Ivy: Exploring Spatially Situated Visual Programming for Authoring and Understanding Intelligent Environments

Barrett Ens[†]
Autodesk Research, University of Manitoba

Fraser Anderson[‡]
Autodesk Research

Tovi Grossman[‡]
Autodesk Research

Michelle Annett[‡]
Autodesk Research

Pourang Irani[†]
University of Manitoba

George Fitzmaurice[‡]
Autodesk Research

ABSTRACT

The availability of embedded, digital systems has led to a multitude of interconnected sensors and actuators being distributed among smart objects and built environments. Programming and understanding the behaviors of such systems can be challenging given their inherent spatial nature. To explore how spatial and contextual information can facilitate the authoring of intelligent environments, we introduce Ivy, a spatially situated visual programming tool using immersive virtual reality. Ivy allows users to link smart objects, insert logic constructs, and visualize real-time data flows between real-world sensors and actuators. Initial feedback sessions show that participants of varying skill levels can successfully author and debug programs in example scenarios.

Keywords

Virtual reality; mixed reality; visual programming language; spatial interaction; internet of things; immersive analytics.

Index Terms

H.5.2. Information interfaces and presentation: User interfaces

1 INTRODUCTION

Ubiquitous Computing scenarios [67] have led to an increased number of objects that contain embedded circuitry and logic. This manifestation, known colloquially as the Internet of Things (IoT), is expected to grow exponentially to over 20 billion connected objects by 2020 [22]. Taking full advantage of this network of sensors, actuators and smart objects will require sophisticated tools that allow users to control the flow of data and understand logical connections between numerous spatially situated devices.

To enable end-users with limited programming skills to author and visualize embedded control logic, 2D visual environments such as Wyliodrin [70] and Node-RED [6] have been introduced. These graphical tools can help users understand logical connections between objects, but do not inherently reveal spatial relationships with other objects or their physical surroundings. However, end-users, including both professionals and amateurs, may benefit from spatial knowledge of connected objects in smart environments. For example, a technician touring a 'smart' building may be able to identify the source of anomalous sensor readings more easily by inspecting the immediate surroundings for air leaks, water damage or other signs of problems. Likewise, spatial information may help a hobbyist identify a specific motor among several distributed objects with less effort than by using abstract identifiers alone.

The Reality Editor [30–32] has recently provided a starting point for spatially situated visual programming, with an implementation that allows authoring of basic connections between smart objects using augmented reality (AR) on handheld devices. We build on these efforts by exploring how such experiences can be improved with an immersive, wearable platform. Wearable mixed reality displays eliminate the mobile screen's barrier, and free the user's hands to interact directly with the surrounding world. While this opens intriguing opportunities for wearable AR platforms, we instead employ a room-scale virtual reality (VR) platform, which provides a wide field of view (FoV) and bimanual controllers, allowing us to focus our efforts on spatial UI design. Further, VR provides opportunities to explore use-cases beyond the user's immediate situation and physical laws, such as working in remote locations, or scaling time and space.

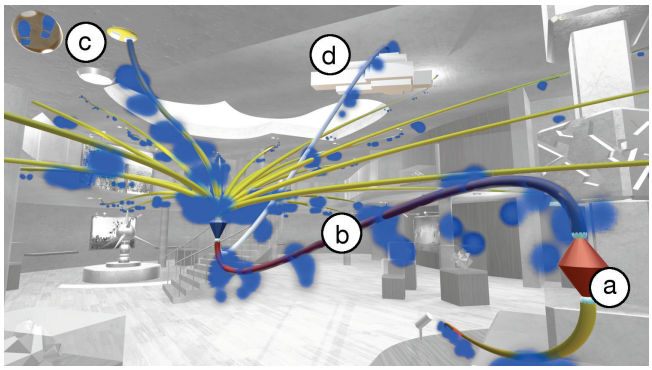

Figure 1: Ivy is an immersive visual programming tool for authoring IoT programs and visualizing sensor data. Users can (a) create program constructs, (b) establish logical links, (c) visualize data flows from real-world sensor data, and (d) upload data to the cloud. Virtual port nodes (c) act as interfaces to real-world sensors, such as foot-traffic readings in this museum scenario.

The main contribution of this paper is Ivy (Figure 1), a spatially situated visual programming environment for authoring logical connections and understanding dataflow between smart objects. Guided by a set of design guidelines for situated programming drawn from prior work on spatial interaction, this immersive VR application explores several areas, such as data flow visualization, debugging tools, and real-time deployment. Users of Ivy can create functioning programs with live sensor data from real objects in the external environment. To gain feedback on the utility of such a system, we invited several participants with varying levels of IoT, VR, and programming expertise to try Ivy. Their performance and positive feedback show potential for such immersive tools, as well as future avenues for AR adaptations.

[†]{bens, irani}@cs.umanitoba.ca
[‡]{first.last}@autodesk.com

16-19 May, Edmonton, Alberta, Canada

2 Related Work

Ivy builds on prior work in visual programming languages, 3D authoring tools, and tools for visualizing and authoring IoT systems.

2.1 Visual Programming Environments

Visual programming languages such as Scratch [50] and Alice [10] provide visual and spatial depictions to make programming more approachable for children or non-experts. Similar approaches have been taken to author IoT control logic, as in Node-RED [6], WoTKit [5] and Wyliodrin [70]. While studies on their cognitive benefits are inconclusive [7,69], visual representations have persisted in numerous professional tools, such as LabView [37], SimuLink [57], Max [42], and Grasshopper for Rhino [25].

2.2 3D and Immersive Authoring Tools

Immersive authoring tools allow users to create content within the 3D virtual environments they are designing. For example, CHIMP [44] and CaveCAD [53] enable users to position and transform 3D objects directly in a scene, while Holosketch [11], TiltBrush [62] and CavePainting [35] provide immersive environments for free-form, 3D sketching. Guven et al. [28] explored handheld AR techniques for authoring multimedia in situ.

A few research projects have explored 3D visual programming with abstract structures [29,47,49,59], however, virtual tools have been more widely applied in situations that encompass clear benefits of 3D space. For example, tools have been incorporated into virtual environments to allow authoring of scene objects and behaviours [38,52,58] while immersed within, allowing users to more readily understand the behaviour of their scene and the element. Commercial tools that combine visual programming with 3D scenes include Dynamo for Revit [16], Blueprints visual scripting for the Unreal Engine [21] and as a Maya plugin [46], Flow for Autodesk's Stingray [60], Virtools [1], and Doom SnapMap [14].

Like these latter examples, Ivy promotes understanding using spatial relationships between logic constructs and their related objects, but in a differing context of control logic for IoT.

2.3 IoT Visualization Tools

A variety of projects have explored the visualization of IoT-generated data. Dashboards [23,54] are commonly applied in desktop environments for monitoring IoT deployments, but do not typically provide access to control logic. Other projects present sensor data visualizations in the context of their spatial environment [17], [36,55], however these tools largely focus on the analysis and aggregation of data that is superimposed on a static scene model.

Handheld AR has been used for in situ visualization of connections between objects [43,63] or sensor data [18,27,34,64,68]. In contrast, Ivy visualizes not only data values, but also reveals the flow of data between program constructs and situated objects to assist the comprehension of IoT program structures.

2.4 IoT Authoring Tools

Numerous authoring tools have been developed to ease the burden of creating networks of sensors or IoT-connected devices, for instance, applications for developing context-aware systems [12,39,51] and for programming by demonstration [13]. Visual programming approaches such as Jigsaw [33] have been developed for IoT applications, however, immersive authoring has been little explored for such applications.

The primary inspirational groundwork for Ivy is the Reality Editor [30,31], a handheld AR interface that allows users to link objects that are within view of a smartphone camera. Expansions of this work include AR-based UIs for smart objects [32], and back-end connections to physical artifacts [45]. We build on this work with an immersive implementation that provides more advanced programming and debugging features, and eliminates the need to hold a device. Ivy's multi-tool palette supports bimanual, spatial interaction to provide direct control of logic constructs, allowing larger and more complex programs to be authored.

3 Advantages of Mixed Reality for Spatial Tasks

A variety of mixed reality platforms have recently become sophisticated enough to support situated programming. Three general categories of platforms are handheld augmented reality (*HAR*), head-mounted augmented reality (*AR*), and virtual reality (*VR*) (Table 1).

Handheld augmented reality overlays virtual objects on the camera feed of a mobile phone or tablet. Such systems are portable but have inherent limitations; for example, presenting a wide FoV results in visible distortion, while interactions are constrained to the display space of the held device. The Reality Editor [30,31], the only mixed reality system for situated programming in smart environments to date, uses handheld augmented reality.

Head-mounted augmented reality superimposes virtual content directly over physical surroundings. The available rendering FoV varies with hardware, but could potentially be wider than HAR; see-through displays provide a wide view of the real-world scene without distortion. A key advantage of head-mounted AR is that users can interact normally with physical objects. The ability to interact with the real world while also manipulating virtual artifacts makes AR potentially useful for spatially situated programming.

Virtual reality systems immerse users in simulated environments. Although users are disconnected from the real world, virtual environments can provide certain advantages. Whereas the advantages of AR are based in the ability to situate users in relevant spaces, the advantages of VR lie in its purely virtual worlds; VR can provide access to remote locations, dangerous environments, or buildings not yet constructed. IoT technicians can assist architects at design time, using simulated sensors to configure smart environment behaviours. VR also supports the manipulation of space and time, such as changes in scale, teleportation, or other interactions not available in physically situated AR systems [8].

Based on these properties, it is promising to explore mixed-reality authoring within a head-mounted configuration, as it can provide a greater level of immersion than HAR, and frees the user's arms for intuitive, bimanual interaction. While there are many potential benefits of a situated, wearable AR system, we focus our current exploration on a room-scale VR platform, which provides several advantages, including a precise global tracking system and a wider FoV than currently available AR platforms.

	HAR [31]	***AR***	***VR***
Immersive Visualization	Limited FOV	Yes	Yes
In Situ Use	Yes	Yes	Occluded
Remote Use	No	No	Yes
Bimanual Interaction	No	Yes	Yes
Physical Access	Yes	Yes	Occluded
Space/Time Manipulation	No	No	Yes

Table 1: Advantages of mixed reality platform categories.

4 Spatially situated Visual Programming

Next, we explore a spatially situated programming environment that supports the authoring and editing of programs by manipulating 3D logic constructs in a relevant (in situ or virtual) spatial context. We propose that spatially situated programming will be well suited to high-level programming tasks where spatial context is important, for instance authoring connections in a smart home, or debugging linkages in a complex network of sensors and actuators. End-users engaging in such tasks will benefit from their inherent spatial knowledge of object layouts and from the advantages of spatial interaction methods. Conversely, as is unlikely that spatially situated programming will support low-level or abstract program components, it will likely complement, rather than replace, current desktop programming methods.

4.1 Design Guidelines

Drawing from the literature on spatial interaction, mixed reality, and interface design, we devised the following four guidelines for developing a spatially situated visual programming environment. These guidelines emphasize the spatial nature of situated constructs.

D1. Maintain spatial relationships: A key advantage of situated programming is that relationships between objects and logic constructs can be clearly depicted. Spatial association has been used to allow users to organize computer artifacts [65] or recall commands [40]. Similarly, spatial coupling between objects and abstractions will help users associate smart objects with related logic constructs to assist in the tracing and debugging of complex systems of sensors and actuators.

D2. Facilitate spatial interaction: Spatial interaction methods have been shown to possess clear advantages over abstract interaction methods in many contexts [2,19,40,41]. For instance, spatial locations can be referenced intuitively, using head or body motion, and efficient bimanual interaction techniques that mimic familiar physical manipulations can be applied.

D3. Embrace physical properties: Just as existing visual environments use properties such as colour and shape to distinguish programming elements [48,50,70], immersive environments provide additional opportunities for symbolic representations, wherein logic constructs can be given a seemingly physical form. Scale, shape, and rendering style can all be used to communicate a construct's function and properties. Top-to-bottom execution of logic constructs can leverage the downward direction of gravity to denote the flow of logic. As users are free to move around the constructs in 3D spatial environments, the traditional left-to-right logic flow breaks down, whereas top-to-bottom remains constant.

D4. Expose minimally sufficient information: Because end-users of situated visual programming environments may be non-expert programmers who are preoccupied by their surroundings, tools should be lightweight, modular, and easy to use. Also, because the program is situated within an external environment, care should be taken to avoid visual clutter [4,24]. Only necessary details about the program surface layer need be presented, with details about specific parameters and the underlying implementation kept at a minimum and presented only as required.

5 Ivy

With the above design guidelines in mind, we created a system to explore spatially situated visual programming on a room-scale VR platform. Ivy allows end-users to visualize, author, and edit the behaviors of physical IoT sensors and actuators. Ivy uses a dataflow programming model and is inspired by visual and functional aspects of Scratch [50] and Max [42].

5.1 Conceptual Model

Smart objects are intertwined with our daily environments, but their interlinking connections and governing logic remains hidden. To visualize these abstract concepts, logic constructs are presented as virtual *nodes* and *links*. Nodes represent interfaces to smart objects, logical operators, and program functions. Links represent the paths of data and program flows between constructs. Each intelligent object has a virtual port, providing access to sensors or actuators.

Programming occurs in an immersive environment, which allows spatial interaction with situated objects. Despite authoring being virtual, the program is connected to real, physical objects. Input is driven by sensors in the environment, program behaviours control real-world actuators, and output is directed to cloud servers. When a program is run, data can be observed flowing through the constructs, to facilitate understanding of the program logic.

5.2 Implementation Platform

Ivy is implemented using an HTC Vive, which provides room-scale VR. Our implementation runs in Unity 3D, on a PC with an Intel Xeon processor and GeForce GTX 970 graphics card. Program constructs are fully functional and can be connected to simulated or real-world sensors and actuators, as well as a real-time cloud-based data logging service (Figure 2). Ivy drives physical object behaviors with actuation commands sent through a REST API.

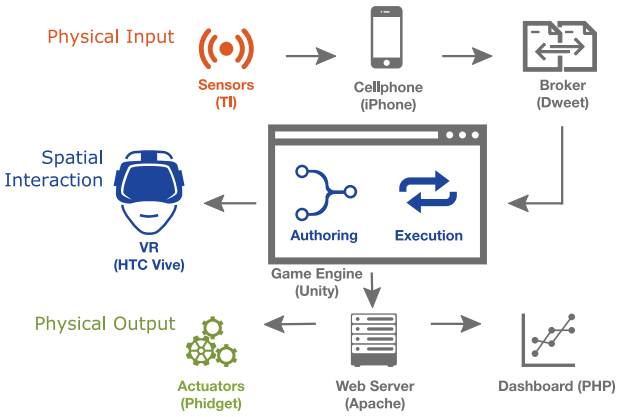

Figure 2: Ivy system overview.

5.3 Program Constructs

Ivy uses shape and colour variations to differentiate between categories of functionality (D3). Users have access to a variety of pre-set functions, with minimal GUI-based controls for adjusting parameters (D4). While a robust system would require a large library of pre-set functions, Ivy's initial subset allows sufficient functionality to demonstrate and evaluate key concepts.

5.3.1 Logic Nodes

Ivy's programming functions are organized within six types of logic nodes (Figure 3). For visual consistency, program flow executes from each node's top input to its bottom output connector (D3).

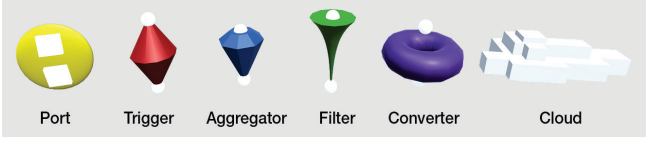

Figure 3: Programming constructs in Ivy are represented nodes. Each category is given a distinct shape and colour.

Ports are virtual representations of sensors and object behaviors and provide access to underlying data. Ports are fixed to the virtual representations of their associated physical objects (D1). To reduce visual clutter, multiple behaviors are grouped in a single Port (D4).

Trigger nodes invoke actions given a specific set of conditions. These are conceptually similar to 'if-then' constructs of imperative languages, and are a fundamental tool within many IoT programming languages [9,48].

Aggregator nodes contain common functions that aggregate multiple input values into a single value, such as a mean or sum. Ivy's aggregator functions also include the n-ary logical operators AND and OR.

Filter nodes represent standard functions for smoothing sensor data such as low-pass filtering.

Converter nodes convert data between types or ranges of values. For example, the 'invert' function flips high output values to low, and vice versa.

Cloud nodes represent channels to external servers, allowing users to log data output from an Ivy program for later inspection or to retrieve data from an external source.

5.3.2 Connector Links

Ivy represents connections between nodes using curved links that connect the output connector of one node to the input connector of the next node of a logic chain (D3). To help identify connected nodes, the end of the link matches the colour of the node at the opposite end of the link. Ivy does not restrict connections between nodes, but relies on the user to create the desired configurations. Ivy allows multiple links to 'fan in' to or 'fan out' of a single connector.

5.4 Interaction

Users of Ivy can author program constructs by creating new logic nodes and specifying links. Users can also inspect and edit existing program structures and observe data flow in real time (D1) to identify potential problems and debug programs. Below we describe what such interaction entails.

5.4.1 Interaction and Navigation

Ivy's functionality is accessed using a bimanual tool palette and wand (Figure 4). The tools are held in the left-hand palette and selected via the right-hand wand. For object manipulation, Ivy uses a combination of proximity and raycasting [66]. Nearby items can be selected using the wand's trigger button (D1), whereas distant items can be selected with the wand's laser. When needed, the *navigator* tool allows for teleportation within a large virtual space.

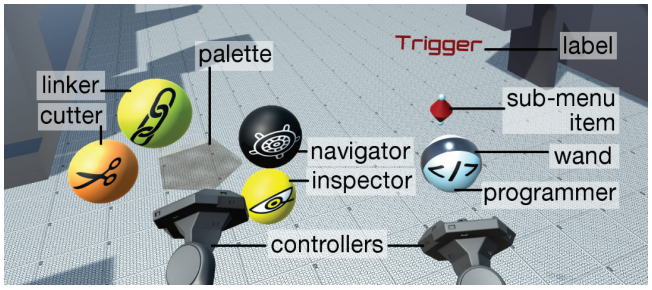

Figure 4: Ivy's Wand and Palette menu system.

5.4.2 Authoring Program Constructs

Users can author new programs within Ivy by creating node and link constructs in the virtual environment. Port nodes are spatially fixed to their related objects in advance (D1) and are initialized with corresponding sensor and actuator types. Ports and Cloud nodes act as inputs and outputs to the system, respectively, and may be linked to any number of other nodes. Once placed, nodes remain fixed at their drop point, except for Clouds, which rise to a height of 4 meters to indicate they are external to the current environment (D3).

5.4.3 Authoring Links

The *linker* tool is used to author node connections. While linking, an *anchor* line from the linker tool to the start node signals that a link is in progress. If a user changes tools during linking, the anchor connection remains intact. The persistence of the anchor allows intermediary operations, for instance, selecting a start node before navigating to a prospective end node. When a new link is initiated, existing links shrink to reduce occlusion of the scene (D4). Nodes can be un-linked with the *cutter* tool.

When linking to or from a Port node, users must select a *source* or *sink*. Sources are an object's sensors or other internal parameters (e.g. temperature), while sinks are actuated behaviors or object functions (e.g. enter/exit sleep mode). When a Port is selected, the available sources and sinks are presented by a set of *medallions*, distributed in an arc above the Port (Figure 5a). Every other node type has a custom GUI that appears when linked to. These GUIs provide access to pre-set functions and parameters (Figure 5b; D4) and can be re-summoned with the *programmer* tool.

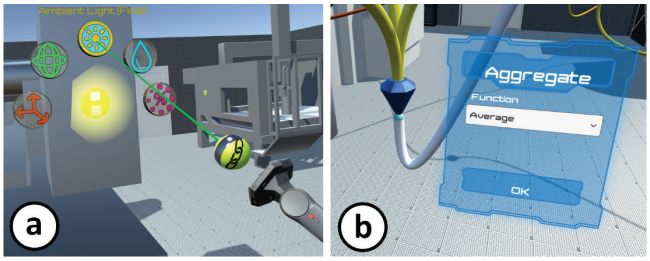

Figure 5: a) Medallions are shown when a port is selected. b) All other node types have an associated GUI.

5.4.4 Link Autocomplete

To expedite the node linking process, we implemented an autocomplete feature inspired by OctoPocus [3]. After a start node is selected, Ivy reveals the paths of all possible links to existing nodes (Figure 6). Pulsating transparent lines differentiate possible links from existing ones. The angle between the wand laser's current direction and a given node are mapped to link transparency (D2). To eliminate the need for precise targeting, the nearest link within a threshold angle of 30° can be selected.

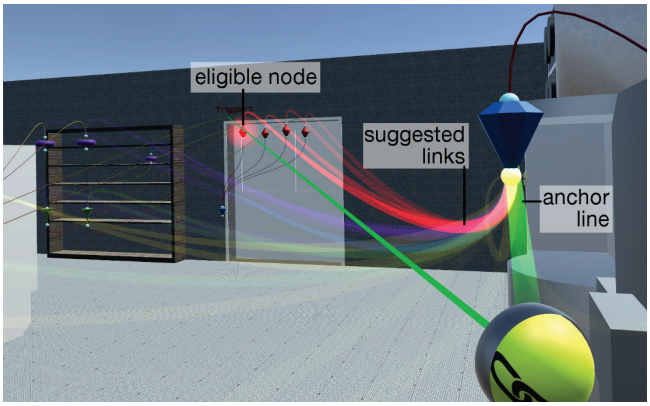

Figure 6: The autocomplete feature displays possible links.

5.5 Program Inspection and Data Flow Visualization

The inspector tool provides features to help users understand a program's control structure and purpose. By investigating program constructs, users can identify and debug potential problems.

5.5.1 Program Execution

Ivy uses a stack-based synchronous execution model [42], employing a single link type for all data. This reduces complexities, such as the unpredictable arrival sequence of various parameters or the need for multiple identical links with different types. Program data is passed along links in generic packages, each containing a list of values with mixed type. Ivy currently supports *Float*, *Vector3*, and *Boolean*, however the data flow model is extensible. If multiple links connect to a node, all incoming packages are merged.

When the program structure is updated, Ivy places all linked nodes into a stack. The stack creation algorithm ensures that each node precedes all other nodes found in its outbound tree of links. One problem facing data flow programming environments is how to specify the order of execution of parallel link chains. For instance, if two nodes emerge from a parent node, the order of execution could have consequences for nodes further along the chain. The current implementation places nodes in the execution stack following the creation order of their connecting links.

5.5.2 Data Flow Visualization

Ivy includes a feature for visualizing data flow between constructs. Data flow can be started, paused, or stopped at any time using the inspector tool. When the program is running, active data packets are represented by particle clouds that flow along each link (Figure 7; D1). The particles encode several properties of the data:

Source – Colour represents the data's original source sensor type, which allows users to trace each source's path through the program structure. Particle colours match the corresponding sensor types shown on the medallions (Figure 5a, D3).

Value – Data values are represented by particle size, with larger particles indicating greater values (D3). Values are scaled linearly based on a low- and high-value pair for each source sensor type.

Frequency – The update frequency of the system is shown by the volume of particles flowing through each link. Each time a function is executed, a burst of particles traverses the link over a duration of 1 second. If a function fails to produce a valid output, no particles are emitted, visibly indicating that there are errors (D1). Ivy currently processes all nodes in the stack at a frequency of 10 Hz.

5.5.3 Debugging Tools

Debugging tool suites are commonplace in many programming environments. In addition to starting and stopping the program and data flow visualization, Ivy's *inspector* tool allows users to *freeze* the data flow visualization in its current state. The *step* feature provides control over program execution, allowing users to step through program cycles at their own pace. Each press of the wand trigger passes execution to the next node in the stack, with the data flow following each step. The *sequence* feature is similar to the step feature, but delays the flow visualization at each step to make it easier for users to follow data flow along the logic chain. Lastly, the *in/out links* feature assists in the identification of linked nodes by highlighting inbound, outbound, or all connected links (Figure 8).

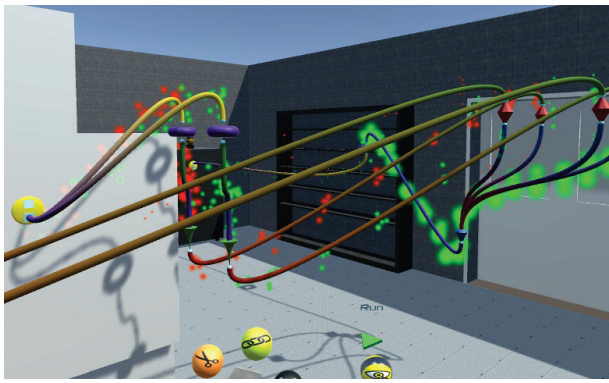

Figure 7: Data flow is visualized with particles of varying colour and size to depict the data source and changing values.

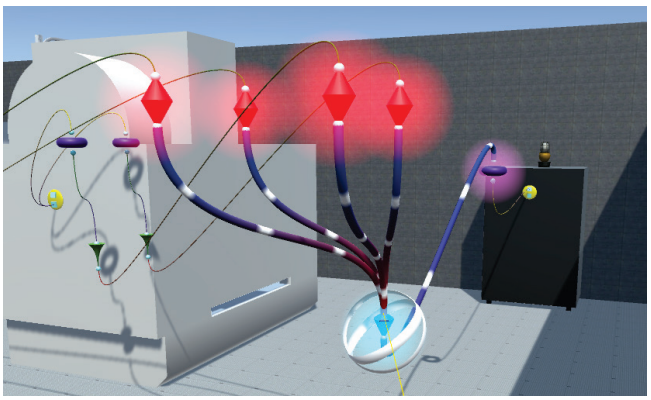

Figure 8: The selected blue node is highlighted with rotating rings, with visual emphasis on its connected links and nodes.

5.6 Real-world Sensing and Actuation

Ivy programs can connect to external environmental sensors and actuators. Sensor input originates from Texas Instrument SensorTags [56]. Each unit contains a 9-axis IMU, and sensors for ambient temperature, pressure and relative humidity. Data is routed through an online data broker, Dweet.io [15], and is polled once per second to retrieve updates.

If the user-defined program results in real-world objects changing states (e.g., lights turning on or off), a web-based REST-API reacts to state changes by controlling a USB-connected Phidget relay board [26] and logs the outgoing data to a web server.

6 Initial feedback sessions

To identify current limitations and opportunities for further exploration, we invited eight professionals (1 female, aged 29-45; Table 2) with expertise in IoT, Information Visualization, and Computer Graphics to explore Ivy's visual programming features. Participants had a range of programming experience, from novice to expert, and represented a wide spectrum of potential end-users. Aside from one first-time user, all had minimal to moderate experience with VR. Sessions lasted 45 to 60 minutes.

	Area of Expertise	*Job Experience (years)*	*IoT Knowledge*
P1	Building Information Modelling	5	4
P2	Program Manager	2	4
P3	Internet of Things	18	5
P4	Interactive Data Visualization	8	3
P5	Information Visualization	6	3
P6	Software Development	18	4
P7	Computer Graphics	12	3
P8	Building Information Modelling	6	3

Table 2: Participant backgrounds, experience. IoT knowledge was self-reported from 1 to 5 (Not at all to extremely knowledgeable).

6.1 Protocol

Participants underwent a brief training session to demonstrate Ivy's user interface and feature set and then were guided through a series of example programs.

Overview and Training (20-30 minutes) - Training took place in a virtual office environment and began with an introduction to the controllers and bimanual tool palette. Next, Ivy's basic node creation and link authoring features were demonstrated by guiding participants through the steps to create two simple programs, with live connections to real-world sensors and actuators.

Test Environments (15-20 minutes) - Users were then shown two example environments (described below) to demonstrate realistic application scenarios. After demonstrating the data flow visualization and debugging features, participants inspected programs to determine their basic purpose and were asked to locate and correct a small bug planted in one of the programs.

Feedback - A questionnaire collected information about each participant's profession and experience with IoT. Participants also answered Likert-based questions (scale 1-5) on the usefulness and various aspects of Ivy.

6.2 Example Application Scenarios

The first test environment was an industrial fabrication workshop. Accelerometer and gyro sensor readings from two machines were fed to triggers that monitored for high values. The trigger outputs were aggregated into a Boolean OR function, linked through an 'invert' function to the workshop's power supply port (Figure 9). This configuration triggered a power cut when excessive vibrations were detected to help reduce damage if a machine malfunctioned.

The second environment was a museum space, in which several foot traffic sensors were linked to a central aggregator node. The mean value was forwarded to a trigger node that activated a large exhibit when enough people were present (Figure 10). These scenarios were used because VR could potentially help managers or technicians monitor and program such remote, intelligent spaces.

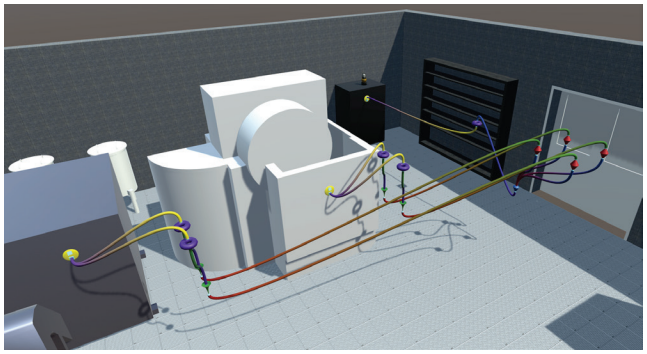

Figure 9: Example of a workshop with program links to automatically trigger a machine shut down.

Figure 10: Example of a museum with program links to trigger the power to a central exhibit based on foot traffic.

6.3 Observations and Feedback

From our observations, participants could quickly learn Ivy's features. Once a particular feature was shown, participants could select and operate the correct tool with little prompting, "*With only minimal instructions at the beginning I was able to carry on the tasks without too much supervision*" (P6).

Participants were generally receptive to Ivy and indicated cases where such a tool would be useful in their profession. P1 summarized, "*I can see this being very useful to gaining understanding of systems built by others or when trying to debug… A visual and physical interface for programming is much more accessible than standard coding*". Participants also recognized the utility of visual tools, in this context, for novice users: "*I think it would be very powerful as a fun, low-barrier entry point to visually programming IoT devices and behaviours*" (P4). Ivy's visual features helped most participants easily identify our planted 'bug', which caused the visual data flow to cease at the affected node.

Survey responses were positive about the virtual logic constructs (Q1; Figure 11) and features for visual debugging and data flow (Q4). P3 commented that the "*Visualization of data flow could help users better understand the logic they are establishing*", with P7 adding "*I believe stepping along a program, and visualizing it in situ… would be very useful to debug very complex programs*".

Participants had mixed opinions about the utility of VR for IoT authoring (Q5). Several identified specific benefits of VR, for example to make buildings "*accessible off-site (remotely) or prior to construction*" (P1). P3 found the spatial interface "*intuitive*" and P4 noted "*I think the immersive aspect really adds to the experience. On a desktop, I think 3D navigation would encumber the experience.*" However, some felt authoring applications in VR would be cumbersome: "*I like the possibilities with VR but feel like it is slower and less direct than the desktop experience would be*" (P2). Participants did, however, recognize the inherent trade-offs between platforms, noting that immersive authoring could likely be used in conjunction with traditional tools: "*In my profession I'd think that an interesting combination would be to have VR for debugging and development phase and also for troubleshooting remotely the actual system, while AR could be useful when deploying the system in the real world*" (P4).

Several participants provided comments about additional features they would like to see in a future version of Ivy, for example to encapsulate groups of constructs within sub-structures. Participants also said they would like to see a greater variety of visual encodings of information in the links and data flow visualization.

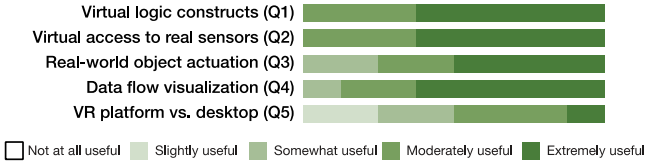

Figure 11: Participant responses to aspects of the Ivy system.

7 Discussion and Future Work

Our prototype development environment provides a proof-of-concept interface for spatially situated visual programming that leverages information about the spatial layouts of smart objects. Ivy demonstrates how the proposed guidelines result in a useful,

functioning programming environment. Initial feedback suggests that it is possible to create 3D programs with an immersive interface that is easy to use and applicable to both novices and professionals. Participants saw value in Ivy for practical IoT applications and welcomed further integration with AR and desktop components.

Our study revealed several limitations of the current system and areas of interest for future research. Participants quickly learned the interface and had little trouble applying the available features in the test scenes. However, we noticed limitations in the raycasting mechanism when selecting at far distances. Although the auto-complete feature was very effective for completing links, more sophisticated selection techniques are likely required for dense arrangements or large environments. Future solutions could potentially leverage the specific advantages of virtual worlds to scale and warp space. For instance links may be authored in a world in miniature view [61] to reduce the required motion, or distant sections of a room could be reconstructed nearby for direct access.

One issue raised by some participants was a need for further directional indicators in the program structure. Although we included elements such as top-to-bottom flow, animated directional markers on selected links, and visible data flow, some participants had trouble identifying the direction of a program before running or inspecting it in detail. Further developments could include modified construct shapes or more intuitive layouts.

We are eager to develop a more fully-featured system with an elaborate range of functions and pre-sets and to explore these in deployments with hundreds or thousands of objects. Questions remain, however, about how to scale Ivy to more complex, real-world situations. Some lessons can be borrowed from existing systems; for instance, whereas Max [42] addresses function scalability by relying primarily on text descriptors to differentiate objects, Scratch [50], NodeRED [6], and Wyliodrin [70] show the potential of organizing a large number of functions into a limited number of colours and shapes. Participants also indicated the need for features to encapsulate program structures into hierarchical elements; for example, once a program subroutine has been constructed, its visual representation can be replaced with a single element, such as a box, which can later be opened for editing. Such encapsulation features are commonplace in data flow languages such as Max, and help to create maintainable program structures.

Participants were generally very receptive to the spatial aspects of virtual logic constructs and visual data flow, but some were hesitant about the utility of a VR system. We attribute this in part to some participants' strong familiarity with desktop programming environments through their work (e.g. P6 with 18 years of desktop development experience). Conversely, other participants appreciated the potential of VR; for instance, P5, with a background in information visualization, commented that he was sceptical at first about the VR interface, but after using Ivy, saw strong potential in the visual and spatial nature of Ivy.

To follow up on this initial feedback, formal studies are needed to identify the specific benefits of a spatial interface for situated programming. These benefits may be more readily realized through the spatially situated advantages of an AR platform, which our VR system did not allow us to fully explore. We would also like to investigate the benefits of a collaborative, mixed-platform system that integrates Ivy with desktop tools for back-end development, along with new interaction methods to increase end-user control over function arguments and parameters. Such future research should also include the invention of new lightweight and portable control devices to provide portability for mobile workers using AR. Furthermore, while many of Ivy's current features may be adapted for an AR system, future work must investigate how to minimize visual clutter and prevent interference during complex real-world tasks [20].

In future work, it will also be interesting to explore the presented concepts in other domains, such as programming behaviours for immersive games, virtual architectural walkthroughs, or 3D storytelling applications.

8 Conclusion

This work introduced Ivy, a spatially situated visual programming environment that enables users to author IoT behaviours in a VR environment. An initial study on the benefits of an immersive VR platform and semantic object references demonstrated the merits of a spatial interface for such systems. More broadly, the work contributes a set of design guidelines for immersive visual programming and moves toward sophisticated spatial interface tools for authoring and understanding complex situated programs in mixed reality systems. These explorations highlight several potential benefits and limitations of such systems and allude to promising areas for future research.

REFERENCES

[1] 3DVIA Virtools. [Online]. Available: http://www.3dvia.com/products/3dvia-virtools/. [Accessed: 11-Apr-2016].

[2] R. Ball and C. North. The effects of peripheral vision and physical navigation on large scale visualization, in *Proc. GI*, pp. 9-16, 2008.

[3] O. Bau and W. E. Mackay. OctoPocus: a dynamic guide for learning gesture-based command sets, in *Proc. UIST*, pp. 37-46, 2008.

[4] B. Bell, S. Feiner, and T. Höllerer. View management for virtual and augmented reality, in *Proc. UIST*, pp. 101-110, 2001.

[5] M. Blackstock and R. Lea. IoT mashups with the WoTKit, in *Proc. IoT*, pp. 159-166, 2012.

[6] M. Blackstock and R. Lea. Toward a distributed data flow platform for the web of things (distributed node-RED), in *Proc. WoT*, pp. 34-39, 2014.

[7] A. F. Blackwell, K. N. Whitley, J. Good, and M. Petre. Cognitive factors in programming with diagrams, *Artif. Intell. Rev.*, 15(1–2): 95–114, 2001.

[8] D. A. Bowman, R. P. McMahan, and E. D. Ragan. Questioning naturalism in 3D user interfaces, *Commun. ACM*, 55(9): 78–88, 2012.

[9] Connect the apps you love - IFTTT. [Online]. Available: https://ifttt.com/. [Accessed: 16-Sep-2015].

[10] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-D tool for introductory programming concepts, in *Journal of Computing Sciences in Colleges*, 15: 107-116, 2000.

[11] M. F. Deering. The HoloSketch VR sketching system, *Commun. ACM*, 39(5): 54–61, 1996.

[12] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, *Hum-Comput Interact*, 16(2): 97-166, Dec. 2001.

[13] A. K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications, in *Pervasive Computing*, K. P. Fishkin, B. Schiele, P. Nixon, and A. Quigley, Eds. Springer, pp. 254-271, 2006.

[14] DOOM SnapMap. [Online]. Available: http://doom.com/en-us/snapmap. [Accessed: 18-Sep-2016].

[15] dweet.io - Share your thing like it ain't no thang. [Online]. Available: http://dweet.io/. [Accessed: 11-Apr-2016].

[16] Dynamo BIM. [Online]. Available: http:// http://dynamobim.org/ [Acessed: 18-Sep-2016].

[17] EBSCOhost | Sensor-enabled Cubicles for Occupant-centric Capture of Building Performance Data. [Online]. Available: http://connection.ebscohost.com/c/articles/67217601/sensor-enabled-cubicles-occupant-centric-capture-building-performance-data. [Accessed: 11-Apr-2016].

[18] N. ElSayed, B. Thomas, K. Marriott, J. Piantadosi, and R. Smith. Situated Analytics, in *Proc. BDVA*, pp. 1-8, 2015.

[19] B. Ens, R. Finnegan, and P. P. Irani. The Personal Cockpit: A spatial interface for effective task switching on head-worn displays, in *Proc. CHI*, pp. 3171-3180, 2014.

[20] B. Ens, E. Ofek, N. Bruce, and P. Irani. Spatial constancy of surface-embedded layouts across multiple environments, in *Proc. SUI*, pp. 65-68, 2015.

[21] Game Engine Technology by Unreal. [Online]. Available: https://www.unrealengine.com/. [Accessed: 11-Apr-2016].

[22] Gartner Says 6.4 Billion Connected. [Online]. Available: http://www.gartner.com/newsroom/id/3165317. [Accessed: 11-Apr-2016].

[23] M. Glueck, A. Khan, and D. J. Wigdor. Dive in!: Enabling progressive loading for real-time navigation of data visualizations, in *Proc. CHI*, pp. 561-570, 2014.

[24] R. Grasset, T. Langlotz, D. Kalkofen, M. Tatzgern, and D. Schmalstieg. Image-driven view management for augmented reality browsers, in *Proc. ISMAR*, pp. 177-186, 2012.

[25] Grasshopper. [Online]. Available: http://www.grasshopper3d.com/. [Accessed: 11-Apr-2016].

[26] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets, in *Proc. UIST*, pp. 209-218, 2001.

[27] A. Gunnarsson, M. Rauhala, A. Henrysson, and A. Ynnerman. Visualization of sensor data using mobile phone augmented reality, in *Proc. ISMAR*, pp. 233-234, 2006.

[28] S. Guven, S. Feiner, and O. Oda. Mobile augmented reality interaction techniques for authoring situated media on-site, in *Proc. ISMAR*, pp. 235-236, 2006.

[29] R. A. Herrera-Acuña, V. Argyriou, and S. A. Velastin. Toward a 3D hand gesture multi-threaded programming environment, in *Adv. in Vis. Inf.*, Springer, pp. 1-12, 2013.

[30] V. Heun. Reality Editor on the App Store. [Online]. Available: https://itunes.apple.com/au/app/reality-editor/id997820179?mt=8. [Accessed: 21-Dec-2016].

[31] V. Heun, J. Hobin, and P. Maes. Reality Editor: Programming smarter objects, in *Ubicomp Adjunct Proc.*, pp. 307-310, 2013.

[32] V. Heun, S. Kasahara, and P. Maes. Smarter Objects: Using AR technology to program physical objects and their interactions, in *Proc. CHI*, pp. 2939-2942, 2013.

[33] J. Humble *et al.* "Playing with the Bits" User-configuration of ubiquitous domestic environments, in *Proc. UbiComp*, pp. 256-263, 2003.

[34] F. Kawsar, E. Rukzio, and G. Kortuem. An explorative comparison of magic lens and personal projection for interacting with smart objects, in *Proc. MobileHCI*, p. 157-160, 2010.

[35] D. F. Keefe, D. A. Feliz, T. Moscovich, D. H. Laidlaw, and J. J. LaViola Jr. CavePainting: A fully immersive 3D artistic medium and interactive experience, in *Proc. I3D*, pp. 85-93, 2001.

[36] A. Khan and K. Hornbæk. Big data from the built environment, in *Proc. 2nd International Workshop on Research in the Large*, pp. 29-32, 2011.

[37] LabVIEW System Design Software. [Online]. Available: http://www.ni.com/labview/. [Accessed: 16-Sep-2015].

[38] G. A. Lee, C. Nelles, M. Billinghurst, and G. J. Kim. Immersive authoring of tangible augmented reality applications, in *Proc. ISMAR*, pp. 172-181, 2004.

[39] J. Lee, L. Garduño, E. Walker, and W. Burleson. A tangible programming tool for creation of context-aware applications, in *Proc. UbiComp*, pp. 391-400, 2013.

[40] F. C. Y. Li, D. Dearman, and K. N. Truong. Virtual Shelves: Interactions with orientation aware devices, in *Proc. UIST*, pp. 125-128, 2009.

[41] C. Liu, O. Chapuis, M. Beaudouin-Lafon, E. Lecolinet, and W. E. Mackay. Effects of display size and navigation type on a classification task, *Proc. CHI*, pp. 4147-4156, 2014.

[42] Max is a visual programming language for media. [Online]. Available: https://cycling74.com/products/max/. [Accessed: 16-Sep-2015]

[43] S. Mayer, Y. N. Hassan, and G. Sörös. A magic lens for revealing device interactions in smart environments, in *Proc. SIGGRAPH Asia*, pp. 1-6, 2014.

[44] M. Mine. Working in a virtual world: Interaction techniques used in the chapel hill immersive modeling program, Tech. Report, Univ. N. C., 1996.

[45] MIT Media Lab. Open Hybrid. [Online]. Available: http://openhybrid.org/. [Accessed: 18-Sep-2016].

[46] mOculus.io | VR-Plugin for Autodesk Maya. [Online]. Available: http://moculus.io/. [Accessed: 11-Apr-2016].

[47] M. A. Najork and S. M. Kaplan. The CUBE languages, in *Proc. 1991 IEEE Workshop on Visual Languages*, pp. 218–224, 1991.

[48] Node-RED. [Online]. Available: http://nodered.org/. [Accessed: 16-Sep-2015].

[49] F. Reeth, K. Coninx, S. Backer, and E. Flerackers. Realizing 3D visual programming environments within a virtual environment, in *Computer Graphics Forum*, 14: 361–370, 1995.

[50] M. Resnick *et al.* Scratch: programming for all, *Commun. ACM*, 52(11): 60–67, 2009.

[51] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications, in *Proc. CHI*, pp. 434–441, 1999.

[52] C. Sandor, A. Olwal, B. Bell, and S. Feiner. Immersive mixed-reality configuration of hybrid user interfaces, in *Proc. ISMA*, pp. 110–113, 2005.

[53] J. P. Schulze, C. E. Hughes, L. Zhang, E. Edelstein, and E. Macagno. CaveCAD: a tool for architectural design in immersive virtual environments, in *Proc. SPIE*, 2014, 9012.

[54] SeeControl. [Online]. Available: http://www.seecontrol.com/. [Accessed: 11-Apr-2016].

[55] Z. Shi *et al.* Digital Campus Innovation Project: Integration of Building Information Modelling with Building Performance Simulation and Building Diagnostics, in *Proc. SimAUD*, pp. 51–58, 2015.

[56] Simplelink SensorTag. [Online]. Available: http://www.ti.com/ww/en/wireless_connectivity/sensortag2015/. [Accessed: 11-Apr-2016].

[57] Simulink - Simulation and Model-Based Design. [Online]. Available: http://www.mathworks.com/products/simulink/. [Accessed: 11-Apr-2016].

[58] A. Steed and M. Slater. A dataflow representation for defining behaviours within virtual environments, in *Proc. IEEE VR*, pp. 163–167, 1996.

[59] R. Stiles and M. Pontecorvo. Lingua Graphica: a visual language for virtual environments, in *Proc. 1992 IEEE Workshop on Visual Languages*, pp. 225–227, 1992.

[60] Stingray Engine. [Online]. Available: http://stingrayengine.com/. [Accessed: 19-Sep-2016].

[61] R. Stoakley, M. J. Conway, and R. Pausch. Virtual Reality on a WIM: Interactive Worlds in Miniature, in *Proc. CHI*, pp. 265–272, 1995.

[62] Tilt Brush by Google. [Online]. Available: http://www.tiltbrush.com/. [Accessed: 11-Apr-2016].

[62] J. Vermeulen, J. Slenders, K. Luyten, and K. Coninx. I bet you look good on the wall: Making the invisible computer visible, in *Proc. AmI*, pp. 196–205, 2009.

[63] J. A. Walsh and B. H. Thomas. Visualising environmental corrosion in outdoor augmented reality, in *Proc. AUIC*, pp. 39–46, 2011.

[64] Q. Wang, T. Hsieh, and A. Paepcke. Piles across space: Breaking the real-estate barrier on small-display devices, *Int J Hum-Comput Stud*, 67(4): 349–365, Apr. 2009.

[65] C. Ware and D. R. Jessome. Using the bat: a six-dimensional mouse for object placement, *IEEE Comput. Graph. Appl.*, 8(6): 65–70, Nov. 1988.

[66] M. Weiser. The computer for the 21st century, *Sci. Am.*, 265(3): 94–104, Sep. 1991.

[67] S. White and S. Feiner. SiteLens: Situated visualization techniques for urban site visits, in *Proc. CHI*, pp. 1117–1120, 2009.

[68] K. N. Whitley and A. F. Blackwell. Visual programming: the outlook from academia and industry, in *Papers presented at the seventh workshop on Empirical studies of programmers*, pp. 180–208, 1997.

[69] Wyliodrin. [Online]. Available: https://www.wyliodrin.com/. [Accessed: 16-Sep-2015].