

# AtomXR: Streamlined XR Prototyping with Natural Language and Immersive Physical Interaction

1 Alice Cai  
23 acai@college.harvard.edu  
4 Harvard University  
5 Cambridge, Massachusetts, USA6 Caine Ardayfio  
78 cardayfio@college.harvard.edu  
9 Harvard University  
10 Cambridge, Massachusetts, USA11 AnhPhu Nguyen  
1213 anugyen1@college.harvard.edu  
14 Harvard University  
15 Cambridge, Massachusetts, USA16 Tica Lin  
1718 mlin@g.harvard.edu  
19 Harvard University  
20 Cambridge, Massachusetts, USA21 Elena Glassman  
2223 eglassman@g.harvard.edu  
24 Harvard University  
25 Cambridge, Massachusetts, USA

## ABSTRACT

As technological advancements in extended reality (XR) amplify the demand for more XR content, traditional development processes face several challenges: 1) a steep learning curve for inexperienced developers, 2) a disconnect between 2D development environments and 3D user experiences inside headsets, and 3) slow iteration cycles due to context switching between development and testing environments. To address these challenges, we introduce AtomXR, a streamlined, immersive, no-code XR prototyping tool designed to empower both experienced and inexperienced developers in creating applications using natural language, eye-gaze, and touch interactions. AtomXR consists of: 1) AtomScript, a high-level human-interpretable scripting language for rapid prototyping, 2) a natural language interface that integrates LLMs and multimodal inputs for AtomScript generation, and 3) an immersive in-headset authoring environment. Empirical evaluation through two user studies offers insights into natural language-based and immersive prototyping, and shows AtomXR provides significant improvements in speed and user experience compared to traditional systems.

## CCS CONCEPTS

- Human-centered computing → Human computer interaction (HCI);
- Computing methodologies → Artificial intelligence.

## KEYWORDS

extended reality, virtual reality, augmented reality, application development, prototyping, natural language processing, immersive development, programming languages

## 1 INTRODUCTION

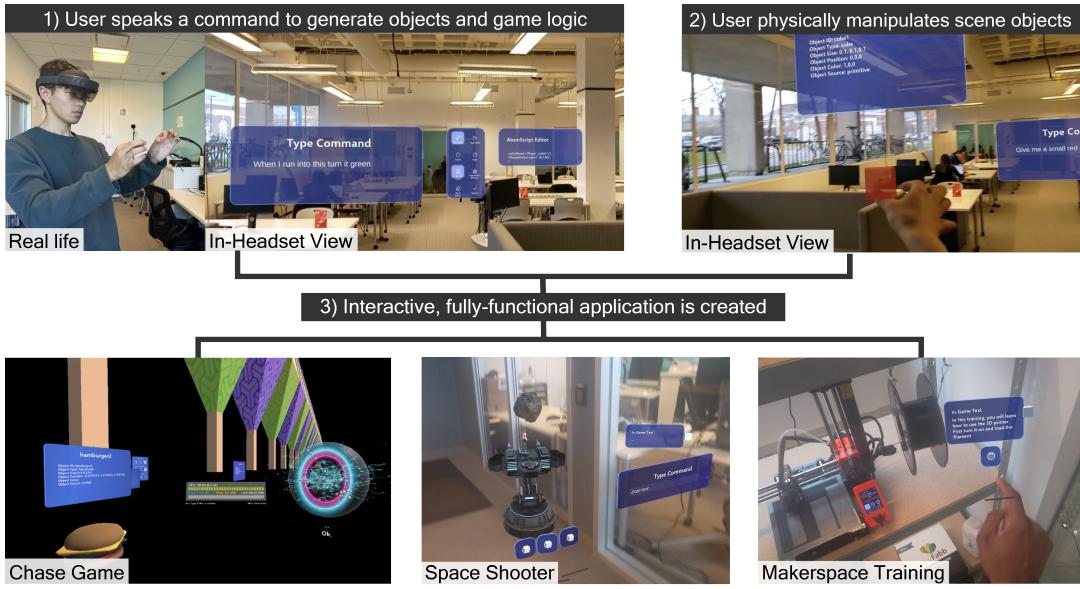
The recent revival of the extended reality (XR) industry, propelled by the pandemic-driven shift toward virtual activities as well as technical advancements in computing and display hardware, calls for easy and effective XR application development tools. XR bears potential to impact a wide range of fields, spanning education, entertainment, productivity, training, manufacturing, and more [15]. Yet existing XR development processes produce significant friction due to their inefficiency and substantial barriers to entry. These barriers include the steep learning curve of development tools, the mismatch between development environments on 2D screens, and the 3D user experience inside the headset, and the long

testing and iteration cycles necessary in conventional workflows where applications are compiled on the computer and then deployed onto the target headset. Not only do these inefficiencies of XR development practices slow progress in the field, they also exclude those without technical or programming experience, reducing the diversity, richness, and potential utility of applications created in XR.

To address these issues, we introduce a streamlined application development system, AtomXR, that uses hybrid interaction methods (voice, eye gaze, touch) to enable no-code immersive authoring. While no-code development tools have been desired in this space for a long time, only recently have advancements in natural language processing (NLP), such as the development of large language models (LLMs) like GPT-3 [10] and GPT-4 [27], enabled this kind of system. AtomXR aims to both improve the prototyping process for experienced developers and empower non-experts to create XR applications.

Specifically, AtomXR allows users to create, update, and delete objects and application logic without code. Instead, these operations are done using natural language, eye-gaze, and touch interactions while inside the headset. Users can describe what they want to build, and our system translates their natural language requests into AtomScript, a high-level interpreted language that aims to be easy to read by users and easy to generate using language models. Using eye-gaze and gesture interactions, users can then reference and manipulate objects in the scene to further edit and build the application. In enabling rapid, low-code development of complex experiences, we aim to drive progress towards a new era of AI-enabled and accessible immersive authoring.

We investigated the usefulness and interaction dynamics of the AtomXR system through two experiments involving a total of 24 participants with various levels of prior technical and XR experience. The first experiment compared user performance on a series of development tasks in AtomXR with the traditional Unity-based desktop system. Our system demonstrated significant improvements in accessibility, efficiency, usability, and new feature discovery. Participants were able to complete tasks 2-5 times as fast in the AtomXR systems, and were able to complete more than 2 times as many tasks. Additionally, AtomXR was perceived to be significantly easier to learn and more intuitive to use. The second experiment investigated user interaction dynamics with the major components



**Figure 1: System Preview of AtomXR:** In the top left image, a user speaks commands to make objects and game logic. The top-right image features an in-headset view of a user using gestures to manipulate the scene. The bottom row shows several games made with AtomXR, including the chase game, where a user must run away from a chaser. The middle game is the space shooter, where the user moves the turret left and right to shoot at incoming asteroids. The bottom right image is a makerspace training experience teaching user’s how to use the equipment in a makerspace. Each of these 3 experiences was created by speaking natural language commands describing the experience in AtomXR.

of the AtomXR system, including programming abstraction, natural language-driven logic generation, and immersive authoring. We synthesize experimental insights on these features and discuss future usage and implications.

Our contributions include both engineering efforts and experimental insights, and are summarized as follows:

- AtomXR, a no-code, XR development engine supported by:
  - A multimodal approach to interpreting user intent using speech, physical interactions, and eye gaze.
  - AtomScript, a high-level programming language for XR applications.
- Two studies evaluating the effectiveness and user experience of 1) AtomXR compared to a traditional Unity-based system and 2) individual features provided by AtomXR.

## 2 RELATED WORK

### 2.1 3D & Immersive Authoring

While individual 3D development systems vary in the processes they use, there are common capabilities shared among nearly all development systems. Generally, these systems support creating, deleting, and updating the properties of virtual objects, arranging them in a scene, and defining their interactions and behaviors over time. Popular game engines, such as Unity [4] and Unreal Engine [3] are based on the entity-component architecture, which allows developers to easily add and modify an object’s properties and behaviors by adding, modifying, or deleting components associated

with the object. Other approaches to 3D development include WebXR methods, which use libraries and frameworks like Three.js [13] and A-FRAME [6] that enable browser-based creation and rendering of 3D applications. All these development environments, however, suffer from steep learning curves when adding functional components as they require prior knowledge of programming languages like C# and Javascript.

A number of development tools have emerged to reduce friction and complexity in XR development. DART [30] was a collection of extensions to a multimedia programming environment that enabled modular composition of functionality for rapid AR prototyping. ProtoAR [35] generates mobile AR views from 2D drawings on paper and Playdoh, but only supports object creation/management and does not support logic generation. Earlier authoring tools like VRML97 [34] and X3D [11] use event-passing mechanisms to define user interactions. More recent examples include Unreal Engine’s Blueprints [42], a node-based visual scripting system, and Alice [37], a block-based programming environment that allows users to rapidly prototype 3D animations.

Because these development systems are 2D screen-based, the mismatch between the development and testing environment leads to context switching, loss of spatial information, and long iteration cycles. Immersive authoring in XR [28] enables developers to more accurately gauge the end user experience inside the headset while developing without context switching. Prior work, drawn from the [47] survey of literature, has explored immersive world

building and modeling tools like Tilt Brush [22], Microsoft’s Maquette [31], ScultUp [40], and many more [12, 24, 32, 33, 36]. However, robust logic generation is still challenging with these tools. Adding experience logic usually requires text-based scripting, which is slow and unreliable inside head-mounted display (HMD) environments due to the lack of efficient text input methods. One solution to this challenge is the use of visual programming languages, which allow users to create logic and control flow using visual elements rather than text-based scripts.

In one of the earliest immersive visual scripting systems, Steed et al. [43] proposed the use of visual dataflow to define object behavior in headset authoring environments, wherein users could draw wires between objects to pass data between them. Flowmatic [47] presents a drag-and-drop visual scripting interface in VR that allows users to create reactive behaviors while inside the headset. More recently, companies developing XR platforms like Meta and Microsoft have also begun developing their own immersive authoring environments, such as BuilderBot [1] and Frame[2].

However, existing immersive authoring systems often lack expressive power and suffer from complex visual interfaces that lack natural interaction methods. In contrast to existing authoring systems, AtomXR leverages natural language interaction in combination with natural physical interactions for immersive authoring, which allows users to directly and flexibly describe their target application and minimizes the need for users to map intention to text-based or visual scripting.

## 2.2 Low-Code Application Development

In recent years, a growing trend has emerged in favor of no-code and low-code platforms enabling users to build applications without requiring programming skills [14]. These platforms use various methods of program synthesis [44] and code generation to enable users to build and customize their applications using non-programming methods like natural language descriptions. This has the potential to greatly expand the number of people who are able to develop applications, increasing the diversity and creativity of the content available.

Low-code development for visual and 3D applications often employ template-based methods, in which users can procedurally generate terrains and even simple runtime behaviors using a set of parameters and rules. For example, ScriptEase allows users to generate interaction code by adapting existing game design patterns [20]. Other low-code development methods include programming by demonstration (PBD), which allows users to develop programs by demonstrating examples of desired behavior, and inductive program synthesis (IPS), which utilizes machine learning algorithms to generate programs from other example programs. For example, Rapido uses PBD to allow desktop and mobile-based AR prototyping [29].

There are also a number of tools and platforms that use NLP techniques to enable users to build applications using natural language descriptions. LLMs have demonstrated the potential to transform the way we generate, communicate, and implement ideas [18]. In particular, integration of LLMs into desktop software development processes has shown significant improvements in efficiency and accessibility [38, 39]. Some GPT-based systems such as Codex [17] have been explored for primitive XR world-building. For example,

Codex Pong uses Codex to generate code for a VR Pong game at runtime [41]. However, like many NLP-based generative programming techniques, their implementation is limited to a specific use case as it depends on the reliable generation of C# code, which becomes increasingly complex and error-prone with the size of the script. Our work avoids this common pitfall and minimizes the risk of generating inaccurate code by abstracting functionality to a higher-level language that is optimized for natural language based generation.

Generally, AtomXR focuses on addressing two key barriers in XR development process identified by Ashtari et al. [7]: 1) difficulty knowing where to start and 2) too many unknowns and changes in development, testing, and debugging. AtomXR was designed to both make XR development faster and easier for beginners to get started, and abstract away fragmented and moving parts in the complex development pipeline to reduce unknowns.

## 2.3 Natural User Interfaces

Natural language interfaces mediating human-computer interaction have been explored since the early days of interface-based computing. Notably in 1980, researchers at MIT demonstrated “Put-That-There”, a natural language and gesture interface that allowed users to control the position of simple shapes on a large screen [9]. From there, natural language interfaces have become increasingly complex and powerful, finding major use cases in voice assistant technologies [25], such as Apple’s Siri [8], Amazon’s Alexa, and Microsoft’s Cortona. These voice assistants perform anything from sending messages to searching the Internet to interfacing with smart home devices. However, natural language interfaces have been under-explored within XR development, largely because of the difficulty of accurately interpreting speech into actions in complex development contexts [21].

On the other hand, gesture-based interfaces have advanced alongside advancements in sensing and video processing technology, with advanced hand tracking and gesture recognition becoming standard within XR headsets. We combine natural language and gesture-based interfacing techniques to achieve accessibility, naturalness, and economy of expression within our application development system.

## 3 DESIGN GOALS

### 3.1 Current Development Process and Pain Points

The following scenario illustrates how Alex, an engineer who is unfamiliar with XR development, would attempt to build a simple coin collision game using Unity, a commonly used desktop software for XR development.

Alex first sets up the Unity project, navigating through project settings, downloading the appropriate XR packages, and adding scene configuration game objects to the scene so that it is compatible with his headset, the HoloLens 2. After the setup process, Alex adds a coin to the scene by searching for, downloading, and importing a model from an online database. As he is on a 2D screen, uses the mouse and keyboard to pan, rotate, and zoom to position the coin in the scene. To add functionality, Alex adds a new script component to the coin object. To implement a collision detection script, Alex

first reads Unity's documentation to understand different listeners and game object properties (rigid bodies, dynamic and kinematic types, the difference between variations of the collision listener function) before he can implement the right listener function. After he writes the script, he navigates back to the Unity interface to add and adjust components to the collision objects.

Alex then presses play to test out his script and uses the WASD keys and the mouse to move around in the simulator. After the simulator test works, Alex wants to make sure that this works inside the headset, so he sets up Unity to live stream into the headset. After pressing play in the headset, Alex realizes that the coin is too small when deployed to the headset (and not on a 2D screen), so he returns to the desktop to adjust the size of the coin and continues this cycle of development for the rest of the game.

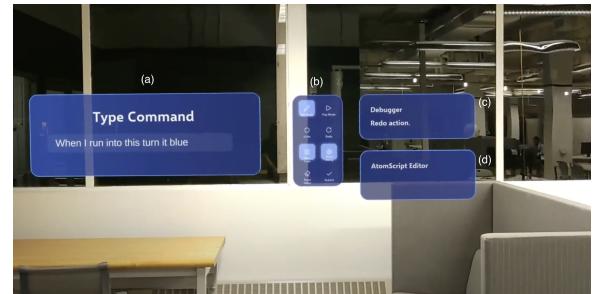
Based on this user journey, we identify three major pain points in the XR prototyping process:

- **Pain Point #1: Complex Workflow.** The current development process requires multiple steps, waiting periods, and environments across which developers must context switch both mentally and physically (e.g. setting up a new project in Unity, installing packages and finding assets online, programming in a separate IDE, testing on the target device), which contributes to a steep learning curve and general development friction.
- **Pain Point #2: Programming.** Users need to program and debug complex C# scripts, which is prohibitive for those without prior programming experience and can be inefficient for those with programming experience trying to rapidly prototype.
- **Pain Point #3: Development-Testing Environment Mismatch.** The current development workflow lacks support for immersive iteration where users can see and test what they are building in 3D as they are building it. Immersive world building tools like ShapesXR and FigminXR lack the ability to create game logic, separating the development process for design and functionality of applications.

## 3.2 Design Objectives

In this work, we explore how natural language combined with physical inputs can improve the XR prototyping experience by both addressing the above-mentioned pain points and opening up new possibilities for interaction. We developed the AtomXR system with the following three design goals:

- **D1: Natural interaction via multimodal intent recognition.** The system should allow users to directly and naturally express intention with natural language input and physical interactions to allow for ease of learning and ease of use.
- **D2: Easy logic design via programming abstraction and NLP.** The system should minimize intent-to-input translation by converting user-described logic into concise code (AtomScript) with transparency to facilitate error correction.
- **D3: Rapid immersive iteration via streamlined in-headset environment.** The system should enable users



**Figure 2: The four main components of AtomXR's user interface.** (a) The Type Command Box allows the user to input natural language commands; (b) the menu allows toggling between edit mode and play mode and performing global functionality; (c) the debugger Panel allows logging errors; (d) AtomScript Editor allows viewing and deleting the generated AtomScript code.

to iterate rapidly while inside the headset, allowing for immediate immersive testing without context switching.

## 4 ATOMXR SYSTEM

In this section, we describe in detail the features, architecture, and implementation of the AtomXR system.

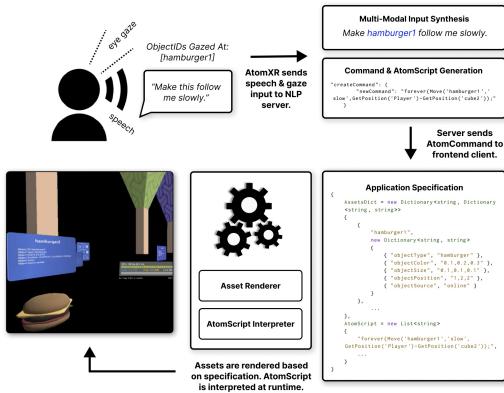
### 4.1 System Components & Architecture

The top-level primitive of AtomXR is a scene. Every scene contains objects and scripts.

**4.1.1 Objects.** Objects are 3D models in the scene. Objects are created based on natural language requests. The 3D models are sourced from a pool of built-in assets. If the requested object does not exist in the built-in asset pool, AtomXR searches an online database to find the closest related model. Each object has several attributes: size, orientation, location, and color. These attributes are represented in an object information panel that hovers above the object in the editor view. Each object also has methods for updating object attributes or deleting the object. These methods are triggered by natural language commands (e.g. *"Make this larger"*, *"Turn this blue"*) and physical interactions (grabbing an object and moving it, pressing the delete button, etc.).

**4.1.2 Scripts.** Scripts are pieces of AtomScript code that define the behavior of the scene and its objects. AtomScript code is generated using our NLP system, which translates natural language to AtomScript. Scripts are interpreted at runtime by the AtomScript interpreter built in Unity. Methods include but are not limited to creation and updating of variables, for and while loops, on start logic, if then logic, mathematical operations, playing sounds, runtime object attribute changes, runtime object creation and deletion, and specialized 3D application functions, such as collision detection and button press detection.

**4.1.3 Global Development Environment.** Beyond the elements within a scene, AtomXR also serves as a global development environment (Fig. 2), including:



**Figure 3:** After a user gives AtomXR a verbal command, we use the HoloLens built-in speech-to-text engine to convert this to text. This text, along with the user’s eye gaze targets while speaking, is sent to an online webserver. The webserver uses the GPT-3 API and semantic embeddings to synthesize the inputs and convert this text into a command that defines how to edit the application specification, including adding new AtomScript code. This application specification is then rendered and AtomScript code is executed on the user’s headset at runtime.

- Play and edit modes, which allow users to alternate between adding functionality in an information-rich display of the scene and testing how the final application will behave.
- Ability to view and delete generated scripts.
- Undo and redo functions, which allow users to revert to a previous or future state.
- Reset function, which allows users to erase the current scene and restart the development process.
- Save function, which allows users to save their application specification to a cloud database.
- Hybrid eye gaze, gesture, and natural language interaction methods to create, manipulate, and reference objects.

**4.1.4 System architecture.** The AtomXR system, diagrammed in Fig. 3, consists of a frontend development environment and a backend web server. The immersive frontend application supports D3 and was built in Unity and deployed to Microsoft’s HoloLens 2. We use the built-in text-to-speech functionality of the HoloLens keyboard to transcribe what the user is saying, and use the eye-tracking and hand-tracking data from Microsoft’s Mixed Reality Toolkit (MRTK) to enable our eye-gaze and touch interactions. This multimodal input data is then sent to a persistent backend web server runs on the cloud via Amazon Web Services (AWS). The server NLP system, described in detail in Sec. 4.5, synthesizes the several input streams and returns a user intention interpretation that is then used to update the specification of the application. The specification of an application follows a nested json format, an example of which is shown for a simple chase game in Fig. 3. This specification is used to render assets and execute AtomScript code at runtime.

Below, we described the implementation details for key components of AtomXR, including the user interface (Sec. 4.2), interaction scheme (Sec. 4.3), AtomScript (Sec. 4.4.1), and NLP system (Sec. 4.5).

## 4.2 User Interface

**4.2.1 Type Command Box.** The Type Command box (Fig. 2a) allows users to input natural language requests to build their application. Pressing on the input box will open up a virtual keyboard. Here, the user can either use their hands to manually type into this virtual keyboard, or they can press the microphone button on the keyboard to speak their request. We include both input methods but recommend the microphone input as typing using a virtual keyboard in augmented reality is less efficient [19].

**4.2.2 Menu.** The first row of the menu panel (Fig. 2b) contains a toggle between AtomXR’s two modes: edit and play mode. In edit mode, users can use natural language to create their application, view the debugger (Fig. 2c) and the current AtomScript code (Fig. 2d), and manipulate the properties of objects. In play mode, users can test the application logic from the end user’s perspective. Users switch between edit and play modes often when developing their applications. In the second row of the menu, users can undo and redo actions that have been taken in edit mode. In the third row, users can toggle the visibility of the Debugger panel (Fig. 2c) and AtomScript Editor (Fig. 2d). In the final row, users can press the reset button to restart their development process, or the submit button to save their game to a persistent cloud database.

**4.2.3 Debugger Panel.** (Fig. 2c) The debugger panel is used as a feedback mechanism to update users on the system status (e.g. when the server is processing requests, when errors arise, etc.).

**4.2.4 AtomScript Editor.** The AtomScript Editor (Fig. 2d) shows all of the currently written AtomScript code. Each segment of code will appear as a separate block. Users can delete code by pressing the delete button on the code block. Due to the difficulties of typing in AR, we do not allow users to edit code but instead encourage users to just delete the code and re-try their voice command. For future work, we expect to trial more manipulable code representations rather than just plain text, such as a block-based system similar to MIT’s Scratch platform.

## 4.3 AtomXR Interaction Design

To support D1 and enable D3, we developed an interaction scheme integrates several natural modalities of user input (natural language, eye gaze, and physical interactions) to allow users to directly express intention without being restricted to a single, pre-defined method of communicating with the system.

**4.3.1 Natural Language Interaction.** The primary method for a user to make requests to the system is through natural language, inputted either through dictation or typing. Voice commands are often faster and more natural for conveying intent than the standard keyboard. However, speech-to-text accuracy issues sometimes makes it slower, so the standard keyboard is also available to the user. Users can speak to the system conversationally as if they were speaking to an assistant. Users can create (e.g., “Create a small red cube”), update (e.g., “Make the cube blue”), and delete objects (e.g.,

*"Delete this cube"*) and specify logic using natural language (e.g., *"If I run into this cube, turn it green"* or *"Make this cube follow me slowly"*). To create a simple chase game, the game designer could say *"Give me a turtle"*, *"Make the turtle chase me"*, *"Make the turtle move faster if I start moving faster"*.

This natural language interaction accommodates the many ways a user may choose to express the same intention. By using LLMs and semantic embeddings to convert natural language into code, variations in language with the same intent yield the same code. Users can express ideas without matching a specific syntax. For example, to create an object into the scene, *"create a cube"*, *"generate a 3-dimensional square"*, *"put a cube into the scene"*, and *"can you make me a box"* all result in the same cube being placed in the scene. This flexibility to linguistic variation makes our natural language interaction method syntax agnostic, where users can directly express their thoughts instead of learning to map their thoughts into a specific accepted programming syntax. The direct expression of thought allows users to focus on creating the substance of an application rather than transcribing substance into specification. The NLP backend that supports these interactions is described in detail in 4.5.

**4.3.2 Eye Gaze Interaction.** Eye gaze interaction, enabled by the eye-tracking capabilities of the HoloLens, provides supplemental information to support natural language interaction, allowing the user to leave ambiguities in their language by predicting intent from eye-tracking. Specifically, eye gaze helps users reference objects when giving natural language commands to specify which objects to apply the command to. Instead of referring to objects by their object ID, a user can use a demonstrative pronoun while gazing at the object- for example, instead of saying *"Make cube1 orange"*, a user can say, *"Make this orange"* while gazing at the cube. This works for any kind of reference substitution, including updating objects (e.g., *"Make this larger"*), deleting objects (e.g., *"Delete that"*), and defining logic involving one or multiple objects (e.g., *"Make this (gaze at object 1) revolve around that (gaze at object 2)"*). The system provides validation for the user by darkening the color of objects when the user gazes at them, indicating that it has detected their reference to that object. This interaction workflow mimics what people naturally do when communicating with others, again reducing the effort required from the user to match their natural expressions of intention with system input methods.

**4.3.3 Physical Manipulation.** We implemented hand-gestural interactions based on the hand-tracking capabilities of the HoloLens to complement the natural language and eye-gaze interactions for physical manipulation. Users can easily resize, reposition, and delete virtual objects through intuitive hand gestures like pinching and dragging. This in-headset object manipulation enables a real-time, 3D perspective view of the real application, which offers a significant advantage over traditional 2D interfaces that lack depth and scale awareness. Users can also interact with virtual buttons and control menu panels with simple hand movements.

## 4.4 AtomScript

To support D2, we designed AtomScript, a high-level programming language that AtomXR generates to define game logic. AtomScript

had two design goals. First, the language should implement the core functionality necessary for developing XR applications, including both fundamental programming constructs and functionality specific to 3D and XR applications. Second, the language's syntax should be as semantically close to human thought as possible, such that it is both easier for the human user to understand and conducive to generation from natural language via LLM.

AtomScript abstracts away the low-level complexity of the underlying Unity C# on which it was built. Directly generating C# code from natural language has been found to be prone to error and difficult to scale when asked to generate increasingly complex code [41]. This is due to the large number of syntax and usage rules that must be followed to generate working code. AtomScript improves the quality of converting natural language to code by following a simpler syntax. While the current version of AtomScript is much less capable than Unity C#, for rapid prototyping of XR experiences, AtomScript consolidates the largely unused optionality that other game engines like Unity and Unreal provide. Through this design, LLMs can effectively generate AtomScript code more accurately and humans can understand it better. The grammar of AtomScript is detailed in 11.2

Below, we introduce AtomScript's core features and properties.

**4.4.1 AtomScript Core Features.** **Basic elements.** AtomScript has three fundamental types: strings, numbers, and arrays. Variables can be declared and assigned to each of these data types. These data can be passed into AtomScript's built-in functions.

**Built-in functions.** To align with the capabilities of most 3D game engines, AtomScript supports essential operations like object creation, deletion, and manipulation. It also features an event-handling system and separate functions for code that initializes at game start and code that runs continuously each frame.

Based on established game design patterns, AtomScript provides several built-in listeners that allow the user to perform certain operations on specific game events, including *onStart*, which runs at the start of the game; *forever*, which runs every frame; *onButtonPress*, which triggers when a user presses a "button" object; *onCollision*, which triggers when two game objects collide. To allow rapid prototyping for common XR applications, AtomScript also features a number of built-in functions. These functions are the "atomic" building blocks of more complex game interactions. These include functions like *Move*, *ChangeColor*, *GetPosition*, and *PlaySound*, among several other built-in functions.

We demonstrated how users use AtomScript to prototype realistic applications in Sec. 5.

**4.4.2 AtomScript Properties.** The design of AtomScript provides the following properties as a prototyping language in comparison to traditional Unity C#.

**Completely text-based:** To enhance the interpretability of prototyping scripts, the generation of AtomScript is purely text-based. This allows AtomScript to be stored completely as human-comprehensible text and could be easily generated by AI. Many interactions in game engines like Unity and Unreal Engine are carried out using a UI rather than code, but using text as the authoring interface allows the entire game space to be within the output space of state-of-the-art generative text AI.

**Simple, high-level interface:** To enhance prototyping for XR applications, AtomScript abstracts away complex functions to improve usability and learnability by non-programmers. AtomXR prioritizes user experiences over robustness with a focus on rapid prototyping. For example, we optimized routine operations in XR experiences like object movement for conceptual simplicity to enhance user experience. As shown in Fig. 4, to move an object in a straight line, a developer would write the following in Unity C# (left), to replaces the target\_obj from its previous position by 2 in the z-axis.

```
Unity Code
void Update() {
    GameObject target_obj = GameObject.Find("target_obj");
    target_obj.transform.position = new Vector3(0,0,2) * Time.deltaTime;
}

AtomScript
forever {
    Move('target_obj', 'slow', [0, 0, 2]);
}
```

**Figure 4: Scripts for moving an object in Unity and AtomScript.**

While such low-level abstraction may be suitable for many complex, non-XR projects, it's unnecessary for most XR applications and can even be difficult for novice developers. The simplified expression of AtomScript (right) allows the user to evaluate the correctness of the AI-generated code blocks even if they lack the expertise to write the code themselves.

To further improve understandability, we made certain design decisions like naming the position command as "GetPosition" instead of "transform.position" and the vector declaration as [0, 0, 2] instead of Vector3(0, 0, 2).

**Limitations.** Although AtomScript is engineered for simplicity and ease of use, it does have its limitations compared to Unity C#. For instance, it lacks the ability for users to create custom functions and doesn't offer fine-grained control over interactions such as collision detection and object attributes.

These features are often crucial in applications requiring detailed control. However, it's worth noting that while Unity is tailored for multi-platform, production-level gaming, AtomScript is purposefully designed with a narrower focus: to facilitate rapid XR prototyping and development.

## 4.5 Natural Language Processing

To support D1 and D2, we developed an NLP system that combines LLMs with semantic embeddings to interpret user intent from multimodal input.

**4.5.1 Intention Recognition.** Each time a user submits a request, their natural language input is sent to the NLP server along with information about their eye gaze (see first process arrow in Fig. 3). Few-shot prompted GPT-3 is then used to generate an AtomCommand, which is a JSON object specifying a modification of the game state.<sup>1</sup>. Example syntax of AtomCommands for creating logic, and creating, updating, deleting objects are shown below.

These AtomCommands are sent to the Unity frontend application and executed (see second process arrow in Fig. 3). Using LLMs to convert natural language (e.g. "create a red ball") into an AtomCommand is generally accurate when classifying the type of command (i.e. logic creation and object deletion/creation/updates).

<sup>1</sup>AtomXR was built in 07/2022, predating GPT-4

```
(a) Create Objects
createCommand = {
    "properties": {
        "objectType": {"type": "string"},
        "objectColor": {"type": "string"},
        "objectName": {"type": "string"},
        "objectPosition": {"type": "string"},
        "objectSource": {"enum": ["primitive", "prefab"]}
    },
    "required": ["objectType", "objectColor", "objectName", "objectPosition", "objectSource"]
}

(b) Update Objects
updateObj = {
    "properties": {
        "objToUpdate": {"type": "string"},
        "propertyToUpdate": {"enum": ["objectColor"]},
        "newPropertyValue": {"type": "string"}
    },
    "required": ["objToUpdate", "propertyToUpdate", "newPropertyValue"]
}

(c) Delete Objects
deleteObj = {
    "properties": {
        "objToDelete": {"type": "string"}
    },
    "required": ["objToDelete"]
}

(d) Create Logic
createCommand = {
    "properties": {
        "newCommand": {"type": "string"}
    },
    "required": ["newCommand"]
}
```

**Figure 5: The expected format of an AtomCommand. AtomCommands are generated by GPT-3 and specify the AtomXR game state.**

The key challenge in the NLP process is generating the AtomScript code that matches the intent of the author.

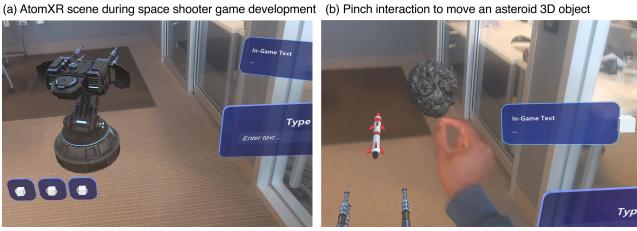
**4.5.2 Generating AtomScript.** To generate AtomScript, we prompted GPT-3 with example pairs of natural language and AtomScript in the following format:

```
1 SPEECH:Make the octopus5 disappear and play a noise after 4 seconds
2 ATOMCOMMAND:{ "createCommand":{ "newCommand": "currTime=TimeSinceStart()";
3 forever{if(TimeSinceStart()-currTime >= 4) {Disappear('octopus5');
4 Play('noise');}}}}
4 ###
5 SPEECH:When the variable building is equal to 3, then make the
6 table3 play a noise
7 ATOMCOMMAND:{ "createCommand":{ "newCommand": "forever{if(building==3){
8 PlaySound('table3')}"}}
7 ###
8 ...
```

We took a trial and error approach to prompt engineering, correcting for edge case errors as they arose.

**4.5.3 Approximation Using Embeddings.** When a user requests an object, they may use a number of different names to refer to the same kind of object. For example, a user may ask for a "red apple", a "fresh apple", or simply an "apple". AtomXR contains a pre-loaded database of 3D models, including built-in Unity primitives (cube, sphere, capsule, etc.) and additional 3D models we curated (fruits, collectible items, household items, parts of nature, etc.). If the name of the object requested by the user does not directly match the name of an object in the AtomXR database, we use a semantic embedding calculation (based on TensorFlow's universal sentence encoder [16]) to determine whether there exists a similar object in the database or whether we should query the object from an online database. Specifically, if the embedding vector of the object name requested by the user is close enough to that of an object name within our database ( $> 0.75$  cosine similarity), we use the model from our database. If not, we query the online Sketchfab database to find relevant models. This method provides approximations of user intent that allow the user flexibility in describing desired objects.

**4.5.4 Referencing Objects.** Our eye gaze referencing system that allows users to use ambiguous language when referring to objects ("Make **this** blue", "Delete **that**", etc.) is enabled by a few-shot prompted GPT-3 call that replaces ambiguous demonstrative pronouns with the objectID of the object the user gazes at. While



**Figure 6: A space shooter game developed using AtomXR.** The in-headset photos show (a) the game scene during development containing AtomXR Type Command Box, the turret, and 3 control buttons, and (b) Alex using simple and intuitive pinch gestures to an asteroid 3D object.

this processing could also be implemented through simple part-of-speech (POS) tagging, we used an LLM method to avoid edge case errors. The few-shot prompt is formatted as follows:

```
Replace the appropriate words in the string:  
STRING: If I touch this blue box then change this box's  
color.  
OBJECT LIST: box12, box23  
NEW STRING: If I touch box12 then change the color of box23.  
###  
Replace the appropriate words in the string:  
STRING: If this sphere runs into that cylinder then increase  
the scoreboard variable by one.  
OBJECT LIST: sphere10, cylinder38  
NEW STRING: If sphere10 runs into cylinder38 then increase  
the scoreboard variable by one.  
###  
...
```

## 5 USAGE SCENARIOS

To demonstrate the expressiveness of AtomXR concretely, we present two example experiences authored by Alex using the AtomXR system. Alex serves as a representative adult with limited coding skills and minimal familiarity with AR environments, yet has a keen interest in prototyping XR experiences.

### 5.1 Space Shooter

The following scenario illustrates how Alex can use AtomXR to quickly build an augmented reality experience using natural language and physical interactions inside his headset. Alex wants to build a complex space shooter game where the player is inside a spaceship, with asteroids flying towards him. The player should be able to press 2 buttons to move the turret left or right, and another button to fire at the asteroids. The game starts with 5 asteroids hurtling toward the player, and every asteroid that is destroyed spawns 2 more asteroids, making the game increasingly hard. If the player surpasses 20 score, they win the game.

To complete the game, Alex prompted AtomXR step-by-step, as shown in Script 7. We described each step sequentially while referring to the prompts and corresponding code in Script 7.

```
[1] "Give me a futuristic white spaceship with large windows"  
[White spaceship created in the AtomXR scene]  
[2] "When the game starts, play some 8 bit music"  
onStart(  
    PlaySound("8bit");  
)  
[3] "Make it so when I press button1, move the turret  
left slowly, and vice versa for button3"  
onButtonPress("button1")<  
    Move("turret1", "slow", [-1, 0, 0]);  
>  
onButtonPress("button3"><  
    Move("turret1", "slow", [1, 0, 0]);  
>  
[4] "Make it so when I press button2, the rocket shoots  
out of the turret"  
onButtonPress("button2"><  
    Move("rocket1", GetPosition("turret"));  
    Move("rocket", "medium", [0, 0, 0.4]);  
>  
[5] Instantiates a limited amount of lives in the game  
Result: onStart(  
    lives = 3;  
)  
forever {  
    if (lives < 0) {  
        Disappear("spaceship");  
        Disappear("turret");  
        PlaySound("scary_sound");  
    }  
}[6] "When the game starts, spawn 5 asteroids. When the rocket  
runs into any of the asteroids, make that asteroid disappear,  
play an explosion sound, and spawn another asteroid"  
Result: onStart(  
    numAsteroids = 5;  
    for (i = 0; i < numAsteroids; i++) {  
        x = (i * 100) + 190;  
        y = (i * 7 + 1) - 25;  
        z = ((i * 3) * 11 + 1) - 125;  
        Create("asteroid" + i, "asteroid", "8,0,0", "1,1,1", [x, y, z], "online");  
    }  
)[7] "Loop through all the asteroids to check if any hit the  
spaceship, and if so, make it disappear, spawn a new asteroid  
far away, and decrease my lives by 1."  
Result:  
for (i = 1; i < numAsteroids; i++) {  
    onCollision("spaceship", "asteroid" + i) {  
        Disappear("asteroid" + i);  
        Create("asteroid" + i, "asteroid", "8,0,0", "1,1,1", [100,0,0], "online");  
        Create("asteroid" + numAsteroids, "asteroid", "8,0,0", "1,1,1", [100,0,0], "online");  
    }  
}  
[8] "Can you make each asteroid continuously move toward the  
player. Each asteroid should rotate in a random direction."  
Result:  
forever {  
    for (i = 0; i < numAsteroids; i++) {  
        Move("asteroid" + i, "fast", GetPosition("spaceship"));  
        Rotate("asteroid" + i, 1 + i * 10, 1 + 20, 1 + 30);  
    }  
}  
[9] "Can you make it so if my score reaches 30 and I still have lives,  
spawn a gold trophy and play a happy song, and write you win!"  
Result: onStart {  
    has_won = false;  
}  
forever {  
    if (score >= 30 && has_won && lives > 0) {  
        Create("trophy", "Gold Trophy", "0,0,0", "1,1,1", [0,0,0], "online");  
        PlaySound("8bit happy");  
        has_won = true;  
        Write("You win!");  
    }  
}
```

**Figure 7: The prompts spoken by Alex and the AtomScript generated while creating the space shooter game.**

Alex starts prototyping with AtomXR by putting on his headset, opening the AtomXR application, and dictating into the command box, “Give me a futuristic white spaceship with large windows” (Fig. 7[1]). On the backend, AtomXR uses the SketchFab API to find a spaceship model matching the query. From there, Alex moves and resizes the spaceship by grabbing and dragging it around with his hands, so that the spaceship is large enough to fit him inside. Alex then asks for a “Sci-fi turret”, a “rocket” for firing, and 3 buttons for moving and firing the turret. These items appear into his AR world as shown in Fig. 6a.

Alex asks for music to be added to the game (Fig. 7[2]). Alex can see the code that was generated in the AtomScript viewer/editor.

Now, with all the components in the world and music, he starts adding turret logic to the game, which allows the user to move the rockets left and right, as well as shoot with the buttons (Fig. 7 [3, 4]). Alex then goes into Play Mode to verify that the AtomXR-generated game works. When he enters play mode, the music plays. When he presses the left and right buttons, the turret moves accordingly. When he presses the middle button, the rocket fires from the turret, as expected.

Alex decides to add a new mechanic, where he wants the user to only have a limited amount of lives in the game. He creates a system where if you lose 3 lives, then the game is over (Fig. 7 [5]). He wants the number of asteroids at the start to be 5, and when the rocket hits an asteroid, increment the number of asteroids by 1, destroy the hit asteroid, and spawn 2 more in random locations (Fig. 7 [6, 7]). After testing this out in play mode, Alex notices the game is too easy because the asteroids are just standing in place. He asks for the asteroids to chase him (Fig. 7 [8]). Alex again goes into Play Mode to test the functionality. Fig. 6 shows the objects in the current scene. Wanting the game to have a satisfying conclusion, Alex adds an end-state where the user can win (Fig. 7 [9]). Entering Play Mode,

Alex plays the completed game, and sees that everything is working together.

## 5.2 Chase Game

The following scenario illustrates how an interactive chase game can be built. This was the game we asked participants in user studies to build in order to compare the effectiveness of AtomXR to other systems. In this game, the player works to collect a cherry, which respawns in a new location every time the user runs into it. The player is being chased by a watermelon. The game also integrates several audio interactions: background music is played continuously, when the player collects the cherry, a coin collection sound is played, and when the watermelon runs into the player, a scary sound is played, indicating failure.

```
[1] "When the game starts, play some piano music"
  onstart {
    PlaySound("No1");
  }
[2] "and can you make cherry one rotate in place"
  forever{
    Rotate("cherry1", 0, 1, 0);
  }
[3] "When I run into cherry1, move it away a little and
  play a coin collection sound"
  onCollision<"Player", "cherry1> {
    MoveTo("cherry1", GetPosition("Player") + [1.0, 0.0, 0.0]);
    PlaySound("coin collect");
  }
[4] "make the watermelon follow me slowly"
  forever {
    Move("watermelon2", 'slow', GetPosition("Player")
      - GetPosition("watermelon2"));
  }
[5] "When the watermelon runs into me, play a scary sound"
  onCollision<"Player", "watermelon2"> {
    PlaySound("scary");
  }
```

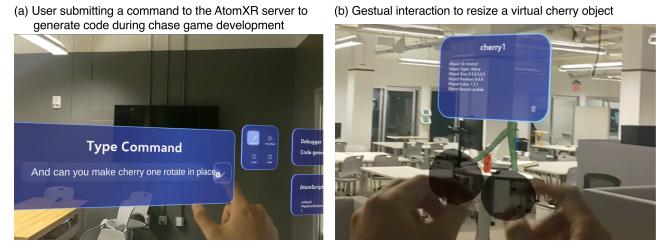
**Figure 8: The prompts spoken by Alex and the AtomScript generated while creating the chase game.**

The user starts by stating adding music to the game (Fig. 8 [1]). The user then states: "Give me a small cherry", and the cherry spawns in their scene. It is still a bit too big and isn't in the right spot, so Alex uses his hands to grab, resize, and move the cherry to the right place, as seen in Fig. 9b. He then says "and can you make this cherry rotate in place" (Fig. 9a). Because Alex was looking at the cherry while saying "this cherry", AtomXR's eye-gaze system understands that he's referring to the recently created cherry. Alex enters play mode and finds that the cherry is behaving as expected. He then moves onto more complex functionality, first asking for the cherry to move away and play a sound every time he collects it (Fig. 8 [3]). With the cherry collection functionality completed, the user creates a watermelon that chases the user around as they collect cherries (Fig. 8 [4]).

Finally, the user encodes the end-game where the watermelon plays a scary sound effect on catching the player. ([5]) Testing the game, Alex sees the watermelon running after him slowly, and the cherry runs away every time it is collected, with coin collection sounds, scary sounds, and background music playing.

## 6 STUDY #1: ATOMXR VS. CURRENT WORKFLOW

To understand how well our system supports XR prototyping tasks in comparison to current development systems, we conducted a study in which participants were asked to complete a series of development tasks using AtomXR (in both desktop and headset variations) and the traditional Unity-based development system. Specifically, the study aimed to explore the following research questions:



**Figure 9: (a)** The user submitted a command to the AtomXR server to generate code during chase game development. **(b)** The user used hand interactions to resize a cherry virtual object. Above the cherry is an info-card about its properties, including ID number, object type, object size, position, and the source of where the 3D object was spawned from.

- **RQ1:** How does AtomXR compare to traditional development methods in terms of learnability, ease of use, and efficiency for prototyping interactive XR applications?
- **RQ2:** How does AtomXR affect the user's subjective experience of difficulty and satisfaction during the development process?

We performed analyses on 1) completion metrics (evaluated by experimenters, including the number of tasks completed, time taken per task), and 2) user experience metrics (evaluated by participants, including ease-of-use, learning curve, enjoyment, physical comfort).

## 6.1 Study Design

We designed a hybrid within-subjects and across-subjects study with three conditions, in which participants were given 20 minutes to complete a series of development tasks using 1) the Unity development system, 2) AtomXR running on desktop, and 3) AtomXR running inside the headset. Each participant experienced the Unity condition and one of the AtomXR conditions, with condition assignment and order counterbalanced to control for learning effects. We included the desktop version of AtomXR in order to disentangle the effects of interfacing with the headset from the effects of the use of natural language on the development experience. Data was collected via two post-condition and one post-experiment survey and interview.

## 6.2 Participants

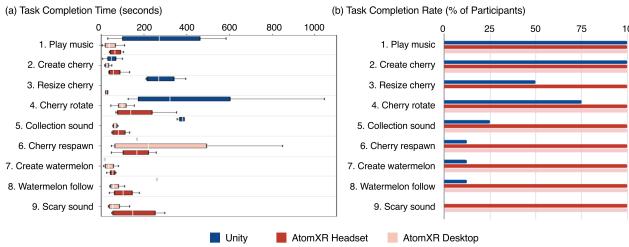
We recruited 8 participants (6 female and 2 male) from university and public interest group channels, selecting to diversify experiences and attitudes toward technology. On average, participants held positive attitudes toward new technologies (mean = 4.25, SD = 0.71, where 1 is very negative and 5 is very positive). On a scale from 1 (layman) to 5 (expert), participants on average reported a 3.13 level of prior programming experience (SD = 1.13), a 2.13 level of prior game development experience (SD = 0.83), and a 1.75 level of prior experience working with AR/VR (SD = 0.71).

## 6.3 Procedure

Participants were first given an introduction and overview of the experiment, followed by a consent form. Participants then received

**Table 1: Experiment Tasks**

Task #	Task Description	Task Category
1	When the game begins, play some piano music.	Audio & Start Logic
2	Create a cherry.	Object Creation
3	Change the size of the cherry to make it realistic.	Object Property Update
4	Make the cherry rotate in place.	Object Behavior
5	Play a collection sound effect when the player hits the cherry.	Audio & Collision Logic
6	Move the cherry forward a little when the player hits it.	Movement & Collision Logic
7	Create a watermelon.	Object Creation
8	Make the watermelon chase the player slowly.	Object Behavior
9	Play a scary sound when the watermelon runs into the player.	Audio & Collision Logic



**Figure 10: Study #1 results showing (a) task completion time and (b) task completion rate for three conditions (Unity, AtomXR Desktop, and AtomXR Headset) across all tasks<sup>2</sup>.**

a 10-minute guided walkthrough of how to use the first development system in their assigned treatment group. This walkthrough covered the basics of the development system and demonstrated how to create objects, update their size and position, track collisions, and play audio. Participants were allowed to ask questions freely. Following the tutorial, participants were given 20 minutes to complete the experiment task using the first system.

We designed an integrative task in which participants were asked to complete a series of smaller tasks to build an AR game involving collecting items while evading an enemy. The task was designed to cover a breadth of different types of development tasks, from object creation to logic implementation. Table 1 details the task descriptions and categories.

We recorded the time participants needed to complete each task. After the time expired, they answered survey questions about their task experience. They then repeated this process for the second system in their assigned group, including another walkthrough and post-task survey. Finally, participants completed a post-experiment survey with multiple choice and open-ended questions about their overall experience during the full experiment, evaluations of each system, and preferences.

#### 6.4 User Performance

In the AtomXR Headset and AtomXR Desktop conditions, all 8 participants successfully completed 100% of the tasks, with an average of 6.4 minutes to spare. By contrast, none of the participants were

<sup>2</sup>When interpreting task-by-task graphs, note that tasks were given in sequential order, so if a participant was stuck on one task, they would be unable to complete subsequent tasks. The data shown in task completion graphs are from participants who completed the task, so time variances may not be directly compared across tasks.

able to complete all tasks in the Unity control condition, and on average were only able to complete 46% of tasks. Moreover, the average task completion time was 98.25 seconds with AtomXR Headset, 93.34 seconds with AtomXR Desktop, and 243.49 seconds with Unity.

Fig. 10 shows the average task completion times across tasks, and Fig. 11a shows the percentage of participants able to complete each task.<sup>3</sup> From this breakdown, we observe that tasks requiring implementation of logic or manipulation of 3D objects are performed much faster using AtomXR systems compared to Unity.<sup>4</sup> On the other hand, simple object creation tasks (e.g., Task #2 for creating the cherry and Task #7 for creating the watermelon) are faster with Unity's drag-and-drop interface. However, in workflows in which users need to search online to find models outside of built-in models, AtomXR's automated search process that queries models from an online database based on semantic similarity to the user's request may provide value.

#### 6.5 User Experience & Preference

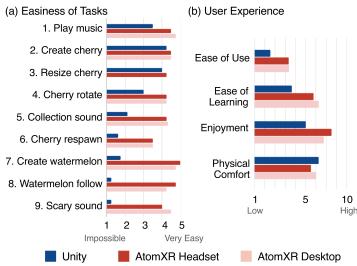
Beyond objective performance improvements, users also experienced the tasks as being easier to implement using AtomXR compared to Unity, as shown in Fig. 11a. Reflecting on their overall experience with both Unity and AtomXR systems, participants found AtomXR systems easier to use and easier to learn, and also found the experience more enjoyable, albeit slightly less physically comfortable, as depicted in Fig. 11b. Some participants felt intimidated by Unity, with P6 noting that Unity "*trigger[ed] [in] me the fear of learning coding*", and others felt some uncertainty with AtomXR, noting they "*feel less confident about troubleshooting mistakes while prototyping*" (P1).

In terms of user preferences, participants generally felt that the relative utility of each system depends on the context, with P8 noting that "*there's an accessibility/extensibility trade-off*". Participants often found AtomXR "*fun and easy*" (P4), and noted that they "*would like to use AtomXR to get prototype and getting start with design games, but Unity for further development*" (P6).

Based on these results, we conducted a follow-up study to further explore the design implications of each component (scripting

<sup>3</sup>Task #3 was omitted for the AtomXR Desktop condition due to the inability to use holographic remoting with AtomXR Desktop.

<sup>4</sup>One participant ran into a bug in Task #6, resulting in anomalous data for AtomXR Desktop in that task.



**Figure 11: Study #1 user ratings comparing Unity, AtomXR Headset, and AtomXR Desktop in the easiness of tasks and user experiences.**

abstraction, natural language based scripting, immersive environment).

## 7 STUDY #2: COMPONENTS OF MULTIMODAL PROTOTYPING EXPERIENCE

This study aimed to dive into each component of the system and investigate its impact on user performance and experience in order to better understand design opportunities for the future of XR development. Specifically, this study explored the following research questions:

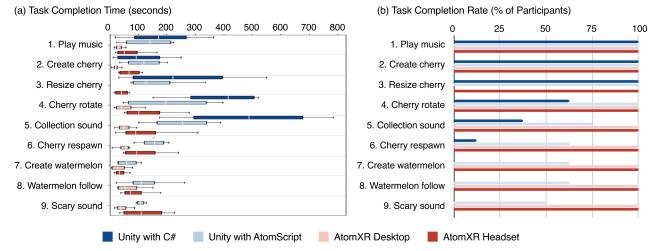
- **RQ3:** How do users interact with and value each feature of AtomXR, including AtomScript, natural language interaction, and immersive authoring?
- **RQ4:** How do variations of the system with different features compare in terms of learnability, ease of use, and efficiency for prototyping interactive XR applications?

### 7.1 Study Design

Similar to the first study, the main study was also a hybrid within-subjects and across-subjects study, in which participants again were given 20 minutes to complete a series of development tasks using two different development systems. In this study, however, we focused on isolating the interaction dynamics and contributions of each feature of the AtomXR system. For this purpose, we designed four conditions with different combinations of abstraction, natural language scripting, and immersive authoring:

- *Unity with C#* - Participants use the traditional Unity development system with their standard C# scripting system.
- *AtomScript* - Participants use the traditional Unity development environment but are able to script using the high-level AtomScript language.
- *AtomXR Desktop* - Participants use the AtomXR system in a desktop environment, in which they build applications on a 2D screen by giving natural language requests.
- *AtomXR Headset* - Participants use the AtomXR system in a headset environment, in which they can experience and edit applications in a 3D AR environment using physical manipulation, eye gaze-assisted object referencing, and natural language requests.

We designed four treatments, each containing two of these four conditions, to investigate the effect of each new feature. Four participants experienced each treatment, and the ordering of conditions



**Figure 12: Study #2 results showing (a) task completion time and (b) task completion rate (% of Participants) for four experimental conditions (Unity with C#, Unity with AtomScript, AtomXR Desktop, and AtomXR Headset) across all tasks.**

within treatments was counterbalanced to minimize any learning effect. The treatments were as follows:

- *Unity with C# vs Unity with AtomScript* - This comparison investigates the effect of allowing users to create logic using a high-level programming language that abstracts away the low-level complexities of the standard C# system.
- *AtomScript vs AtomXR Desktop* - This comparison investigates the effect of allowing users to create logic and objects using natural language instead of writing code and using a graphical user interface.
- *AtomXR Desktop vs AtomXR Headset* - This comparison investigates the effect of allowing users to author in an immersive, 3D environment instead of on a 2D interface.
- *AtomXR Headset vs Unity* - This comparison investigates the integrative effects of all features of AtomXR.

### 7.2 Participants & Procedure

We recruited 16 participants (4 female, 12 male) from university and public interest group channels, selecting for varied levels of prior experiences and perspectives to create a balanced pool. On average, participants held positive attitudes toward new technologies (mean = 4.81, SD = 0.40). On a scale from 1 (layman) to 5 (expert), participants on average reported a 3.63 level of prior programming experience (SD = 1.20), a 2.19 level of prior game development experience (SD = 1.11), and a 1.75 level of prior experience working with AR/VR (SD = 1.00).

The procedure for this study followed the same structure as the procedure for the first study, with the only differences being in the systems participants used and the questionnaires they were given. In this study, the surveys given were more in-depth, and included open-ended questions asking about user experiences with each component of the system (high-level language, natural language logic creation, immersive authoring, etc.).

## 8 RESULTS

Combining the quantitative analyses and ratings with qualitative responses from participants' open-ended answers and interviews, we synthesized the following insights on user experiences, perception, and interaction with different features of the system.

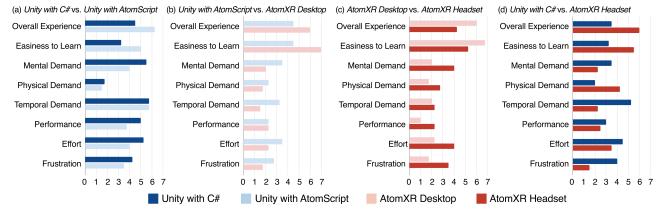
## 8.1 AtomScript enables improved task completion and development experience compared to traditional C# (Unity with C# vs. Unity with AtomScript)

Fig. 12 shows the average task completion times across tasks (Fig. 12a) and the percentage of participants able to complete each task (Fig. 12b). Within the allotted time frame, participants were able to complete on average 45% of the tasks in the *Unity with C#* condition and 79% of tasks in *Unity with AtomScript*. They completed tasks on average 1.48 times as fast using *Unity with AtomScript* compared to *Unity with C#*, suggesting a significant barrier and point of friction to early stage application prototyping is the complexity of using low-level C# code.

From Fig. 12a, we observe that *Unity with AtomScript* provides general speed improvements over *Unity with C#* for all available tasks except for the object creation task (i.e., Task #2 for creating the cherry). In particular, AtomScript provided a large speed improvement in Task #4 (make the cherry rotate) and Task #5 (make the cherry play a collection sound when the player runs into it). This could potentially be explained by the large reductions in syntax and implementation complexity of rotation and collision detection logic from C# to AtomScript. This highlights the potential advantages of high-level functional abstractions in accelerating and simplifying early-stage prototyping for users. Specifically, AtomScript offers significant benefits in scenarios where traditional low-level implementations are syntactically complex or require multiple steps.

Comparing the two conditions, we also found experience improvements in ease of learning, overall user experience, and nearly all dimensions of evaluation measured by the NASA TLX set [23], as shown in Fig. 13a. The most notable improvements were in the overall experience and the easiness to learn the system (39% and 54% improvement of *Unity with AtomScript* over *Unity with C#*, respectively). All four participants in this treatment found *Unity with AtomScript* to be easier to use, and preferred it over *Unity with C#* for prototyping. Three out of four participants found *Unity with AtomScript* easier to learn, while one participant thought *Unity with C#* was easier because there was more online documentation that could be copied and pasted in contrast to AtomScript (P2).

In general, participants appreciated the reduction in complexity of the code they needed to write, with P2 observing, AtomScript "allowed me to significantly reduce the amount of code I had to physically type out" and "reduced the number of errors that arose during compilation/runtime" and P4 saying that the "complicated syntax of C# made it difficult to pick up and use" in comparison to AtomScript. Regarding the impact of AtomScript on developer experiences, P6 noted that "reduced stress level leads to more creativity". In general, when discussing the trade-offs of AtomScript and C#, participants noted that AtomScript lacked the community support that an established language like C# had, while C# lacked beginner friendliness and would at times be over complicated for the purpose it served.



**Figure 13: Study #2 user ratings comparing four treatment groups across experience metrics from the NASA TLX set, along with two additional evaluations of overall experience and easiness to learn, which are reversed in polarity in the graph.**

## 8.2 Natural language accelerates the development process and enables more creativity (Unity with AtomScript vs. AtomXR Desktop)

The introduction of natural language driven AtomScript generation allowed users to complete 100% of tasks (Fig. 12b), and complete tasks far faster than by directly writing AtomScript. On average, participants completed tasks 3.38 times as fast using *AtomXR Desktop* compared to *Unity with AtomScript* (Fig. 12a).

In general, participants noted several components of the *AtomXR Desktop* system they appreciated, particularly the use of natural language and the reduction of context switching. Reflecting on the two systems, P12, an expert in prior programming experience, said, "*Natural language is a lot easier than code, even for a relatively experienced coder*", and other participants similarly expressed that the ability to use natural language made the tasks easier and more stress-free. Beyond natural language, P13 observed that "*one big advantage of the AtomXR userface is that it is obviously not as bloated, so switching from Edit to Test only takes a second - whereas it takes much longer to test anything in the Unity environment*". Moreover, "*AtomXR has a huge advantage of having a naturally aligned play and edit mode: mapping the axes on the monitor [in Unity] to real life 3D was nonobvious and challenging*" (P12). In contrast to AtomXR, Unity's fragmentation between play and edit mode, combined with the need to change software to find assets and write logic (P13 found the "*switching between VSC<sup>5</sup> and Unity [to be] annoying*"), creates constant context switching that may feel overwhelming, particularly for beginners. Avoiding this cognitive load, participants found the ability to even be more creative, with P13 saying, "*AtomXR saves so much time and I feel like it gives me creative superpowers*". However, participants noted system features that could use improvement, with P7 saying they "*wish AtomXR gave the ability to manually edit the code, or look at text prompt history, since that would greatly increase developer workflow*" and P13 noting that "*AtomXR is still lacking some basic UI improvements*". These limitations are addressed in Sec. 11.3.

One interesting observation based on Fig. 12a is that there exist significant differences in variance of average task completion time across tasks for each system. In particular, the variance for *Unity with C#* is approximately 21502 seconds, which is reduced around ten-fold to around 2988 for *Unity with AtomScript*, and ten-fold

<sup>5</sup>VSC = Visual Studio Code

again to around 208 for *AtomXR Desktop*. This is likely because natural language input takes similar amounts of time for commands of different levels of complexity, while time taken to write code scales with complexity the more low-level the programming language is.

### 8.3 Immersion slows down the process due to input method issues but enhances fine-tuning objects in space (*AtomXR Desktop* vs. *AtomXR Headset*)

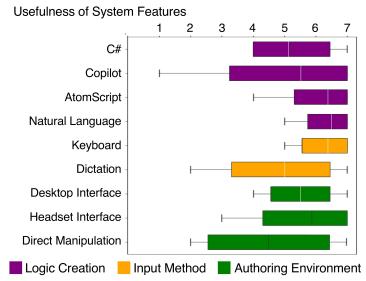
Interestingly, the addition of the immersive authoring component resulted in worse quantitative and qualitative metrics overall in the *AtomXR Headset* condition in comparison to *AtomXR Desktop*. Participants completed tasks on average 0.57 times as fast using *AtomXR Headset* compared to *AtomXR Desktop*. Still, *AtomXR Headset* allowed users to complete 100% of tasks in the allocated time.

In terms of user experience, *AtomXR Desktop* performed better along all dimensions, as shown in Fig. 13c. Qualitative data speaks to the main reason why participants felt this way. P14 explained that their headset performance was worse because of "*lag in pressing buttons and dictation*" and "*having to go back and edit text that didn't come through*" in dictation. Issues with pressing buttons, dictation inaccuracies, and other difficulties interacting with the 3D in-headset interfaces, caused the development system to be harder to use and slower. Participants tended to prefer familiar desktop input methods, as "*being able to use the mouse and type with a keyboard was much faster and less physically strenuous than headset*" (P14). Because of this, all four participants who experienced the desktop and headset variations preferred the desktop variation for early prototyping and found it easier to use.

Participants did, however, credit the immersive authoring environment for making it easier to move around, and visualize and manipulate objects. This is reflected quantitatively, as *AtomXR Headset* provides a significant improvement in speed for tasks involving editing of objects to match end-user perception (i.e., Task #3 for resizing the cherry to be life-sized). In the non-immersive authoring conditions, participants had to change the size using a desktop interface, stream the application to the headset for testing, and then go back and adjust the size again if their original estimation was incorrect. Being able to use hands to directly manipulate the object in 3D was a clear improvement for these types of authoring tasks. Moreover, participants appreciated other immersive methods, such as the "*eye-based "this" feature for naming objects instead of using their object id name*" (P10). Based on each system's advantages, P14 suggested "*combining the two systems, and using the VR for its advantages - spatial visualization, tactile manipulation - and using the desktop for its advantages of speed, precision of language input, and lack of physical exertion*".

### 8.4 Immersive authoring with natural language enhances overall development experience (*Unity with C#* vs. *AtomXR Headset*)

Finally, comparing *Unity with C#* with *AtomXR Headset*, we found participants completed tasks on average 2.84 times as fast using *AtomXR Headset* and were able to complete 100% of the tasks using *AtomXR Headset* in contrast to only 45% using *Unity with C#*. We



**Figure 14: User ratings on the usefulness of each component of the system, including approaches to logic creation, input methods, and authoring environments.**

also found experience improvements along all dimensions of evaluation except physical demand, which is reasonably explained by the strain of wearing the headset. Specifically, participants reported a 72% improvement in overall experience and 69% improvement in learning ease using *AtomXR Headset* over *Unity with C#*. Impressively, the average frustration experienced by participants was reduced by 63% using *AtomXR Headset*, followed by a 57% reduction in temporal demand, 36% reduction in mental demand, and 22% reduction in effort. Participants unanimously preferred AtomXR over Unity for early prototyping, and found it easier to use and easier to learn, supporting the results from Study #1.

Beyond being "*simple to use*" and "*alot easier to use for prototyping*" (P9), participants also "*felt great after [using AtomXR]*" (P8) and found it "*more fun to get results immediately after saying something*" (P9). Consistent with previous reports from participants, P1 explained that the ability to use natural language "*made the tasks so much easier; [as they] did not have to worry about learning the syntax or other programming related details*". P9 elaborated that "*using natural language helped with the performance because it is a more direct connection between your thoughts and input*." However, participants expressed concerns about the customizability and controllability of AtomXR, noting that "*it seems like [Unity] provides the developer with way more room for customization*" (P1).

Reflecting on in-headset authoring, participants again expressed it was "*much easier to get a sense of space*" (P5), which made manipulating the size and position of objects easier. Beyond functionality, the immersion of the authoring environment made users enjoy the experience more and perceive it as fun, rather than as purely a task- "*Once I got the hang of it, I kind of enjoyed it, and played it like a game,*" P8 commented. However, participants again reported frustrations with in-headset issues like lag time in interactions, interface sensitivity, and other issues with the HoloLens hardware.

**Usefulness of Features.** Aggregating the usefulness ratings for all different features from both the traditional Unity system and AtomXR Headset in Fig. 14, we found that the major trade-off made between the traditional Unity system and the full AtomXR Headset system revolves around the affordances of the device hardware. In developing the system, we had hoped that dictation would be a more natural form of interaction—however, participants indicated in their usefulness ratings and open-ended comments that they preferred the traditional keyboard as the input method within

development processes. Additionally, although participants appreciated the ability to directly manipulate objects from the perspective of their end user, the benefits of this feature could not outweigh the slowness of headset interaction. While in-headset input methods are expected to improve in the future, considering these trade-offs, P5 suggested "*AtomXR could use a hybrid system where I could do some authoring on the desktop.*"

Interestingly, when considering the integrated AtomXR Headset system, participants tended to notice the natural language and immersive authoring features more than AtomScript. This may be because the tasks in the experiment didn't necessitate extensive debugging or customization, reducing the need for participants to engage deeply with the generated AtomScript code. This illuminates the importance of different features based on application context, suggesting initial logic generation through natural language may be crucial during early prototyping stages, while fine-tuning via AtomScript becomes more vital as the project matures or becomes more complex. These two components of the system complement and depend on each other—the more robust the natural language code generation, the less reliant users become on AtomScript for adjustments (we examine the current limitations of our NLP methods in Appendix 11.3).

## 9 DISCUSSION AND FUTURE WORK

**Reduced intent-to-input translation contributes to better development experience.** Our results indicate that our immersive natural language driven authoring system is generally easier to learn and use than the traditional Unity development system. While the abstraction provided by AtomScript contributed to this ease, participants found the most added value in the ability to use natural language to describe their requests. Especially in logic creation tasks, participants appreciated the direct translation from intent to system input within AtomXR in comparison to the longer process required to translate intent to code using traditional methods. This lower cognitive demand often improved participants' sense of enjoyment during the development process. In object creation tasks, however, using natural language as an input method often slowed participants down. This can be explained by the innate difference between how people imagine different tasks—language is more often used in imagining functionality than in imagining visual designs. Thus, it may be more intuitive to use visual methods such as drag-and-drop GUIs for object or world creation tasks. Based on these insights, we believe design of future development tools would benefit from designing affordances based on characterization of how the user imagines tasks anticipated in the authoring process.

**Immersive authoring is not yet preferred due to inconvenient input methods.** The in-headset variation allowed participants to get a better sense of their end user's experience, reducing the gap between the development and testing process, in particular for manipulating objects in 3D space. The immersive, game-like environment of the in-headset variation was also found to be more fun and engaging. However, these advantages were outweighed by the difficulties of current input and interaction methods for headset devices. The frustrations expressed regarding lag and inaccuracies in dictation and touch interactions suggest that hardware issues still

may present a barrier to intensive usage of immersive authoring systems.

Based on these insights, we believe design of future development tools should optimize usability based on the provided affordances—given difficulties with direct text input and editing, systems can develop methods of communication focused less on precision. More modalities of input, such as eye gaze and gesture as we see in AtomXR, and better understanding of user-system interaction history can help elucidate imprecise requests and minimize the impact of input issues.

**Heading toward streamlined and naturalized prototyping environments.** Beyond natural language, participants appreciated AtomXR's streamlined environment, in which asset selection, world design, logic creation, and testing all happened within the same environment, eliminating the cognitive and time cost of context switching necessary in the traditional desktop systems. Most participants performed better and preferred using all variations of the AtomXR authoring system over traditional methods. The trend toward streamlined development is reflected in industry as well, where NVIDIA's Omniverse [5] attempts to provide a cohesive platform for development, deployment, and management of 3D applications. Based on our results, we believe that streamlined and naturalized authoring environments provide significant value to early stages of the XR prototyping process and make XR more accessible to a wide range of creators from various fields.

**Limitations.** Our studies and system have several limitations. The studies focused on a set of relatively simple development tasks. To fully gauge AtomXR's utility in real-world settings, a broader array of more complex usage scenarios should be evaluated. Additionally, the studies compare AtomXR with Unity to understand relative advantages and disadvantages of our system and a commonly-used traditional development system, but further studies are necessary to understand AtomXR's utility relative to other development tools.

In terms of system limitations, AtomXR is currently a proof-of-concept for a natural language-driven immersive prototyping tool. As such, it is not as capable as full development systems like Unity. For example, the current implementation of AtomXR does not allow users to build and deploy their applications to platforms outside of HoloLens 2. As suggested by study participants, AtomXR could be integrated with existing authoring workflows. AtomXR and Unity could complement each other in the development pipeline, where the developer can quickly create 3D worlds and prototype basic logic in AtomXR and then use Unity to add more complex functionality such as integrating 3rd party libraries or creating networking requests.

Another challenge lies in the system's accuracy in converting natural language intent to valid AtomScript code. GPT-3 often generates functions that are not implemented in AtomScript. See Appendix 11.3 for examples of errors in natural language to AtomScript translation. Specifically, Yin and Neubig [46] have shown that it's possible to parse natural language descriptions into syntactically valid code using a novel neural architecture. Future work could use this model to take in the underlying grammar of AtomScript as prior knowledge to ensure that generated AtomScript is correct.

Beyond initial generation accuracy, AtomXR's editing affordances are sparse. Editing in immersive authoring presents a major

challenge due to the difficulty of text input in XR [26, 45]. A potential solution to explore in future work is visual scripting as a method for fine-tuning generated logic. AtomXR could also benefit from retaining and using a history of user interactions as context for understanding user intents. Being a research prototype, AtomXR also lacks the support, documentation, and community of more popular tools. Moreover, the user interface could be dramatically improved in future work.

**Implications for Future XR Prototyping.** In the evolving landscape of XR and AI-driven content, our work reflects the shift from passive observation to active participation in world creation. By making functional XR development accessible, we blur the lines between participant and author towards shared, peer-to-peer realities. Our use of NLP techniques to power our system reflects the larger rise in usage of NLP technologies across fields, which may have fundamental implications for the ways in which we think and create in the future.

## 10 CONCLUSION

In this paper, we introduce AtomXR, an innovative XR authoring system that leverages NLP with multimodal interactions to simplify the prototyping process. We contributed a streamlined immersive authoring environment, natural language driven multimodal interaction scheme, and a custom, high-level language designed for XR prototyping. In a two-part user study, we evaluated AtomXR on typical XR development tasks and found it not only accelerated task completion by 2-5 times but was also more intuitive and easier to learn. We further compared the effects of immersion, natural language, and AtomScript on development experiences. Based on these findings, we suggest avenues for future research to refine this promising approach to XR prototyping.

## REFERENCES

- [1] [n. d.]. Builder Bot demo. <https://www.youtube.com/watch?v=62RJv514ijQ>
- [2] [n. d.]. Frame at Microsoft Build: Speaking Worlds into Existence. <https://learn.framevr.io/post/msbuild2022#:~:text=Frame%20is%20a%20web%20based,a%20Frame%20for%20free%20here.&text=If%20playback%20doesn't%20begin%20shortly%C2%20try%20restarting%20your%20device>.
- [3] 1998. Unreal Engine. <https://www.unrealengine.com/>
- [4] 2005. Unity. <https://unity.com/>
- [5] 2021. NVIDIA Omniverse. <https://www.nvidia.com/en-us/omniverse/>
- [6] A-FRAME 2015. A-FRAME. <https://aframe.io/>.
- [7] Narges Ashtari, Andrea Bunt, Joanna McGrenere, Michael Nebeling, and Parmit K Chilana. 2020. Creating augmented and virtual reality applications: Current practices, challenges, and opportunities. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–13.
- [8] Jerome R Bellegarda. 2013. Spoken language understanding for natural interaction: The siri experience. *Natural Interaction with Robots, Knobots and Smartphones: Putting Spoken Dialog Systems into Practice* (2013), 3–14.
- [9] Richard A. Bolt. 1980. “Put-That-There”: Voice and Gesture at the Graphics Interface. *SIGGRAPH Comput. Graph.* 14, 3 (jul 1980), 262–270. <https://doi.org/10.1145/965105.807503>
- [10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [11] Don Brutzman and Leonard Daly. 2010. *X3D: extensible 3D graphics for Web authors*. Elsevier.
- [12] Jeff Butterworth, Andrew Davidson, Stephen Hench, and Marc T Olano. 1992. 3DM: A three dimensional modeler using a head-mounted display. In *Proceedings of the 1992 symposium on Interactive 3D graphics*. 135–138.
- [13] Ricardo Cabello. 2010. three.js. <https://threejs.org/>.
- [14] Jordi Cabot. 2020. Positioning of the Low-Code Movement within the Field of Model-Driven Engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (Virtual Event, Canada) (MODELS ’20)*. Association for Computing Machinery, New York, NY, USA, Article 76, 3 pages. <https://doi.org/10.1145/3417990.3420210>
- [15] Leonor Adriana Cardenas-Robledo, Óscar Hernández-Uribe, Carolina Reta, and José Antonio Cantoral-Ceballos. 2022. Extended reality applications in industry 4.0-A systematic literature review. *Telematics and Informatics* (2022), 101863.
- [16] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. 2018. Universal sentence encoder for English. In *Proceedings of the 2018 conference on empirical methods in natural language processing: system demonstrations*. 169–174.
- [17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paine, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). <https://doi.org/10.48550/ARXIV.2107.03374>
- [18] Robert Dale. 2021. GPT-3: What’s it good for? *Natural Language Engineering* 27, 1 (2021), 113–118. <https://doi.org/10.1017/S1351324920000601>
- [19] Jessyca L Derby, Christopher T Rarick, and Barbara S Chaparro. 2019. Text input performance with a mixed reality head-mounted display (HMD). In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Vol. 63. SAGE Publications Sage CA: Los Angeles, CA, 1476–1480.
- [20] Marin Cutumisu, et al. 2007. ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming* 67, 1 (Jan. 2007), 32–58. <https://doi.org/10.1016/j.scico.2007.01.005>
- [21] Luis Fernández-Núñez, Dario Penas, Jorge Viteri, Carlos Gómez-Rodríguez, and Jesús Vilares. 2020. Developing Open-Source Roguelike Games for Visually-Impaired Players by Using Low-Complexity NLP Techniques. In *Proceedings*, Vol. 54. mdpi, 10.
- [22] Google. 2016. *Google Tilt Brush*. <https://www.tiltbrush.com/>
- [23] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage publications Sage CA: Los Angeles, CA, 904–908.
- [24] Bret Jackson and Daniel F Keefe. 2016. Lift-off: Using reference imagery and freehand sketching to create 3d models in vr. *IEEE transactions on visualization and computer graphics* 22, 4 (2016), 1442–1451.
- [25] Veton Kepuska and Gamal Bohouta. 2018. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*. IEEE, 99–103.
- [26] Florian Kern, Florian Niebling, and Marc Erich Latoschik. 2023. Text Input for Non-Stationary XR Workspaces: Investigating Tap and Word-Gesture Keyboards in Virtual and Augmented Reality. *IEEE Transactions on Visualization and Computer Graphics* 29, 5 (2023), 2658–2669.
- [27] Anis Koubaa. 2023. GPT-4 vs. GPT-3.5: A concise showdown. (2023).
- [28] Gun A Lee, Gerard J Kim, and Mark Billinghurst. 2005. Immersive authoring: What you experience is what you get (wyxiwyg). *Commun. ACM* 48, 7 (2005), 76–81.
- [29] Germán Leiva, Jens Emil Grønbæk, Clemens Nylandsted Klokmose, Cuong Nguyen, Rubaiat Habib Kazi, and Paul Asente. 2021. Rapido: Prototyping Interactive AR Experiences through Programming by Demonstration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 626–637.
- [30] Blair MacIntyre, Maribeth Gandy, Steven Dow, and Jay David Bolter. 2004. DART: a toolkit for rapid design exploration of augmented reality experiences. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*. 197–206.
- [31] Microsoft. 2018. *Microsoft Maquette*. <https://www.maquette.ms/>
- [32] Mark Mine. 1995. ISAAC: A virtual environment tool for the interactive construction of virtual worlds. (1995).
- [33] Mark Mine, Arun Yoganandan, and Dane Coffey. 2014. Making VR work: building a real-world immersive modeling application in the virtual world. In *Proceedings of the 2nd ACM symposium on Spatial user interaction*. 80–89.
- [34] David R Nadeau. 1999. Building virtual worlds with VRML. *IEEE Computer Graphics and Applications* 19, 2 (1999), 18–29.
- [35] Michael Nebeling, Janet Nebeling, Ao Yu, and Rob Rumble. 2018. Protoar: Rapid physical-digital prototyping of mobile augmented reality applications. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.
- [36] Oculus. 2016. *Oculus Medium*. <https://www.oculus.com/medium/>

- [37] Randy Pausch, Tommy Burnette, AC Capeheart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, and Jeff White. 1995. Alice: Rapid prototyping system for virtual reality. *IEEE Computer Graphics and Applications* 15, 3 (1995), 8–11.
- [38] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).
- [39] Russell A Poldrack, Thomas Lu, and Gašper Beguš. 2023. AI-assisted coding: Experiments with GPT-4. *arXiv preprint arXiv:2304.13187* (2023).
- [40] Kevin Ponto, Ross Tredinnick, Aaron Bartholomew, Carrie Roy, Dan Szafir, Daniel Greenheck, and Joe Kohlmann. 2013. SculptUp: A rapid, immersive 3D modeling environment. In *2013 IEEE Symposium on 3D User Interfaces (3DUI)*. IEEE, 199–200.
- [41] Jasmine Roberts, Andrzej Banburski-Fahey, and Jaron Lanier. 2022. Steps towards prompt-based creation of virtual worlds. (2022). <https://doi.org/10.48550/ARXIV.2211.05875>
- [42] Brenden Sewell. 2015. *Blueprints visual scripting for unreal engine*. Packt Publishing Ltd.
- [43] Anthony Steed and Mel Slater. 1996. A dataflow representation for defining behaviours within virtual environments. In *Proceedings of the IEEE 1996 Virtual Reality Annual International Symposium*. IEEE, 163–167.
- [44] Oleksandr Polozov Sumit Gulwani and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (July 2017). <https://doi.org/10.1561/2500000010>
- [45] Tian Yang, Powen Yao, and Mike Zyda. 2022. Flick Typing: Toward A New XR Text Input System Based on 3D Gestures and Machine Learning. In *2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*. IEEE, 888–889.
- [46] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 440–450. <https://doi.org/10.18653/v1/P17-1041>
- [47] 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.

## 11 APPENDICES

### 11.1 Survey Questions

The following are the post-task survey questions participants answered after each task:

- How easy was it to complete each task with the system? (rating)
- How would you rate your overall experience with this system? (rating)
- How easy was it to learn how to use this system? (rating)
- How would you rate your physical comfort while using this system (e.g. eye strain, head ache, etc.)? (rating)
- How mentally challenging was the task? (rating)
- How physically challenging was the task? (rating)
- How hurried or rushed was the pace of the task? (rating)
- How successful were you in accomplishing what you were asked to do? (rating)
- How hard did you have to work to accomplish your level of performance? (rating)
- How insecure, discouraged, irritated, stressed, and annoyed were you? (rating)

The following are the post-experiment survey questions participants answered after completing the entire experiment:

- Which system was easier to use? (multiple-choice)
- Which system was easier to learn? (multiple-choice)
- Which system would you prefer for early prototyping? (multiple-choice)
- Which system did you prefer overall and why? (open-ended)

- What other functionality do you think each system is lacking? (open-ended)
- What did you like most about each tool? (open-ended)
- What did you dislike most about each tool? (open-ended)
- Reflect on the differences in your experience using each system. What did you like and dislike about each? (open-ended)
- How useful was each of the features in assisting with development? (rating)
- Describe if and how each feature impacted your performance on the task in comparison to using Unity. (open-ended)

## 11.2 AtomScript Specification

For reproducibility, the grammar of AtomScript is defined below in G4 file format (used to define grammars for ATNLR4).

```

1 grammar Hello;
2
3 program: line* EOF;
4
5 line: statement | ifBlock | foreverBlock | onStartBlock |
       onCollisionBlock | onButtonPressBlock;
6
7 statement: (assignment | functionCall) ';';
8
9 ifBlock: 'if' expression block ('else' elseifBlock)?;
10
11 elseifBlock: block | ifBlock;
12
13 foreverBlock: FOREVER block;
14
15 FOREVER: 'forever';
16
17 onStartBlock: ONSTART block;
18
19 ONSTART: 'onStart';
20
21 onCollisionBlock: ONCOLLISION '<' constant ',' constant '>' block;
22
23 ONCOLLISION: 'onCollision';
24
25 onButtonPressBlock: ONBUTTONPRESS '<' constant '>' block;
26
27 ONBUTTONPRESS: 'onButtonPress';
28
29 COMMENT: '/*' .*? '*/';
30
31 LINE_COMMENT: '//' ~[\r\n]*;
32
33 assignment: IDENTIFIER '=' expression;
34
35 functionCall:
36   IDENTIFIER '(' (expression (',' expression)*)? ')';
37
38 array
39 : '[' ( expression ( ',' expression )* )? ']'
40 ;
41
42 expression:
43   constant
44   | IDENTIFIER
45   | array
46   | functionCall
47   | '(' expression ')'
48   | '!' expression

```

```

49 | expression multOp expression
50 | expression addOp expression
51 | expression compareOp expression
52 | expression boolOp expression;
53
54 multOp: '*' | '/';
55 addOp: '+' | '-';
56 compareOp: '==' | '!=' | '<' | '>' | '<=' | '>=';
57 boolOp: '&&' | '||';
58
59 INTEGER: [0-9]+;
60 FLOAT: [0-9]+ '.' [0-9]+;
61 STRING: ("\" ~\"* \"") | ('\\' ~'\\'* '\\');
62 BOOL: 'true' | 'false';
63 NULL: 'null';
64
65 constant: INTEGER | FLOAT | STRING | BOOL | NULL;
66
67 block: '{' line* '}';
68
69 WS: [ \t\r\n]+ -> skip;
70 IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*;
```

### 11.3 Natural Language to AtomScript Translation Error Examples

We present several examples of natural language-code pairs and compare the code outputted by GPT-3 with the expected code output to demonstrate the limitations of the current NLP process (see Table 2).

We observed that in examples 1, 2, and 5, the AtomScript generated by GPT-3 worked equivalent to the reference AtomScript. However, in examples 3 and 4, the generated AtomScript showed two common failure points.

Firstly, in example 3, GPT-3 did not define the scoreboard variable beforehand, likely because the boundaries of this natural language request are unclear. This kind of request could often appear in a context in which the user has already defined a scoreboard system. GPT-3 also omits the implementation of the collision logic, likely because it assumes these functions have been implemented in surrounding context. To address this, we plan to store a history of past commands to give more context.

The other failure point was in example 4, where GPT-3 referenced a function that actually doesn't exist in our system (there is no "Wait" function). Although this could be easily implemented and is a plan for future work, a syntax-based language model would be a better approach for generating valid code.

Received 14 Sep 2023; revised 14 Sep 2023; accepted 14 Sep 2023

Table 2: A comparison of AI generated AtomScript translation and reference translation

Prompt	GPT Generated AtomScript	Reference AtomScript
Whenever the player collects coin1, play a coin collection sound	<pre> 1 onCollision&lt;"Player", "coin1"&gt; { 2     PlaySound("Coin collect"); 3 }</pre>	<pre> 1 onCollision&lt;"Player", "coin1"&gt; { 2     PlaySound("Coin collect"); 3 }</pre>
Make box1 disappear if at least 5 seconds have passed and I've hit it twice	<pre> 1 currTime=TimeSinceStart(); 2 numCollisions=0; 3 onCollision&lt;'Player', 'box1'&gt; { 4     numCollisions=numCollisions+1; 5 } 6 forever { 7     if(TimeSinceStart()-currTime &gt;= 5 &amp;&amp; 8         numCollisions&gt;=2) { 9         Disappear('box1'); 10    } 11 }</pre>	<pre> 1 numCollisions = 0; 2 forever { 3     if (TimeSinceStart() &gt; 5) &amp;&amp; (numCollisions 4         &gt;= 2) { 5         Disappear("box1"); 6     } 7 onCollision&lt;"Player", "box1"&gt; { 8     numCollisions = numCollisions + 1; 9 }</pre>
Make it so that enemy1 starts following me after I get 5 coins or 30 seconds have passed	<pre> 1 forever { 2     if(scoreboard&gt;=5    TimeSinceStart()&gt;=30) { 3         Move('enemy1', 'slow', GetPosition('Player' 4             )-GetPosition('enemy1')); 5     } 5 }</pre>	<pre> 1 coinsCollected = 0; 2 forever { 3     if (coinsCollected &gt;= 5)    (TimeSinceStart() 4         &gt; 30) { 5         Move("enemy1", "slow", GetPosition( 6             "Player") - GetPosition("enemy1")); 7     } 8 onCollision&lt;"Player", "Coin1"&gt; { 9     coinsCollected = 1 + coinsCollected; 10 }</pre>
Make apple1 flash on and off every second	<pre> 1 forever { 2     ChangeColor('apple1',[1,0,0]); 3     Wait(1); 4     ChangeColor('apple1',[0,0,0]); 5     Wait(1); 6 }</pre>	<pre> 1 isOn = true; 2 timeElapsed = 0; 3 iter = 0; 4 forever { 5     if(timeElapsed &gt;= 1){ 6         if(isOn){ 7             Disappear("apple1"); 8             isOn = false; 9         } 10        if(isOn == false){ 11            Appear("apple1"); 12            isOn = true; 13        } 14        iter = iter + 1; 15    } 16    timeElapsed = timeSinceStart() - iter*1; 17 }</pre>
Make it so that cube2 chases me.	<pre> 1 forever { 2     Move('cube2', 'fast',GetPosition('Player')- 3         GetPosition('cube2')); 3 }</pre>	<pre> 1 forever{ 2     Move('cube2', 'fast',GetPosition('Player')- 3         GetPosition('cube2')); 3 }</pre>