

# MoGraphGPT: Creating Interactive Scenes Using Modular LLM and Graphical Control

Hui Ye

Hong Kong University of Science and Technology  
Hong Kong, China  
huiyehy@outlook.com

Chufeng Xiao

Hong Kong University of Science and Technology  
Hong Kong, China  
chufengxiao@outlook.com

Jiaye Leng

City University of Hong Kong  
Hong Kong, China  
jiayeleng2-c@my.cityu.edu.hk

Pengfei Xu

Shenzhen University  
Shenzhen, Guangdong, China  
xupengfei.cg@gmail.com

Hongbo Fu\*

Hong Kong University of Science and Technology  
Hong Kong, China  
hongbofu@ust.hk

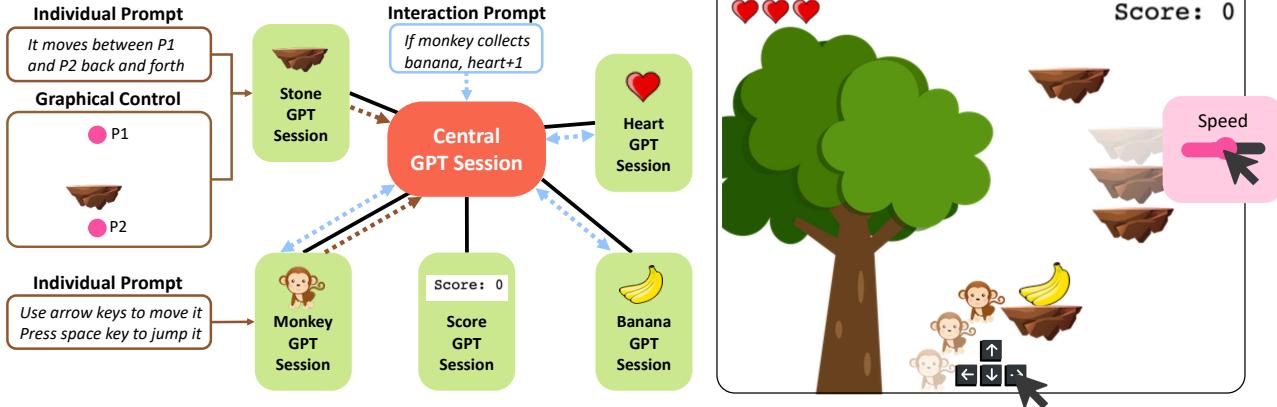


Figure 1: We introduce *MoGraphGPT* to facilitate the easy creation of interactive scenes using a graphical user interface powered with modular LLMs. Users can input the text descriptions for individual element properties or behaviors, or interactions among multiple elements, along with the directly specified or drawn graphical information into our system. The code for each element and interactions among elements is generated from different modules, with automatically-generated sliders to control effects precisely.

## 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

\*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY'

© 2025 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

of individual elements, and lacking graphical and precise control complicates visual integration. To address these issues, we integrate an *element-level modularization* technique that processes textual descriptions for individual elements through separate LLM modules, with a central module managing interactions among elements. This modular approach allows for refining each element independently. 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. Our comparative evaluation against an AI coding tool, Cursor Composer, as the baseline system and a usability study show *MoGraphGPT* significantly improves easiness, controllability, and refinement in creating complex 2D interactive scenes with multiple visual elements in a coding-free manner.

## CCS CONCEPTS

- Human-centered computing → Graphical user interfaces; Natural language interfaces; User interface toolkits; Interaction techniques; Interactive systems and tools.

## KEYWORDS

Code Generation, Modularization, Large Language Models, ChatGPT, Graphical Control, Interactive Scenes

### ACM Reference Format:

Hui Ye, Chufeng Xiao, Jiaye Leng, Pengfei Xu, and Hongbo Fu. 2025. *MoGraphGPT: Creating Interactive Scenes Using Modular LLM and Graphical Control*. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Interactive scenes combine artistic visuals with interactive elements, allowing users to immerse themselves in a richly crafted environment. They are widely used in various scenarios, including video games [42, 44, 64], educational tools [27, 39], and interactive demonstrations [17, 59]. Creating interactive scenes often involves coding with engines like Unity [58], or frameworks like Phaser [45], requiring skills in scripting interactivity, managing animations, and debugging. This complexity is crucial for delivering engaging user experiences but can be difficult for beginners [25, 40, 54, 72, 78].

Recently, Large Language Models (LLMs) like ChatGPT [43] have emerged as powerful tools for users, enabling them to generate code given natural text as input. They can assist users in writing scripts [16], creating simple animations [29], and even debugging [79], significantly reducing the time and effort required for coding tasks [55]. Some ad-hoc tools for code generation and refinement [6, 13, 18, 19, 46] 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 may produce incomplete or incorrect code sometimes [32, 41, 50]. 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 [77]. 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 modification. 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 [38]. 4) **lack of precise control** – fine modifications on the generated effects (e.g., effect parameters) require users to adjust text prompts iteratively [15], which usually traverse users between excessive and inadequate controlling.

By reviewing videos on using LLMs to create interactive scenes and comparing existing tools on code generation using content analysis [20], we identified the challenges of applying existing tools for generating codes for 2D dynamic and interactive scenes. Based on the insights, we design and develop *MoGraphGPT*, an interactive system for creating interactive scenes using modular LLM and graphical control without coding. To enable independent code generation for individual scene elements, we integrate an *element-level modularization* technique, which maintains independent LLM modules for individual elements, where users can input text descriptions to generate class codes for the element features and actions. On top of individual modules, a central LLM module manages the relationships and interactions of all the elements. To avoid the central module's lack of context-specific details, we guide LLM to distill contextual information about the variables and functions along with the generated code for individual elements. So the central LLM can generate correct and desired interaction code based on the contextual information. We implement this concept in the *MoGraphGPT* system, a graphical interface for users to create 2D interactive scenes using modular LLM and graphical control. In *MoGraphGPT*, users draw/import their prepared elements or ask the system to generate elements and then input the text for each element and multiple-element interaction separately. Four types of graphical proxy, including point, line, curve, and region, can be specified by direct pointing and drawing, and then be explicitly mentioned in the text prompt. The positions, orientations, and sizes of elements can be adjusted manually and integrated into the generated code automatically. Our system automatically generates sliders from the code for users to interactively adjust the parameters of the effects in the code, to reduce the text description input. A comparative study shows that *MoGraphGPT* outperforms a state-of-the-art AI coding tool, Cursor Composer in creating interactive scenes for fixed tasks. An open-ended study demonstrates *MoGraphGPT* enables users to create diverse 2D games, animations, and demonstrations easily.

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 *element-level 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.
- Two evaluations that compare the performance of *MoGraphGPT* and Cursor Composer and validate the system usability.

## 2 RELATED WORK

### 2.1 LLM-based Code Generation and Improvement

LLM-based code generation leverages advanced language models [32, 41, 50] 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, need refactoring, security smells, etc [36, 37, 51, 55]. In particular, ChatGPT struggles to generate code for new and unseen problems [55]. Lengthy prompts might have negative impacts on the code generation [36, 55]. 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 [36].

Researchers have proposed to use Chain-of-Thought [63, 67] to make LLMs more controllable for complex tasks, especially for text-based organization tasks. For coding tasks, many decomposition strategies are employed using interactive decomposition [23], block-based hierarchical structure [48], node-based diagrams [66]. CoLadder [75] further enables programmers to decompose tasks flexibly. These works mainly focus on general programming tasks, which typically involve static code generation and logic implementation. Modularized LLM generation mechanisms have been researched by Tree-of-Thoughts and its variations [7, 73]. They focus on general thought exploration, and we aim for a specific scenario of interactive scene creation. We integrate textual and graphical inputs with LLM code generation to enable intuitive control over element behavior and interaction. Agentic workflows [47] achieve agent-level modularization and communication for entire software development across roles (Table 1). In contrast, we emphasize element-level modular creation. We focus on precise and independent control over code generation for individual elements and their interaction. Kim et al. [28] explore a design framework for users to control the modularized generation of configuration components for writing tasks. We extend this framework to a specific application of interactive scene creation, where elements act as objects. This shift presents unique challenges in managing element interactions. We address this by encapsulating the code from the interaction generation or refinement for each element within its module while invoking the interaction logic in a central module.

### 2.2 LLM-powered Tools for Visual Design and Development

LLMs have been employed to enhance visual design [9] and development processes and generate 2D static visual designs, such as personalized logos [69], interior color designs [21], storybook [71], and editorial illustrations [35]. Designing dynamic 2D visuals with LLMs, however, presents greater challenges. Unlike static designs, animations require consideration of timing, motion, and interaction. Keyframer [57] leverages LLMs to empower users in creating animations by generating keyframes based on textual descriptions. LogoMotion [34] develops an LLM-based system that automatically generates content-aware animations for logos by synthesizing code from visual layouts. Spellburst [3] introduces a node-based

interface that enables users to explore creative coding through natural language prompts for interactive visual design with precise parameter control (Table 1). While these works enhance the use of LLMs in animation and dynamic visual design, they do not address interactions between multiple elements, thus overlooking the challenges of independent control. v0 [61] can generate interactive and dynamic effects, but lacks clear control for individual elements and graphical information.

Games are one of the most popular applications of interactive scenes. Researchers have explored the potential of LLMs in game content design, including investigating the use of video game description language [22], automated level design and generation from text [53, 56], and co-creative game design [4]. 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 [47], assisting end-users in generating robot programs [8], enabling zero-code generation of trigger-action IoT programs [31], and providing an AI-augmented system for autonomous visual programming learning for children [11]. They aim to lower the barriers to programming, making it easier for non-experts to create and manage complex applications. 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, previous works introduce sketch-based interactions to animate virtual elements [33], manipulate kinetic textures [26], and using filters for dynamic illustrations [70]. Besides dynamic effects, Kitty [25] 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 [74], node-graphs [52, 60], and flowcharts [12, 76, 78] to build interactive scenes. For example, as a pioneering platform, Scratch [39] enables users to create animations and games through a block-based coding environment. Our graphical interface is largely inspired by the interface of Scratch. 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 [49] 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

**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	Element-level	Yes	Yes
DrawTalking [49]	Interactive scene creation	Rule-based	Yes	/	Yes	No
Spellburst [3]	Generative art creation	LLM-based	No	Node-level	No	Yes
DirectGPT [38]	Text/code/vector images edition	LLM-based	No	No	Yes	No
ChatDev [47]	General software development	LLM-based	Yes	Agent-level	No	No
Cursor [14]	General programming	LLM-based	Yes	Code line/block/file-level	No	No

scripting customized effects. For example, our system supports the use of different development frameworks to better create scenes according to users' intention, e.g., Phaser to create games, p5.js to create creative coding effects. Such a feature is more difficult to achieve in DrawTalking since it is designed with defined rules. 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 LLM input and output to enable more controllability. For example, DirectGPT [38] 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—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 [24] 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 [65] 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 [5] serves as a visual toolkit for prompt engineering and hypothesis testing, allowing users to iteratively refine their inputs and analyze outputs in an intuitive visual format, thereby enhancing the overall interaction with LLMs. Compared to these works, we use graphical control and direct manipulation to specify both input and output. 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 generate codes for interactive scenes, we employed a mixed-method approach in our formative steps: (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 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”<sup>‡</sup>). 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 [10] 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

<sup>‡</sup><https://www.youtube.com/watch?v=8y7GRYaYYQg>

references and a limited understanding of ChatGPT. The second and third strategies require coding skills to some degree. Sometimes GPT generates incorrect results, so the users need to use their prior knowledge to fix the errors. 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) preparing 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 the Mario jumps 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 ChatGPT: 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. These challenges highlight the complexities users face when leveraging LLMs for interactive scene creation.

### 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 between multiple classes. While these tools can generate basic class structures, they typically lack support

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 [14], 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 for direct manipulation of graphical elements. 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 an ideal LLM-based tool for creating interactive scenes should consider the following points:

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

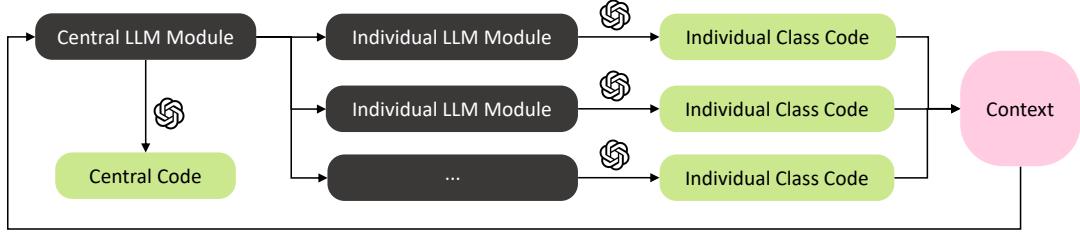
D2. Context-aware code generation: independency will not lose context.

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 an element-level 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, *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).



**Figure 2: The framework of our context-aware LLM modularization technique.** The central LLM module generates and maintains 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 Element-level Context-aware LLM Modularization

2D interactive scenes contain elements in various forms. We define the *element* as a general representation of the content within these scenes, encompassing both individual visual components and broader concepts. For example, a layered character animation includes animations for individual body parts as well as a global transformation, meaning both the parts and the entire body are considered elements in our design.

Our element-level 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, allowing modifications to a single element without affecting others. When creating interactions, the relevant function code is updated within the element class, while the central module uniformly calls these functions. 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, it will search the created Mario's module and continue to update there. This approach allows each element to operate independently, enabling users to customize and enhance each element without disrupting other elements with rapid iteration and testing.

**Central LLM Module.** In contrast to the individual LLM modules, 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 the elements and scripting interactions among elements (e.g., when Mario falls on the spring, Mario bounces up and the spring is

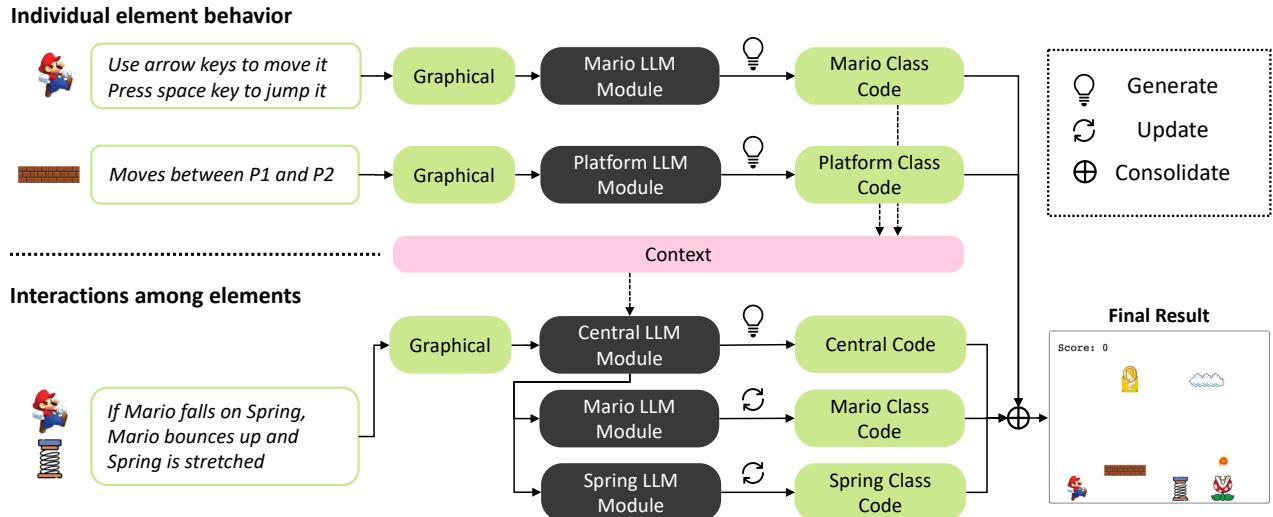
stretched). Importantly, when generating interaction code, it may involve variables and behaviors specific to individual elements. To prevent interference between elements, we instruct it to define the code of variables and functions for each element within their respective classes (e.g., Mario bouncing 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 do 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 updating ensures that the central module remains aware of all changes, promoting a more responsive and flexible operation.

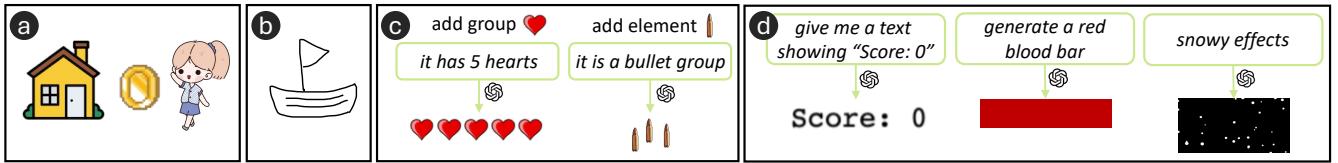
By integrating the strengths of the individual and central LLM modules with contextual communication, our context-aware modularization technique not only enhances independence in code generation but also fosters a more dynamic and interconnected interaction creation.

#### 4.2 Graphical Interface

In our graphical interface *MoGraphGPT* (Figure 6), context-aware LLM modularization and graphical control are seamlessly integrated to facilitate the creation of 2D interactive scenes using natural language inputs and graphical specifications. Our target users are those with no or limited programming skills, and our goal is to help them



**Figure 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.



**Figure 4: Four ways to create elements in our system.** (a) Upload an image. (b) Draw a sketch. (c) Add a group and let LLM generate a group of elements (with a user-uploaded element image), either explicitly mentioning “group” in text prompt or not. (d) Ask LLM to generate elements.

create interactive scenes rather than learning programming. To simplify the user experience and avoid overwhelming newcomers, we do not reveal the generated code in the UI, as common in other tools [1, 2, 39]. Instead, we focus entirely on prompts and graphical elements, encouraging users to engage with this specialized 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 (Figure 4(a)) and draw elements with 2D sketches on the canvas area (Figure 4(b)). 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 (Figure 4(d)), such as texts, graphics, and particle effects. Besides single elements, users can create element groups in two ways: 1) the user can press the “Add Group” button and upload an element image, and then add a text description to let our system generate a group of elements with the uploaded image; 2) the user can upload an image element and let our system generate

an element group with a proper text prompt including words like “group” (Figure 4(c)).

Once the element is created, it is displayed in the canvas area (Figure 6) and rendered in the result area (Figure 6). It opens a dedicated ChatGPT session for that element in the left text pane (Figure 6). Our system automatically switches the GPT session to an element after its selection (by pressing its associated button or clicking on it in the canvas). 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 a 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 5). 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. Users can

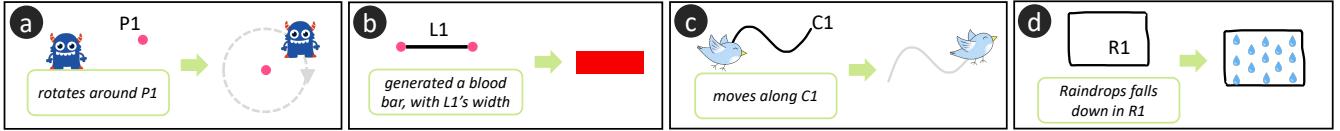


Figure 5: We allow users to specify four types of graphical inputs: (a) point, (b) line, (c) curve, and (d) region. Users can refer to their names in the text prompts.

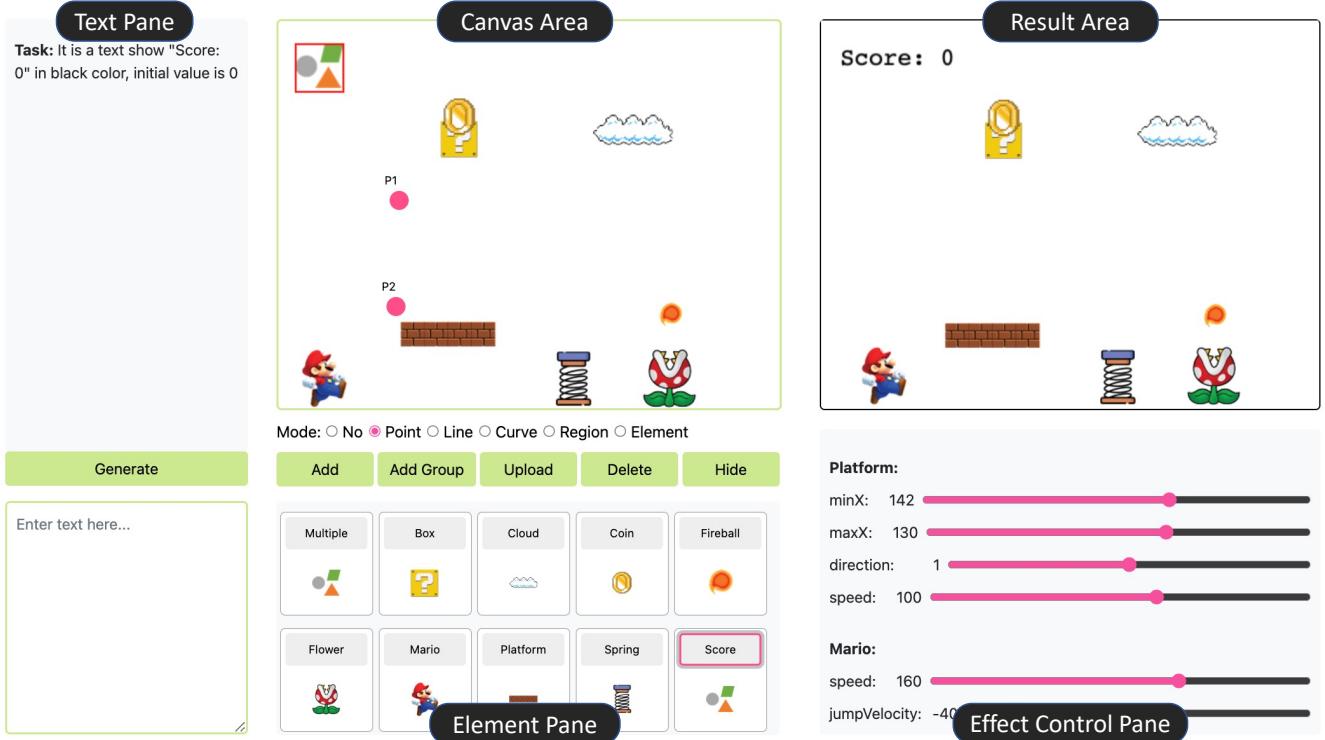


Figure 6: *MoGraphGPT* user interface. Element Pane contains the buttons and preview images for all the created elements in the scene. Canvas Area shows all the elements that can be manipulated by users directly. Once users press the “Generate” button, the result is generated or updated in the Result Area. Effect Control Pane displays the automatically generated parameter values and sliders for precise control.

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. For properties and behaviors of individual elements, users enter the text in the module of each element by selecting 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 “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.** After the code for each element is generated, we let LLM to extract the defined variables and their current

values. Then the system automatically generates sliders and number input fields in the effect control pane (Figure 6), with the range normalized. This allows users to quickly and precisely adjust the parameter values (e.g., movement speed, shake amplitude) without needing to describe the desired changes in text and ask for refinement again.

**Result Testing.** Users can watch and test the created results in the result area (Figure 6) at any time during the creation process. Any change, including text revision, slider revision, 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 [43] 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 is 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 has to follow a certain format. For individual element code, we design the prompt format mainly including *Task*, *Requirement*, *Reference Code Template*, and *Output Format*. The “elementname”, “effect”, and “framework” in the format will be replaced by the created element names, user’s input texts (including the translated graphical information), and suggested framework name. Then they forms a text prompt to GPT, which generates the code and context for each element. For interactions codes among multiple elements, we design the prompt in similar format with individuals, except for an extra context information and the output format. Please refer to the supplementary materials for the specific prompt designs.

## 5.2 Code Integration

Once receiving the response, for individual modules, we extract the code, insert it to the element classes, and insert the context to 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 code generated and replace the original one, and insert the newly defined variables and functions in the relevant individual element class code. The context information is also compared and updated correspondingly. The insert positions is determined by the pre-defined or guided-maintained flags (e.g., “//variable start”, “//variable end”, “//function start”, “//function end”) in the code.

## 6 COMPARATIVE STUDY

To evaluate the effectiveness of element-level modularization, graphical control, and precise refinement of our system, we designed a *within-subject* study to compare *MoGraphGPT* with a state-of-the-art tool. We selected Cursor [14] as the baseline since it is a well-recognized AI code editor that allows for partial code modification and simultaneous updates of files or modules. Participants created interactive effects using both *MoGraphGPT* and Cursor.

### 6.1 Baseline Setup

The Cursor Composer 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 those 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 the underlying code.

## 6.2 Participants and Apparatus

We invited 10 participants (aged 22-34, M: 28, SD: 3.77, 6 females and 4 males, U1-U10) from our personal and university network. They included 4 university students and 6 staff. They had diverse backgrounds, including atmospheric environment (U1, U10), fine arts (U4), interaction design (U8), computer science (U2, U5-6), entrepreneurship (U3), chemistry (U7), and business (U9). On a self-rated 5-point scale (1-no to 5-strong) for coding experience, 2 (U4 and U9) of them rated 1, 3 users (U3, U8, U10) rated 2, 1 user (U7) rated 3, 3 users (U1-2, U5) rated 4, and 1 user (U6) rated 5. In a self-rated 5-point scale (1-no to 5-strong) for ChatGPT using experience, 2 users (U9-10) of them rated 2, 3 users (U1, U7-8) rated 3, 3 users (U3-5) rated 4, and 2 users (U2, U6) rated 5. They used ChatGPT for searching information (U1, U4), polishing writing (U1-3, U6-7, U9-10), checking codes (U2, U6-7), generating codes (U2, U5). The study was conducted on a laptop running both systems, and the participants could use a keyboard, touchpad, and mouse for inputting.

## 6.3 Tasks

We designed three tasks (Figure 7) for users to reproduce the following effects using *MoGraphGPT* and Cursor Composer: (Task1) a fish moves from a specific point to another point; (Task2) a fish moves along a curved path with a constant speed; (Task3) a three-step iterative animation: place the sun and the earth at specific positions on the canvas, then let the earth rotate around its own center, and finally let the earth orbit the sun while keeping self-rotation. These three tasks are common in 2D interactive scenes and involve typical features such as behaviors of single elements and interactions between two elements, spatial properties like positions, translations, rotations, paths, and speed. Users were required to create their results as similar as 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 starting point and ending point in the canvas; (Task2) the moving path and the speed; (Task3) the positions of the sun and earth and the rotation speeds of the earth.

## 6.4 Procedure

We gave participants a 15-minute introduction of the tasks and two systems and allow 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 can 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 has been reproduced successfully, it was double-checked by two of our authors. If both of us reach a consensus, it was considered a complete result. He or she can move to the next step. If the participant tried over

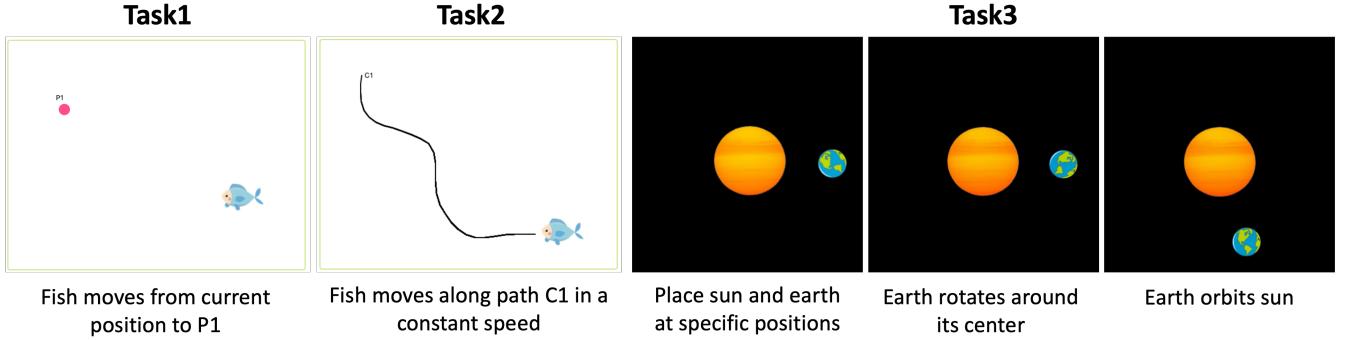


Figure 7: Three tasks in the comparative study.

**Table 2: The performance of MoGraphGPT and Cursor across three metrics averaged over 10 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	402.40	4.80	27.70
Cursor	1339.00	17.20	269.30
Reduction	69.57%	69.34%	89.21%

10 minutes for similar text prompts but the system still does not provide a clear result, or the participant thinks the effect is very difficult to achieve and he/she does not have any idea for it, it is considered a incomplete result, and it can move to the next step as well. After completing all the tasks, they were asked to fill in the questionnaire on a 5-Likert scale. The questions are elaborated in Figure 8. We then conducted semi-structured interviews with them to collect their feedback, including the differences between *MoGraphGPT* and Cursor Composer and our observations during the study. We recorded the time spent on each task, text prompts, operations, and results. The whole process was audio-recorded and later transcribed with their agreement by filling out an informed consent form. In compensation for their time, each participant received a 13-USD gifted card for about one-hour participation.

## 6.5 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 8). Objective metrics consist of time taken, the number of prompts, and prompt word counts. We conducted Wilcoxon signed-rank tests to analyze significant differences.

## 6.6 Results

**Completion.** All participants successfully completed each task in our system within 6 minutes. However, U7 spent over 10 minutes on Task1 and U4-6 and U8-10 spent over 10 minutes on Tasks using

Cursor but were still unhappy with their results. 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 10 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. The results indicate *MoGraphGPT* achieves desired outcomes in significantly less time and with fewer prompts than Cursor.

**Time.** The average time spent on each task using *MoGraphGPT* is significantly lower than Cursor, as confirmed by the Wilcoxon signed-rank test ( $p < 0.01$ ): Task1: M: 46.1s (SD: 12.5s) and M: 268.0s (SD: 134.4s) for *MoGraphGPT* and Cursor, respectively; Task2: M: 152.1s (SD: 73.9s) for *MoGraphGPT* and M: 448.0s (SD: 110.7s) for Cursor; Task3: M: 204.2s (SD: 79.6s) for *MoGraphGPT* and M: 623.0s (SD: 161.6s) for Cursor. Here, we trim the time for cases over 10 minutes to 10 minutes. Our graphical specification feature enables participants to quickly define positions, moving paths, and both absolute and relative positions quickly. Participants (U1, U4, U6-7, U9-10) were able to adjust motion parameters, such as speed, using sliders and numerical inputs with precision. In contrast, participants using Cursor spent considerable time formulating prompts to integrate graphical information. Some (U1, U4-6, U8) struggled with precise descriptions for several minutes, especially when Cursor continued to produce undesired results. Participants (U1-3, U5, U7-9) 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. In *MoGraphGPT*, individual and interaction behaviors can be created in the earth and central sessions just one trial. This is because our modular structure provides 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

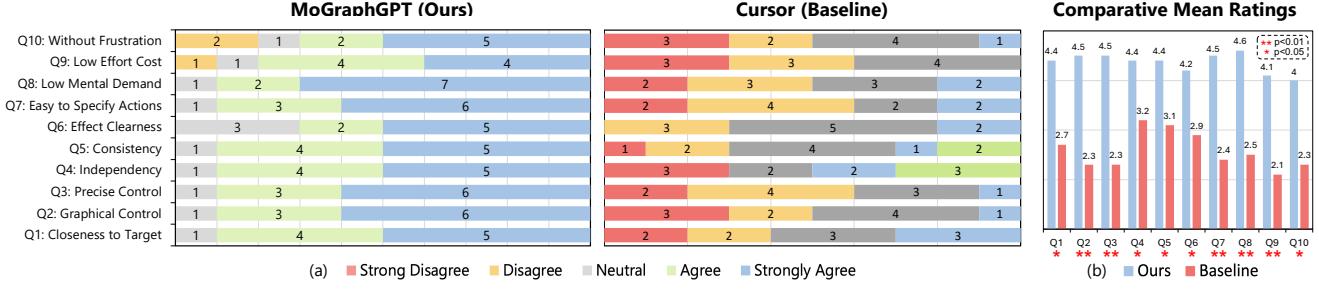


Figure 8: Subjective ratings on *MoGraphGPT* and *Cursor Composer*. For the scores, the higher, the better.

only soft constraints, causing interaction code and individual behavior code to become intertwined, leading to inconsistent updates and chaotic modifications.

**Prompt numbers and length.** Our system results in significantly fewer and shorter prompts, as confirmed by Wilcoxon signed-rank tests ( $p < 0.01$ ). In Task1 and Task2, the participants used pronouns to refer to each element in element sessions and mentioned the created graphical proxies (e.g., P1, C1) in their prompts. In contrast, accurate mention of element names and graphical details is required in Cursor.

Participants described the relative positions to the canvas (e.g., left-top, right-bottom, U1-3, U5, U7-9), estimated specific coordinates (U4, U6, U10), 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). They adjusted coordinates through multiple iterations and articulated the curved motion path with terms like "curves with two circles" (U2), "waved curves" (U7), and "tilde" (U1, U3, U9). They further specified the shape with phrases such as "make it more curved" (U2) 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 target effects on the first trial. Success typically came only after participants added the earth file as context and retried multiple times. Some participants (U1-2, U5, U8-10) 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". In contrast, with *MoGraphGPT*, participants could input the self-rotation instruction in the earth module and the orbiting effect in the central module independently, requiring less prompt engineering effort.

**Subjective ratings and qualitative feedback.** We analyzed the participants' rating and feedback, distilling our findings into four key aspects.

**User Overall Experience.** Overall, subjective ratings and feedback indicate that our system offers a more intuitive, easy-to-use, and effective experience compared to Cursor. As shown in Figure 8, *MoGraphGPT* significantly outperformed Cursor across all metrics. Wilcoxon signed-rank tests confirmed the significance of all aspects ( $p < 0.05$ ). As shown in Figure 8 (b), the participants had a strong preference for *MoGraphGPT*, particularly in terms of Q2 - Graphical Control, Q3 - Precise Control, Q7 - Easy to Specify Actions, Q8 - Low Mental Demand, and Q9 - Low Effort Cost ( $p < 0.01$ ).

**Strength of Modularization.** Participants (U2, U4-6, U7, U9) 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."

**Strength of Graphical Control.** Participants appreciated our GUI for its intuitive manipulation of visual elements (Q2). Several participants (U1, U4, U6, U9) were surprised that they could create target spatial effects in just a few seconds using our system (Q7). Even those who redrew curves in Task 2 to better match the target path (U2, U8-9) expressed a willingness to experiment without significant effort (Q8 & Q9). Most participants noted that relying solely on text to describe shapes made it difficult and cumbersome for Cursor to generate accurate results, leading to frustration (Q10).

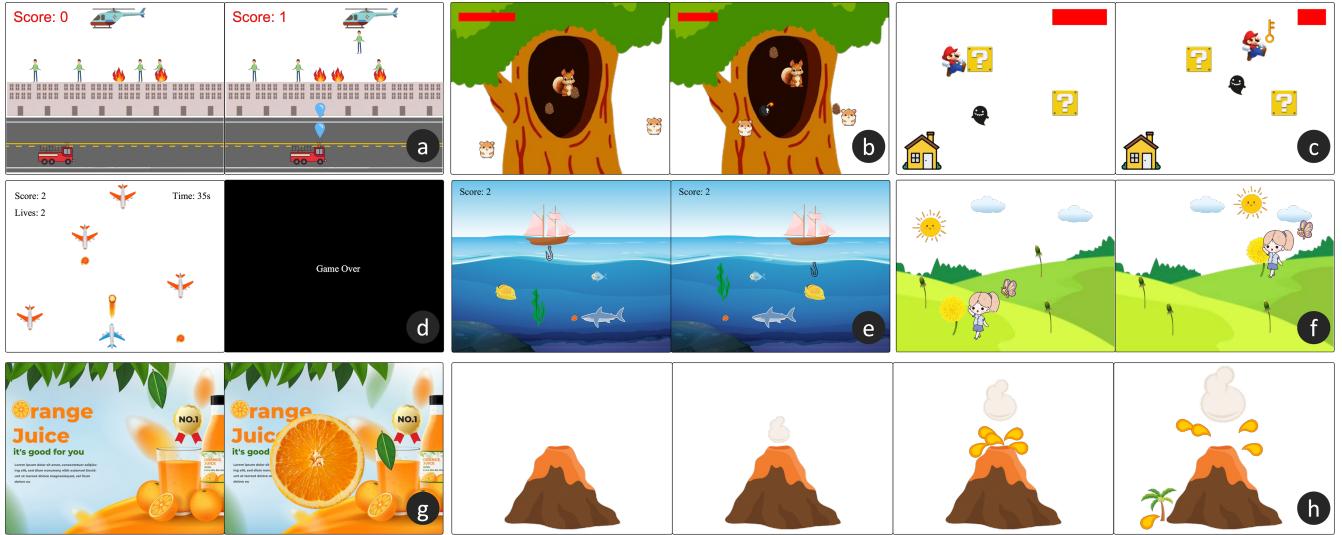
**Strength of Precise Control.** The precise control over effect parameters (Q3) in *MoGraphGPT* was praised by participants. They highlighted its intuitiveness and effectiveness, even for the interaction effects with multiple elements (e.g., orbit radius, orbit speed). In contrast, they found that using text to adjust parameters in Cursor "does not have a reference" (U7) and required balancing between excessive and inadequate control (U2, U9).

## 7 OPEN-ENDED STUDY

To further evaluate the usability and expressiveness of *MoGraphGPT*, we invited participants for an open-ended study, allowing them to create their own desired 2D interactive scenes freely.

**Participants and Apparatus.** We recruited 6 participants (aged 25-33, M: 29.5, SD: 2.66, 4 females and 2 males, P1-6), and three of them participated in our comparative study. The study was conducted on a laptop or a tablet running *MoGraphGPT*, and the participants could use a keyboard, touchpad, stylus, and mouse for inputting and drawing.

**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 element images for those elements requiring image uploading from our devices. At the beginning of the study, after a brief introduction and guidance of our system, the participants started creating 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.

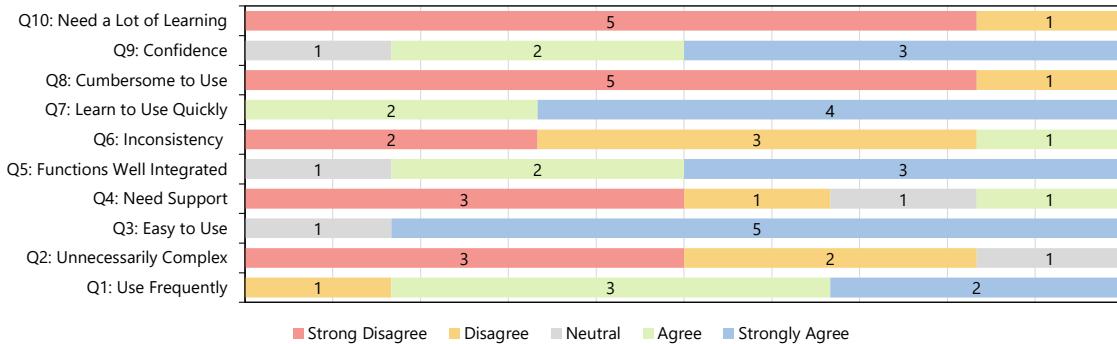


**Figure 9: A gallery of selected results in the open-ended study.** (a) Two-player Rescue Game (P2). (b) Squirrel Guard Game (P1). (c) Scavenger Hunt Game (P4). (d) Airplane War Game (P4). (e) Sea Fishing Game (P6). (f) Interactive Animation Demo (P3). (g) Website Ad Design Demo (P5). (h) Dynamic Illustration for Academic Paper (P6). Please refer to supplementary materials for detailed descriptions.

**Results.** The participants created 10 results in total. Figure 9 shows parts of result snapshots, including 5 games (1 two-player game (Figure 9(a)) and 5 single-player games (Figure 9(b)-(e))), 1 interactive animation demo (Figure 9(f)), 1 website ad interaction design demo (Figure 9(g)), and 1 dynamic illustration for an academic paper (Figure 9(h)). Each result contains 4-8 elements and various types of single element behavior and interactions among multiple elements. Each result was completed between 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 along a designated path (Figure 9(f)). 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 9(b)), enhancing the scene's liveliness. Additionally, users set up clickable regions (Figure 9(g)), such as an orange that, when clicked, displayed an image of the fruit, fostering engagement and interaction. The results showed that users could create and edit various individual objects while facilitating interactions between them. They utilized text inputs to modify or redefine the behavior of single elements and employed automatically generated sliders to fine-tune specific details. Importantly, the operations on individual objects did not affect others or their interactions, ensuring clarity and control in complex scenarios. Overall, the findings indicate that our *MoGraphGPT* system effectively supports user creativity and exploration in graphical interactions, making the creative process both enjoyable and intuitive.

The SUS score rated by the participants is 85 on a 100-point scale, indicating our system has good usability. The distribution of the SUS score for each question is shown in Figure 10. 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) 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 (P3, P6). For instance, one participant with an environmental science background (P6) mentioned that generating dynamic visualizations for his papers was often challenging. With our system, he could quickly and easily create effective graphics to illustrate the core concepts of his work (Figure 9(h)).

The participants with programming experience also found our system very useful. One participant with 8-year programming experience, P3, noted that if she were to create a game using traditional methods, it would take her at least two hours, but with our system, she completed it in just 15 minutes. This significantly saved her time, effort, and the need to learn new programming languages or frameworks, as she could directly generate game code through natural language and graphic controls. The participants (P1, P4) mentioned that our system did not require a technical person to guide them closely; a simple introduction and brief instructions at the start were sufficient for them to get up to speed quickly. P1 particularly enjoyed that our system offers a “modular architecture”. This capability allows users to navigate complex relationships among objects more effectively. P1 said, “even for programming



**Figure 10: SUS score distribution. The question description is the key points from the full SUS questions.**

simple games, this system is much more easier to use than ChatGPT”.

However, the participants also highlighted some limitations of our system, such as the currently limited graphic controls (P5–6), which only include points, straight lines, curves, and user-drawn regions. They desired additional geometric shapes, such as circles, rectangles, and triangles, as well as the ability to automatically segment imported background images. The ability of code generation is also questioned by participants (P2–3) in some aspects including adding unnecessary conditions. For example, in creating Scavenger hunt Game (Figure 9(c)), when scripting the effect of a key appears and always follows Mario after Mario comes close to the box, it first generates the result that the key only follows Mario when Mario is close to the box. It adds a condition determination on whether Mario collides with the box, while it is unnecessary in the participant’s expectation.

## 8 DISCUSSIONS AND FUTURE WORK

### 8.1 Modularization with LLMs

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. Despite advancements in LLM models, the problem of interdependence is likely to persist for the foreseeable future and is not confined to specific iterations, such as different versions of GPT. Our technique presents a novel and valuable approach applicable to other LLM-based frameworks (e.g., Google Gemini, Claude, LLaMA), extending beyond just 2D visual creation to encompass a broader range of interactive scenarios.

### 8.2 Trade-off between Modularization and Original ChatGPT

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. Users are required to manually delineate the behaviors of each element, in contrast to the original ChatGPT interface, where they can combine these elements more fluidly, although it often results in undesired and uncontrolled outcomes.

Looking ahead, we plan to develop an advanced input parsing and segmentation method that will allow users to input their descriptions without the need for strict separation. Our system will intelligently analyze and partition the descriptions of both individual and multiple elements, feeding them into distinct modules as necessary. This enhancement aims to strike a balance between user control and the fluidity of interaction, ultimately improving the overall user experience.

### 8.3 Limitation of Context-awareness

Although we provide context for code generation, when tasks become extremely complex and conversations lengthen, the issue of context loss for individual element creation persists. The inherent token limits of LLMs restrict the amount of information that can be processed within a single session. As context accumulates, it contributes to the text prompt, which can lead to truncated or incomplete responses, ultimately hindering the model’s ability to generate coherent and contextually relevant outputs.

To address these challenges, future work should focus on developing effective context management strategies that prioritize and summarize essential information while discarding less relevant details. Implementing dynamic context windowing could allow the model to allocate more tokens to critical portions of the conversation while compressing less relevant exchanges. Additionally, exploring chunking mechanisms to break down complex tasks into smaller, manageable parts could help preserve context during longer interactions. By enhancing context-awareness in these ways, we aim to improve the robustness and coherence of LLM outputs, even in complex scenarios.

### 8.4 Potential Use Cases

The open-ended study indicated that *MoGraphGPT* can be used for creating dynamic and interactive scenarios for diverse purposes, such as interactive gaming experiences, engaging educational tools, visual communication in research, and UI design. Beyond these applications, we envision its use in other domains. For example, users can create their own medication reminders, journaling tools, and interactive mood trackers, by creating 2D representations of medicine, feelings, and mood and then crafting interactivity for them. Multiple

family members can co-create interactive shared albums by importing photos to *MoGraphGPT* and designing visualization patterns for memory recording and photo management. In addition, museums and exhibition centers can leverage *MoGraphGPT* to enable visitors to express their feelings by creating interactive featured items, which can be analyzed for insights into visitor preferences. Furthermore, *MoGraphGPT* can facilitate interactive storytelling, enabling children to create their own books and share their narratives in a fun and engaging way.

### 8.5 Integrating Code Representations and Element Visualizations

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. Besides, we are considering the development of educational modules aimed at teaching programming fundamentals. These components will be designed to prepare users for future coding tasks through engaging scene creation.

Incorporating visual chips into the UI design presents a significant opportunity to enhance user interaction. Recent generative AI tools [30, 62] effectively utilize chips to visually associate text prompts with corresponding visual elements, allowing users to quickly identify and swap components such as subjects and adjectives. While our current UI lists raw text, exploring a chip-based interface in future iterations could greatly improve user experience and accessibility. By visually representing elements like “Mario,” “Platform,” and “P2,” we can make it easier for users to understand and manipulate their inputs.

### 8.6 Scalability for Complex Scenarios

As more elements are added to a scene, the behavior and interactions among them can become complex. Users may need to view and manage these elements, their relationships, and interaction logic. To support this, we can enhance our system by integrating a graph representation [71], where each node represents an element and each edge represents their relationships, with additional connected lines indicating the next event logic. This allows us to visualize both element behaviors and interactions as attributed objects [68], after the system generates corresponding codes. Users can then easily reuse these behaviors and interactions by dragging objects to other elements in the graph, as well as directly reconnecting or deleting elements and edges for quick management.

In our current implementation, *MoGraphGPT* supports creating results containing multiple scenes (e.g., Figure 9(d)) by switching in the central module to manage the visibility of created elements. However, more complex applications may contain more scenes. Our current implementation involves two levels: scene and scene elements. We might add one more level for scene management, which opens modules for scenes. Users can manually create scenes and add elements, or we can extract scenes and elements from users’ text input. We can visualize scenes as workflow UIs, connected by key condition variables, allowing users to manipulate

scene order and relationships directly. This will open new opportunities for exploring conditional layered and nested LLMs in various applications.

## 9 CONCLUSION

This paper has introduced *MoGraphGPT*, a novel LLM-based system to simplify the creation of 2D interactive scenes without coding from natural language input and graphical control. We utilized content analysis on video tutorials about creating interactive scenes using ChatGPT and existing AI coding tools, and distilled several issues including the lack of independent generation and refinement, graphical control, and precise refinement. Based on these findings, we proposed a context-aware modularization technique that processes textual descriptions through individual LLM modules, with a central module coordinating interactions, allowing for independent refinement of each element. Our graphical user interface combined these modular LLMs with advanced graphical controls, enabling seamless code generation for 2D interactive scenes and direct integration of graphical information. We conducted a comparative study between *MoGraphGPT* and Cursor Composer, and found *MoGraphGPT* significantly reduced the time, prompt trials, and prompt lengths and achieved better graphical and precise control in creating interactive scenes. Another open-ended usability study demonstrated that *MoGraphGPT* allowed users to create various desired scenes easily without the need for coding, benefiting diverse application scenarios.

## REFERENCES

- [1] [n. d.]. Flutter - Build apps for any screen. <https://flutter.dev/>. Accessed: 2024-11-26.
- [2] [n. d.]. Python Playground - Online Python IDE. <https://programiz.pro/ide/python>. Accessed: 2024-11-26.
- [3] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. 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*. 1–22.
- [4] Asad Anjum, Yuting Li, Noelle Law, Megan Charity, and Julian Togelius. 2024. The Ink Splotch Effect: A case study on ChatGPT as a co-creative game designer. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*. 1–15.
- [5] Ian Arawjo, Priyan Vaithilingam, Martin Wattenberg, and Elena Glassman. 2023. 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*. 1–3.
- [6] Claude Artifacts. 2024. Claude Artifacts. <https://madewithclaude.com/> Accessed: 2024-12-10.
- [7] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawska, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczek, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.
- [8] Giorgio Bimbelli, Daniela Fogli, Luigi Gargioni, et al. 2023. Can ChatGPT support end-user development of robot programs?. In *CEUR Workshop Proceedings*, Vol. 3408.
- [9] Stephen Brade, Bryan Wang, Mauricio Sousa, Sageev Oore, and Tovi Grossman. 2023. 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*. 1–14.
- [10] Kathy Charmaz. 2008. Constructionism and the grounded theory method. *Handbook of constructionist research* 1, 1 (2008), 397–412.
- [11] Liuqing Chen, Shuhong Xiao, Yunlong Chen, Yaxuan Song, Ruoyu Wu, and Lingyun Sun. 2024. 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*. 1–19.
- [12] Mengyu Chen, Marko Peljhan, and Misha Sra. 2021. Entanglevr: A visual programming interface for virtual reality interactive scene generation. In *Proceedings of the 27th ACM symposium on virtual reality software and technology*. 1–6.

- [13] CodePen. 2012. CodePen. <https://codepen.io> Accessed: 2024-09-11.
- [14] Cursor. 2023. Cursor - The AI Code Editor. <https://www.cursor.com/> Accessed: 2024-11-30.
- [15] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. 2022. 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).
- [16] Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. 2024. Llmr: Real-time prompting of interactive worlds using large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–22.
- [17] Exploratorium. 1993. Exploratorium. <https://www.exploratorium.edu> Accessed: 2024-09-11.
- [18] GitHub. 2021. GitHub Copilot. <https://github.com/features/copilot> Accessed: 2024-09-11.
- [19] Glitch Team. 2017. Glitch. <https://glitch.com> Accessed: 2024-09-11.
- [20] Tracy G Harwood and Tony Garry. 2003. An overview of content analysis. *The marketing review* 3, 4 (2003), 479–498.
- [21] Yihan Hou, Manling Yang, Hao Cui, Lei Wang, Jie Xu, and Wei Zeng. 2024. C2Ideas: Supporting Creative Interior Color Design Ideation with a Large Language Model. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–18.
- [22] Chengpeng Hu, Yunlong Zhao, and Jialin Liu. 2024. Generating Games via LLMs: An Investigation with Video Game Description Language. *arXiv preprint arXiv:2404.08706* (2024).
- [23] Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, et al. 2024. ANPL: towards natural programming with interactive decomposition. *Advances in Neural Information Processing Systems* 36 (2024).
- [24] Peiling Jiang, Jude Rayan, Steven P Dow, and Haijun Xia. 2023. Graphologue: Exploring large language model responses with interactive diagrams. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–20.
- [25] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: sketching dynamic and interactive illustrations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. 395–405.
- [26] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: bringing life to illustrations with kinetic textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 351–360.
- [27] Khan Academy. 2008. Khan Academy. <https://www.khanacademy.org> Accessed: 2024-09-11.
- [28] Tae Soo Kim, Yoonjoo Lee, Minsuk Chang, and Juho Kim. 2023. 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*. 1–18.
- [29] Chong Lan, Yongsheng Wang, Chengze Wang, Shirong Song, and Zheng Gong. 2023. Application of ChatGPT-Based Digital Human in Animation Creation. *Future Internet* 15, 9 (2023), 300.
- [30] Baiqi Li, Zhiqin Lin, Deepak Pathak, Jiayao Li, Yixin Fei, Kewen Wu, Xide Xia, Pengchuan Zhang, Graham Neubig, and Deva Ramanan. 2024. Evaluating and Improving Compositional Text-to-Visual Generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5290–5301.
- [31] Fu Li, Jiaming Huang, Yi Gao, and Wei Dong. 2023. ChatIoT: Zero-code Generation of Trigger-action Based IoT Programs with ChatGPT. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*. 219–220.
- [32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
- [33] Jingyuan Liu, Hongbo Fu, and Chiew-Lan Tai. 2020. Posetween: Pose-driven tween animation. In *Proceedings of the 33rd annual ACM symposium on user interface software and technology*. 791–804.
- [34] Vivian Liu, Rubaiat Habib Kazi, Li-Yi Wei, Matthew Fisher, Timothy Langlois, Seth Walker, and Lydia Chilton. 2024. LogoMotion: Visually Grounded Code Generation for Content-Aware Animation. *arXiv preprint arXiv:2405.07065* (2024).
- [35] Vivian Liu, Han Qiao, and Lydia Chilton. 2022. Opal: Multimodal image generation for news illustration. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–17.
- [36] Yue Liu, Thanh Le-Cong, Ratnadiria Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2024. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–26.
- [37] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering* (2024).
- [38] Damien Masson, Sylvain Malacria, Géry Casiez, and Daniel Vogel. 2024. Directgpt: A direct manipulation interface to interact with large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16.
- [39] MIT Media Lab. 2003. Scratch. <https://scratch.mit.edu> Accessed: 2024-09-11.
- [40] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Amy Ko. 2008. How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 177–184.
- [41] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
- [42] Nintendo. 1986. The Legend of Zelda. [https://en.wikipedia.org/wiki/The\\_Legend\\_of\\_Zelda](https://en.wikipedia.org/wiki/The_Legend_of_Zelda) Accessed: 2024-09-11.
- [43] OpenAI. 2023. ChatGPT. <https://chat.openai.com> Accessed: 2024-09-11.
- [44] Pajitnov, Alexey. 1984. Tetris. <https://en.wikipedia.org/wiki/Tetris> Accessed: 2024-09-11.
- [45] Phaser Team. 2013. Phaser. <https://phaser.io> Accessed: 2024-09-11.
- [46] Project Jupyter. 2014. Jupyter Notebook. <https://jupyter.org> Accessed: 2024-09-11.
- [47] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weizh Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15174–15186.
- [48] Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C Shepherd. 2022. Can guided decomposition help end-users write larger block-based programs? a mobile robot experiment. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 233–258.
- [49] Karl Toby Rosenberg, Rubaiat Habib Kazi, Li-Yi Wei, Haijun Xia, and Ken Perlin. 2024. DrawTalking: Building Interactive Worlds by Sketching and Speaking. *arXiv preprint arXiv:2401.05631* (2024).
- [50] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [51] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. 2024. Quality Assessment of ChatGPT Generated Code and their Use by Developers. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 152–156.
- [52] Snap. 2023. Lens Studio. <https://ar.snap.com/en-US/lens-studio> Accessed: 2024-09-11.
- [53] Shyam Sudhakaran, Miguel González-Duque, Matthias Freiberger, Claire Glamois, Elias Najarro, and Sebastian Risi. 2024. Mariopt: Open-ended textlevel generation through large language models. *Advances in Neural Information Processing Systems* 36 (2024).
- [54] Penny Sweetser. 2024. Large language models and video games: A preliminary scoping review. In *Proceedings of the 6th ACM Conference on Conversational User Interfaces*. 1–8.
- [55] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant—how far is it? *arXiv preprint arXiv:2304.11938* (2023).
- [56] Graham Todd, Sam Earle, Muhammad Umair Nasir, Michael Cerny Green, and Julian Togelius. 2023. Level generation through large language models. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*. 1–8.
- [57] Tiffany Tseng, Ruijia Cheng, and Jeffrey Nichols. 2024. Keyframer: Empowering Animation Design using Large Language Models. *arXiv preprint arXiv:2402.06071* (2024).
- [58] Unity Technologies. 2005. Unity. <https://unity.com> Accessed: 2024-09-11.
- [59] University of Colorado Boulder. 2002. PhET Interactive Simulations. <https://phet.colorado.edu> Accessed: 2024-09-11.
- [60] Unreal Engine. 2023. Introduction to Blueprints. <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/> Accessed: 2024-09-11.
- [61] Vercel. 2024. v0 by Vercel. <https://v0.dev/> Accessed: 2024-12-10.
- [62] Zhijie Wang, Yuheng Huang, Da Song, Lei Ma, and Tianyi Zhang. 2024. PromptCharm: Text-to-Image Generation through Multi-modal Prompting and Refinement. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–21.
- [63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [64] Wikipedia contributors. 2023. Super Mario. [https://en.wikipedia.org/wiki/Super\\_Mario](https://en.wikipedia.org/wiki/Super_Mario) Accessed: 2024-09-11.
- [65] Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671* (2023).
- [66] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–10.

- [67] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. 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*. 1–22.
- [68] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. 2016. Object-oriented drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 4610–4621.
- [69] Shishi Xiao, Liangwei Wang, Xiaojuan Ma, and Wei Zeng. 2024. TypeDance: Creating semantic typographic logos from image through personalized generation. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–18.
- [70] Jun Xing, Rubaiat Habib Kazi, Tovi Grossman, Li-Yi Wei, Jos Stam, and George Fitzmaurice. 2016. Energy-brushes: Interactive tools for illustrating stylized elemental dynamics. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 755–766.
- [71] Zihan Yan, Chunxu Yang, Qihao Liang, and Xiang'Anthony' Chen. 2023. XCreation: A Graph-based Crossmodal Generative Creativity Support Tool. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–15.
- [72] Daijin Yang, Erica Kleinman, and Casper Harteveld. 2024. GPT for Games: A Scoping Review (2020–2023). *arXiv preprint arXiv:2404.17794* (2024).
- [73] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2024).
- [74] Hui Ye, Jiaye Leng, Pengfei Xu, Karan Singh, and Hongbo Fu. 2024. ProInterAR: A Visual Programming Platform for Creating Immersive AR Interactions. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–15.
- [75] Ryan Yen, Jiawen Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2023. Coladder: Supporting programmers with hierarchical code generation in multi-level abstraction. *arXiv preprint arXiv:2310.08699* (2023).
- [76] Enes Yigitbas, Jonas Klauke, Sebastian Gottschalk, and Gregor Engels. 2023. End-user development for interactive web-based virtual reality scenes. *Journal of Computer Languages* 74 (2023), 101187.
- [77] JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. 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*. 1–21.
- [78] Lei Zhang and Steve Oney. 2020. 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*. 342–353.
- [79] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879* (2023).