executable scenes for most

generation and refinement steps, and participants rarely reported blocking failures attributable to this process. However, for highly coupled behaviors that implicitly change several interaction patterns at once, consolidation can still introduce redundant or unused code, which users currently resolve by issuing additional high-level prompts rather than directly editing the code. In future work, we plan to instrument the system with code-level metrics and explicit logging of consolidation outcomes, so that we can report more systematic statistics on conflict resolution success and expose targeted code-editing options when automatic consolidation is insufficient.

10.5 Code Visibility and Parameter Control

Currently, we do not display the generated code in our interface. However, we plan to gradually introduce code features as users become more comfortable with the basics. This will help them understand the underlying logic of their creations and provide a pathway to more advanced programming concepts. In future iterations, we aim to provide mechanisms that help users identify and resolve discrepancies between intended and actual outcomes. For example, users will be able to view, edit, and refine auto-generated code directly, in addition to the core prompt-based iteration. Highlighted code sections, contextual explanations, and interactive feedback will further support users in debugging their scenes and manually correcting logic. For effect control, we instruct the LLM to expose behavioral and interaction parameters as editable variables. However, the quality of parameter exposure depends on how the LLM interprets the prompt and structures the code. For example, if the underlying code does not define the curve of a path as a variable (e.g., directly hard-coding Bezier points rather than using named parameters), the corresponding slider may not be generated. We plan to enhance this by introducing guided prompt templates and validating code structure for more consistent UI controls.

10.6 Scalability for Complex Scenarios

As more elements are added, behaviors and interactions can become complex. To support this, we can enhance our system by integrating a graph representation [30], where nodes represent elements and edges represent their relationships, with additional lines indicating event logic. This visualizes both element behaviors and interactions as attributed objects [49] after code generation. Users can then reuse behaviors by dragging objects to other elements, or directly reconnect or delete elements and edges for quick management. Currently, *MoGraphGPT* supports multiple scenes (e.g., Figure 10(d)) by switching in the central module to manage element visibility. For more complex applications, we might add a scene management level that opens modules for scenes. Users can manually create scenes and add elements, or the system can extract them from text input. We can visualize scenes as workflow UIs connected by key condition variables, allowing users to manipulate scene order and relationships directly, opening opportunities for exploring conditional layered and nested LLMs.

10.7 Reflections on Formative Steps

In our formative step, we used content analysis of tutorial videos to identify challenges for novice users. While capturing practical difficulties, this approach has limitations. Tutorial videos reflect creators' practical experiences and problem-solving processes but may not represent all users or rare edge cases. Tutorials often showcase standard workflows (e.g., basic animations) but may omit complex scenarios like coordinating multiple animations with intricate timing or handling conflicts between overlapping interactive elements. Such advanced cases may pose barriers for users with limited programming or design experience. We acknowledge these limitations and will include expert interviews or direct observations in future work to uncover less visible learning barriers.

11 Conclusion

This paper presents *MoGraphGPT*, an LLM-based system for creating 2D interactive scenes from text and graphical input without coding. We identified three key issues: lack of independent element generation, insufficient graphical control, and imprecise refinement. We propose context-aware modularization with individual LLM modules for elements and a central module coordinating interactions. Our graphical interface integrates modular LLMs and advanced controls for seamless code generation. Comparative studies show *MoGraphGPT* reduces time, prompt trials, and prompt length versus Cursor Composer with better control. Ablation and usability studies confirm that users can create diverse scenes easily without coding.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and the user study participants for their time. This work was partially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (HKUST PDFS2223-1S10), the National Natural Science Foundation of China (62472287), and the Natural Science Foundation of Shenzhen City (JCYJ20250604181519025).

References

[1] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko, "How designers design and program interactive behaviors," in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2008, pp. 177–184.

[2] L. Zhang and S. Oney, "Flowmatic: An immersive authoring tool for creating interactive scenes in virtual reality," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 342–353.

[3] F. De La Torre, C. M. Fang, H. Huang, A. Banburski-Fahey, J. Amores Fernandez, and J. Lanier, "Llmr: Real-time prompting of interactive worlds using large language models," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–22.

[4] C. Lan, Y. Wang, C. Wang, S. Song, and Z. Gong, "Application of chatgpt-based digital human in animation creation," *Future Internet*, vol. 15, no. 9, p. 300, 2023.

[5] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.

[6] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant–how far is it?" *arXiv preprint arXiv:2304.11938*, 2023.

[7] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[8] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[9] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[10] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–21.

[11] D. Masson, S. Malacria, G. Casiez, and D. Vogel, "Directgpt: A direct manipulation interface to interact with large language models," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.

[12] H. Dang, L. Mecke, F. Lehmann, S. Goller, and D. Buschek, "How to prompt? opportunities and challenges of zero-and few-shot learning for human-ai interaction in creative applications of generative models," *arXiv preprint arXiv:2209.01390*, 2022.

[13] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.

[14] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proceedings of the 2022 CHI conference on human factors in computing systems*, 2022, pp. 1–22.

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[16] N. Ritschel, F. Fronchetti, R. Holmes, R. Garcia, and D. C. Shepherd, "Can guided decomposition help end-users write larger block-based programs? a mobile robot experiment," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 233–258, 2022.

[17] T. Wu, E. Jiang, A. Donsbach, J. Gray, A. Molina, M. Terry, and C. J. Cai, "Promptchainer: Chaining large language model prompts through visual programming," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–10.

[18] R. Yen, J. Zhu, S. Suh, H. Xia, and J. Zhao, "Coladder: Supporting programmers with hierarchical code generation in multi-level abstraction," *arXiv preprint arXiv:2310.08699*, 2023.

[19] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk *et al.*, "Graph of thoughts: Solving elaborate problems with large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, 2024, pp. 17 682–17 690.

[20] H. Le, H. Chen, A. Saha, A. Gokul, D. Sahoo, and S. Joty, "Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules," *arXiv preprint arXiv:2310.08992*, 2023.

[21] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong *et al.*, "Chatdev: Communicative agents for software development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 15 174–15 186.

[22] Y. Zhao, J. Wang, L. Xiang, X. Zhang, Z. Guo, C. Turkay, Y. Zhang, and S. Chen, "Lightva: Lightweight visual analytics with llm agent-based task planning and execution," *IEEE Transactions on Visualization and Computer Graphics*, 2024.

[23] T. S. Kim, Y. Lee, M. Chang, and J. Kim, "Cells, generators, and lenses: Design framework for object-oriented interaction with large language models," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–18.

[24] K. T. Rosenberg, R. H. Kazi, L.-Y. Wei, H. Xia, and K. Perlin, "Drawtalking: Building interactive worlds by sketching and speaking," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–25.

[25] T. Angert, M. Suzara, J. Han, C. Pondoc, and H. Subramonyam, "Spellburst: A node-based interface for exploratory creative coding with natural language prompts," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–22.

[26] Cursor, "Cursor - the ai code editor," 2023, accessed: 2024-11-30. [Online]. Available: https://www.cursor.com/

[27] S. Brade, B. Wang, M. Sousa, S. Oore, and T. Grossman, "Promptify: Text-to-image generation through interactive prompt exploration with large language models," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–14.

[28] Y. Cui, W. G. Lily, Y. Ding, L. Harrison, F. Yang, and M. Kay, "Promises and pitfalls: Using large language models to generate visualization items," *IEEE Transactions on Visualization and Computer Graphics*, 2024.

[29] S. Xiao, L. Wang, X. Ma, and W. Zeng, "Typedance: Creating semantic typographic logos from image through personalized generation," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–18.

[30] Z. Yan, C. Yang, Q. Liang, and X. Chen, "Xcreation: A graph-based crossmodal generative creativity support tool," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–15.

[31] L. Shen, Y. Zhang, H. Zhang, and Y. Wang, "Data player: Automatic generation of data videos with narration-animation interplay," *IEEE Transactions on Visualization and Computer Graphics*, 2023.

[32] T. Tseng, R. Cheng, and J. Nichols, "Keyframer: Empowering animation design using large language models," *arXiv preprint arXiv:2402.06071*, 2024.

[33] Vercel, "v0 by vercel," 2024, accessed: 2024-12-10. [Online]. Available: https://v0.dev/

[34] C. Hu, Y. Zhao, and J. Liu, "Generating games via llms: An investigation with video game description language," *arXiv preprint arXiv:2404.08706*, 2024.

[35] L. Chen, S. Xiao, Y. Chen, Y. Song, R. Wu, and L. Sun, "Chatscratch: An ai-augmented system toward autonomous visual programming learning for children aged 6-12," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–19.

[36] Q. Chen, S. Cao, J. Wang, and N. Cao, "How does automation shape the process of narrative visualization: A survey of tools," *IEEE Transactions on Visualization and Computer Graphics*, 2023.

[37] J. Liu, H. Fu, and C.-L. Tai, "Posetween: Pose-driven tween animation," in *Proceedings of the 33rd annual acm symposium on user interface software and technology*, 2020, pp. 791–804.

[38] R. H. Kazi, F. Chevalier, T. Grossman, S. Zhao, and G. Fitzmaurice, "Draco: bringing life to illustrations with kinetic textures," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 351–360.

[39] J. Xing, R. H. Kazi, T. Grossman, L.-Y. Wei, J. Stam, and G. Fitzmaurice, "Energy-brushes: Interactive tools for illustrating stylized elemental dynamics," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 2016, pp. 755–766.

[40] R. H. Kazi, F. Chevalier, T. Grossman, and G. Fitzmaurice, "Kitty: sketching dynamic and interactive illustrations," in *Proceedings of the 27th annual ACM symposium on User interface software and technology*, 2014, pp. 395–405.

[41] H. Ye, J. Leng, P. Xu, K. Singh, and H. Fu, "Prointerar: A visual programming platform for creating immersive ar interactions," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–15.

[42] MIT Media Lab, "Scratch," 2003, accessed: 2025-02-08. [Online]. Available: https://scratch.mit.edu

[43] P. Jiang, J. Rayan, S. P. Dow, and H. Xia, "Graphologue: Exploring large language model responses with interactive diagrams," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–20.

[44] C. Wu, S. Yin, W. Qi, X. Wang, Z. Tang, and N. Duan, "Visual chatgpt: Talking, drawing and editing with visual foundation models," *arXiv preprint arXiv:2303.04671*, 2023.

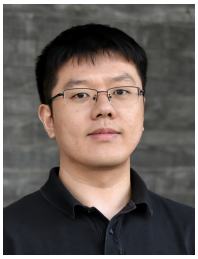
[45] I. Arawjo, P. Vaithilingam, M. Wattenberg, and E. Glassman, "Chainforge: An open-source visual programming environment for prompt engineering," in *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–3.

[46] K. Charmaz, "Constructionism and the grounded theory method," *Handbook of constructionist research*, vol. 1, no. 1, pp. 397–412, 2008.
[47] R. Cheng, T. Barik, A. Leung, F. Hohman, and J. Nichols, "Biscuit: Scaffolding llm-generated code with ephemeral uis in computational notebooks," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 13–23.
[48] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 20 032–20 051.
[49] H. Xia, B. Araujo, T. Grossman, and D. Wigdor, "Object-oriented drawing," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 4610–4621.

Hui Ye is an Assistant Professor at the Department of Interactive Media, School of Communication, Hong Kong Baptist University. She was previously an RGC Postdoctoral Fellow at the Division of Arts and Machine Creativity of HKUST and the School of Creative Media of CityUHK. She obtained her PhD degree from CityUHK and a Bachelor's degree from USTC. Her research interests lie in Human-Computer Interaction and Computer Graphics. She published papers in ACM SIGCHI, ACM TOG, IEEE TVCG, etc.

Chufeng Xiao is a Post-doctoral Fellow at Hong Kong University of Science and Technology. He obtained his Ph.D. degree from the School of Creative Media, City University of Hong Kong, and his B.E. degree from the School of Computer Science and Engineering, South China University of Technology. His research interests include Computer Graphics and Human-Computer Interaction, with a special focus on controllable AIGC.

Jiaye Leng is a PhD candidate at the School of Creative Media, City University of Hong Kong. Before joining CityUHK, he received a Master's Degree in Computer Technology from Beihang University and a Bachelor's Degree in Electronic Information Science and Technology from China University of Mining and Technology. His research interests are Human-AI Interaction and Creativity Support.



Pengfei Xu is an Associate Professor in the College of Computer Science and Software Engineering at Shenzhen University. He received his Bachelor's degree in Math from Zhejiang University, China, in 2009 and his Ph.D. in Computer Science from the Hong Kong University of Science and Technology in 2015. His primary research lies in Human-Computer Interaction and Computer Graphics.

Hongbo Fu is a Professor at the Division of Arts and Machine Creativity, Hong Kong University of Science and Technology. Before joining HKUST, he worked at the School of Creative Media, City University of Hong Kong, for over 15 years. He received a Ph.D. degree in Computer Science from HKUST in 2007 and a BS degree in information sciences from Peking University, China, in 2002. His primary research interests fall in Computer Graphics, Human-Computer Interaction, and Computer Vision. He has served as an associate editor for The Visual Computer, Computers&Graphics, and Computer Graphics Forum.