Immersive Visual Scripting of Gamified Training based on VR Software Design Patterns

Paul Zikas

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *George Papagiannakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Immersive Visual Scripting of Gamified Training based on VR Software Design Patterns

Thesis submitted by
Paul Zikas
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Paul Zikas

Committee approvals: _____
George Papagiannakis
Associate Professor, Thesis Supervisor

_____
Constantine Stephanidis
Professor, Committee Member

_____
Anastasios Roussos
Principal Researcher, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, October 2019

Immersive Visual Scripting of Gamified Training based on VR Software Design Patterns

Abstract

Virtual reality (VR) has re-emerged as a low-cost, highly accessible consumer product, and training on simulators is rapidly becoming standard in many industrial sectors. Combined with the continued advancements in VR technology, the interest in platforms that generate immersive experiences has increased. However, the available systems are either focusing on gaming context, featuring limited capabilities (embedded editors in game engines) or they support only content creation of virtual environments without any rapid prototyping and modification. In this thesis, we focused on authoring tools and systems that generate or modify VR content, utilizing visual node-based editors or directly support prototyping from within the virtual environment (VR Editors).

Thus, we propose an innovative coding-free, visual scripting platform to replicate gamified training scenarios through Rapid Prototyping via newly defined VR software design patterns. We implemented and compared three authoring mechanisms, a) classic scripting, b) visual scripting and c) VR Editor for rapid reconstruction of VR training scenarios based on our novel design patterns. Our Visual Scripting module is capable of generating training applications utilizing a node-based scripting system whereas the VR Editor gives the user/developer the ability to customize and populate new VR training scenarios directly from within the virtual environment. In addition, we present the scenegraph architecture as the main model to represent training scenarios on a modular, dynamic and highly adaptive acyclic graph. To the best of our knowledge, there is no similar system that generates VR training scenarios utilizing such visual and virtual scripting tools. Inspired from game programming patterns, we implemented new design patterns for VR experiences. Such custom action prototypes support a variety of commonly used interactions and procedures within training scenarios offering great flexibility in the development of VR metaphors.

Οπτικοποιημένη ανάπτυξη παιχνιδοποιημένων σεναρίων εκπαίδευσης βασισμένη σε σχεδιαστικά πρότυπα εικονικής πραγματικότητας

Περίληψη

Η εικονική πραγματικότητα εχει αναπτυχθει ιδιαίτερα τα τελευταία χρόνια εξαιτίας του χαμηλού της κόστους και της προσβασιμότητας που προσφέρει εδραιώνοντας παράλληλα τους εξομοιωτές εκπαίδευσης σε πολλούς τομείς της βιομηχανίας. Συνδυάζοντας τις καινοτομίες στον τομέα της εικονικής πραγματικότητας, το ενδιαφέρον για πλατφόρμες παραγωγής εκπαιδευτικών σεναρίων έχει αυξηθεί δραματικά. Ωστόσο, τα διαθέσιμα συστήματα επικεντρώνονται είτε σε εφαρμογές παιχνιδιών, με περιορισμένες δυνατότητες (ενσωματωμένα σε μηχανές παιχνιδιών) είτε υποστηρίζουν μόνο τη δημιουργία εφαρμογών χωρίς διαδραστικότητα. Σε αυτή την διατριβή, επικεντρωθήκαμε σε εργαλεία και συστήματα παραγωγής και τροποποίησης εικονικού περιεχομένου, χρησιμοποιώντας οπτικούς κόμβους επεξεργασίας ή απευθείας παρέμβαση απο το εικονικό περιβάλλον.

Παραθέτοντας τα παραπάνω στοιχεία, προτείνουμε μια πλατφόρμα οπτικοποιημένης παραγωγής παιχνιδοποιημένων σεναρίων εκπαίδευσης χρησιμοποιώντας ταχεία προτυποποίηση και σχεδιαστικά πρότυπα λογισμικού εικονικής πραγματικότητας. Δημιουργήσαμε και συγκρίναμε τρία εργαλεία παραγωγής περιεχομένου εικονικής εκπαίδευσης με χρήση α) κώδικα β) οπτικοποιημενου συντάκτη και γ) συντάκτη εικονικής πραγματικότητας βασισμένο σε σχεδιαστικά πρότυπα λογισμικού. Το οπτικοποιημένο σύστημα κώδικα που αναπτύχθηκε παράγει εκπαιδευτικά διαδραστικά σενάρια μέσα από διεπαφές χρήστη χωρίς την χρήση κώδικά. Ο συντάκτης εικονικής πραγματικότητας δίνει την δυνατότητα στους χρήστες/προγραμματιστές να παραμετροποιήσουν την εφαρμογή απευθείας από το εικονικό περιβάλλον. Επιπροσθέτως, παρουσιάζουμε μια αρχιτεκτονική βασισμένη σε δυναμικό γράφο ικανή να αναπαραστήσει το σενάριο εκπαίδευσης. Από όσο γνωρίζουμε, δεν υπάρχει παρόμοιο σύστημα που να δημιουργεί σενάρια εκπαίδευσης χρησιμοποιώντας οπτικοποιημένα εργαλεία ανάπτυξης. Εμπνευσμένοι από τα πρότυπα προγραμματισμού παιχνιδιών, υλοποιήσαμε καινοτόμες τεχνικές σχεδίασης αποκλειστικά για εμπειρίες εικονικής πραγματικότητας. Αυτά τα πρωτότυπα υλοποίησης υποστηρίζουν μια ποικιλία κοινώς χρησιμοποιούμενων διαδικασιών αλληλεπίδρασης προσφέροντας μεγάλη ευελιξία στην μεταφορά περιεχομένου στον εικονικό κόσμο.

Acknowledgements

I would first like to thank my supervisor George Papagiannakis for his tireless support, inspiration, and for believing in me while still undergraduate to join his team, where I have gained invaluable experience and met people and partners sharing the same passion for knowledge as I do.

Thanks to my co-workers: Nikos Lydatakis for his programming guidance and contribution to Action Prototypes, support in stressful moments and for always trying to draw a smile on my face. His advice to clear my mind and focus on the end goal proven to be life saving for this project. Steve Kateros, always available for brainstorming sessions on design thoughts and special thanks for his contribution to Storyboard Importer and Alternative Paths. Stelios Georgiou, raising the standards and motivate me to work harder no matter what. Efstratios Geronikolakis, research and academic partner, always has an available publication to share or review, special thanks for his design comments on VR Editor for helping me finalize the last programming part of this thesis. Michael Kentros, for countless brainstorming session, debates and strong arguments on theoretical and programming topics. Back in my hometown, special thanks to my fellowship: Orestis, Dimitris, Kuriakos and Kostas for always giving me the strength to keep going, especially during the final two semesters. I feel so blessed to have them by my side.

Finally, I would like to express my gratitude to my family for their support, sacrifices, motivation and love.

Explore, Experiment, Learn

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It was the year 1968 when Thomas Ellis, John Heafner and William Sibley launched GRaIL, a GRaphical Input Language [15], designed by and programmed at the RAND Corporation. Using a tablet with a stylus, the system would allow freehand input of letters, boxes, and lines, as well as corrections to previous drawings. The drawings that were input consisted of meaningful objects within a flow chart. It was used to make sophisticated programs that can be compiled and run at full speed, or stepped through with a debugging interpreter that can run the program at variable speeds.

> *"When I saw it, I felt like I was sticking my hands right through the display and actually touching the information structures directly."*
>
> – Alan Kay, *1987*

GRaIL was one of the first systems that featured a visual scripting method for the creation of computer instructions based on cognitive visual patterns. Since the advancement of software systems in complexity demands, the need for authoring tools increased to maintain an easy to use and productive environment for a wide range of users with different levels of experience. Programmers are trained to write code, but what about the artists, designers or even end users? Wouldn't be great if everyone can contribute to a project or develop a new one based on a coding-free platform without advanced programming knowledge?

An authoring tool encapsulates various functionalities and features for the development of a specific product. The software architecture of such system empowers user/programmer with the necessary tools for content creation following two basic fundamental rules: 1) limit inexperienced users with intuitive and easy to use tools (visual scripting, VR editors) but 2) provide advanced users with enhanced tools to extend the capabilities of the system. This ambiguity can be over-passed with an elegant solution: t**he rapid prototyping of a modular architecture following object oriented design patterns**. Prototyping consist of a circular pipeline focusing on different stages of problem solving. This process usually begins with the problem redefinition followed by the fuelled ideation, usability testing

and finally the focused development. **Rapid Prototyping provides the bridge from product conceptualization to product realization** and development in a reasonably fast manner without the fuss of complex programming and fixtures.

Intelligent technologies are reforming the game: Artificial Intelligence, Machine Learning, Big Data acquisition, Blockchain, Mixed Reality and many more are growing rapidly, offering new opportunities and solutions to unsolved problems, reforming rules and possibilities. The nature of work is evolving quickly since the advancement of such technologies along with the human-computer interaction following collaborative form of activities rather than physical and repetitive ones. Learning and education are equally important and mandatory to develop skills over a specific subject. They are closely related but they differ in context since education refers to the processes and top-down transmission of knowledge whereas learning has a much wider concept.

Virtual reality (VR) has advanced rapidly in the recent years, gathering more attention from the developers' community. VR extends the possibilities of currently used systems offering countless interactive possibilities and arousing the interest of general public. In more detail, VR is characterized by its ability to recreate highly immersive and interactive digital environments where user experiences another dimension of possibilities. As already known from scientific studies [25], the training capabilities of VR simulations offer skill transfer from the VR to real life, reflecting the educational aspect of such systems and proposing an assistive tool to fit in modern curricula.

1.1 Scope and Objectives

The main goal of this project is to implement and compare three different authoring mechanics a) classic scripting, b) visual scripting and c) VR editor for rapid reconstruction of VR training scenarios based on design patterns. In more detail, the proposed system facilitates a VR playground to recreate training scenarios using the developed tools and functionalities. From the developer's perspective, this system forms a Software Development Kit (SDK) to generate VR content, which follows a well-structured educational pipeline. After coding the training scenario, users would be able to run the exported simulation using a VR headset and a pair of compatible controllers. To the best of our knowledge, there is no similar system that generates VR training scenarios from visual scripting tools.

1.2 Achievements

The visual creation of gamified scenarios requires a carefully designed software architecture to enable the content creation through object-oriented abstraction and embedded functionalities. For this reason, the proposed system consist of different architectural modules that each one fulfills its own purpose and all of them construct a collection of tools for rapid prototyping of VR training scenarios.

In this thesis, the initial task was to define how to construct an educational process or a more complex training scenario. The key to this problem was the modular composition of complex behaviours from simplified elements (Lesson, Stage, and Actions) used to define a tree-like graph, which separates the educational process into steps. Each step is a node and each node is a programmable process. The division into distinct objects/behaviours characterizes a major part of this system. For VR content creation, three different systems were integrated to the main architecture to access the same functionalties using different methodologies depending on the situation. For rapid operation adaptation to variations, we implemented a sematic representation of VR experiences to replicate training scenarios in a directed acyclic graph. By prototyping commonly used patterns and techniques we managed to create a customizable platform able to generate new content with minimal changes. Inspired from game and software programming patterns, **we implemented new design patterns specifically for VR experiences** to support a variety of commonly used interactions and procedures within training scenarios offering great flexibility in the development of VR scenes.

Our system proposes three different ways - authoring tools to develop gamified behaviours in VR. The first one is through classic **C# scripting**. xploiting the prototyping capabilities of the system, the developer can design a training scenario from scratch using the Unity's scripting language. This methodology offers great freedom on the implementation of training steps since the developer has all the needed tools for content creation. The next method is the **Visual Scripting Editor**, offering a hierarchical visual representation of the educational process with nodes, edges, drop down menus and all the necessary tools for the creation of training scenarios. This method adds a visual abstraction to the training scenario from the ability to inspect the whole project following intuitive visual patterns and design mechanics. The third and final method for content creation is the **VR Editor**. This tool extends the capabilities of scripting, as we currently know it. Developers are able to recreate a training scenario directly from the virtual environment. This method is ideal to experiment with new ways of interaction and have a better perception of the virtual environment while developing each training step. The described methods are linked into a common scenegraph architecture forming an architectural block to transform ideas into scenarios for the virtual world.

As a pilot application, we developed a digital heritage training scenario regarding the restoration of an antique clock to test the capabilities of the system. This application, integrates all of the implemented features and it was generated using the visual scripting and VR Editor authoring tools.

Figure 1.1 reflects the main concept of our system: the generation of training scenarios through visual scripting, VR Editor and classic C# code, utilizing our platform and functionalities.

Figure 1.1: Generation of interactive VR training scenarios through VR Editor, Visual Scripting and C# code.

1.3 Overview of Dissertation

Over the next chapters we will analyze the components of scenegraph architecture, focusing on key functionalities like Action Prototypes, Alternative Paths and auto-generated scripts. In more detail, on chapter 2 we will present related work in the thematic areas of visual programming, authoring tools, the impact of VR in training education, serious games and many more. On chapter 3, we will begin with the implementation phase of our system and specifically we will present the scenegraph architectural model along with Action Prototypes, Alternative paths and the fundamental principles of developing VR experiences using our system. On chapter 4, we will present the Visuals Scripting tool, one of the most important achievements of this thesis, discussing its features and describing how developers can generate gamified VR experiences without writing a single line of code. Moving on to chapter 5, we will propose VR Editor, an authoring tool to visualize and manage scenegraph as well Action scripts directly from the virtual environment. In chapter 6, we will present the demo training scenario we developed using the proposed system. In chapter 7 we will analyze a user-based qualitative evaluation we conducted to test our system in real-life scenarios. Finally, in chapter 8 we will summarize our results and conclude with discussion over future work and possible upgrades.

Chapter 2

State of the Art

This section describes previous work on training and education in VR as well as similar projects and publications for visual programming.

2.1 The impact of VR in Training and Education

Since the advancement of VR technology, modern VR applications have been transformed into a rich learning experience leading to the increase of published educational applications in recent years. Technological tools used in education provide new opportunities to increase collaboration and interaction through participants and to learn by enjoying, making the learning process more active, effective, meaningful, and motivating [12],[2]. Collaborative VR applications for learning [20], studies for the impact of VR in exposure treatment [5] as well as surveys for human social interaction [39] have shown the potential of VR as a training tool. The cognitive aspect of VR learning is already known from conducted trials [18], [17]. However, existing platforms, such as Facebook Spaces, Bigscreen, VRChat, AltpaceVR, and Rec Room, do not sufficiently cover training and education. The existing VR platforms are mainly focused on social interaction, providing a virtual environment to meet and discuss with other users while supporting collaborative mini games for entertainment purposes. In addition, the majority of VR simulators primary provide training, neglecting the educational factor [16]. It is a common misconception to confuse the terms training and education: training refers to the acquisition of skills (cognitive or psychomotor) whereas education refers to the acquisition of knowledge and information.

Focusing on the educational factor, the use of VR for knowledge transfer and e-learning is now extended as the R&D grows around entire VR environments where the learning takes place [36]. Virtual Reality rapidly increases its potential and influence on e-learning applications and simulations by taking advantage of two basic principles: a) the immersion and b) the embodiment. In more detail, immersive environments are capable to present a realistic scenario as it is, as it would be in real life. For this reason it is mandatory to maintain high fidelity, and

advanced realism to enhance this particular factor. An immersive environment is capable to present a realistic scenario, almost identical to a real situation where the participant needs to react under certain circumstances and with the proper methodology. Visual surroundings, audio feedback and virtual characters [3],[40] are essential to recreate an immersive world. The importance of virtual characters is profound in VR environments, in a previous project [44] we developed a crowd simulation for mobile, low cost VR HMDs studying optimizations in the rendering pipeline. In addition, embodiment is equally important. To achieve this feeling, it is essential to approach the sense of self-location, the sense of agency and the sense of body ownership [29]. Nevertheless, the exploitation of immersive virtual reality has allowed to experience the same sensations towards a virtual body inside an immersive virtual environment as towards the biological body, and if so, to what extent [33]. **The connection of embodiment and learning is profound**. Studies have shown the impact of embodiment in learning for virtual multimodal environments [26], is capable to affect the skill transfer when is deployed realistically. Embodiment is of vast importance in collaborative e-learning and training scenarios since user is further motivated to participate and gain experience from a situation that needs particular actions if he is linked psychologically to the virtual avatar in the digital environment.

Figure 2.1: Virtual Reality has great impact on museums, galleries and art (image reference: https://jasoren.com/vr-in-museums/).

VR-based learning, serious games and gamification approaches for interactive learning events extend beyond simple technical and procedural skills. VR environments allows trainees to engage with a multidisciplinary groups and focus on individual as well as team-based cognitive skills including problem solving, decision-making, and team behaviour skills within a realistic, reactive virtual environment. These concepts are essential to develop an educational curriculum [28] to enhance knowledge and skill transfer from the virtual to the real world. The learning capabilities of VR have great potential from surgical simulations

[42], [41], to supplementary material for learning courses [9]. Recently (September 2018), Walmart[1] announced they will purchase 17.000 Oculus Go headsets in its US stores for training their employees in demanding situations. Walmart already uses VR in its 200 Academy training centres running more than 45 modules, simulating events that would be difficult to run as physical training scenarios, like a Black Friday shopping rush.

As it seems, VR technology facilitates mixed-ability learning, knowledge transfer, and help participants to interact and collaborate fruitfully through different modalities. Research has shown that immersive Virtual Environments (VEs) are beneficial for training motor activities and spatial activities [51]. The concept of "Presence" refers to the phenomenon of behaving and feeling as if we are in the virtual world created by computer displays [49]."Presence is an incredibly powerful sensation, and it's unique to VR; there's no way to create it in any other medium. Most people find it to be kind of magical" It is not the same as "Immersion", where the user is simply surrounded by digital screens [1]. VR technology is capable to enrich the immersion and the embodiment factor by presenting a realistic multisensory virtual environment.

2.2 Serious Games and Gamification in VR

The appeal of VR digital games arouses interest among researchers and education specialists who since their recent proliferation, they have been trying to introduce their motivating potential in learning contexts. Serious games are complete games with serious intentions and designed accordingly whereas in gamified application only certain elements from games are used. Game-based learning also involves the incorporation of games into lessons. The main goal of applying games in education is to increase student's engagement and motivation. Serious games are closely related with gamification elements to enhance the education material with interesting, appealing and motivate features. Gamification is the application of game-design elements and game principles in non-game contexts. Examples of gamified experiences can be seen in mainstream applications for everyday activities: health trackers, fitness applications and personal calendars are the most common examples of applications with embedded gamification features.

However, does gamification work? Juho Hamari et al. published a survey [22] of 24 empirical studies on gamification trying to answer this question. They also create a framework for examining the effects of gamification by drawing from the definitions of gamification and the discussion on motivational affordances. Another more technical approach [55], presented two gamified experiences, one in VR and one in AR with the same content: the Minoan Palace of Knossos in Crete, Greece discussing the differences of gamification techniques between augmented and virtual reality. Sergi Villagrasa et al. [54] described the use of gamification and virtual reality-enhanced learning in university engineering and architecture

[1]https://www.walmart.com/

classes. This study focuses on gamification for new technologies and assistive tools for e-learning in a modern educational environment. For this purpose, they developed GLABS, a management assistive tool for gamification in the classroom and studied the impact on the learning curve. They conclude that the he use of gamification in a classroom increase the engagement and the motivation of students when compared with traditional methods.

2.2.1 Authoring tools for content creation

The main concept behind authoring tools is to develop a platform capable to generate content with minimal changes each time. This procedure speeds up the content creation while improving product maintenance.

The M.A.G.E.S. platform [41] proposes an SDK to deliver psychomotor VR surgical training solutions. The system provides software tools to generate orthopaedic VR simulations with minimal adaptations. Following a similar pattern, BricklAyeR [53] is a collaborative platform designed for users with limited programming skills that allows the creation of Intelligent Environments through a building-block interface. Another interesting project is ARTIST [31], a platform which provides methods and tools for real-time interaction between human and non-human characters to generate reusable, low cost and optimized MR experiences. Its aim is to develop a code-free system for the deployment and implementation of MR content, while using semantically data from heterogeneous resources. In addition, #FIVE [6] and #SEVEN [11] propose two frameworks for the development of interactive and collaborative virtual environments through a collection of embedded tools.

Another authoring tool, ExProtoVAR [45] generates interactive experiences in AR featuring development tools specially designed for non-programmers, without necessary a technical background with AR interfaces. The MASCARET model [7] proposes the organization of interactions between agents to enable cognitive and social abilities in virtual environments. This tool focuses on pedagogical aspects presenting a model for the generation of an intelligent tutoring system. The HUMANS platform [32] is a framework designed to export adaptive virtual environments used from both technological and domain experts. In addition it monitors learners actions through the analytics system to detect errors and propose solutions. Finally, in the field of interactive storytelling, StoryTec [21] platform facilitates an authoring tool to generate and represent storytelling-based scenarios in various domains (serious games, e-learning and training simulations). The platform aims to standardize the content creation of storytelling experiences following a descriptive format.

2.3 Visual Programming as an authoring tool

Visual programming is getting more publicity as more platforms and tools are emerging to enlarge the community. In recent years, many different approaches

emerged in the field of visual programming all focusing on different aspects and variations according to their use. We will separate them into two categories according to their visual appearance and basic functionalities: a) visual languages that utilize **blocks** as their way to represent methods, instances and patterns and b) **node-based** scripting languages where nodes are linked together with edges instead of merging or nesting together like blocks. In the following paragraphs we will present mentionable projects from the two categories of visual scripting systems.

2.3.1 Block-based visual languages

In more detail, Block-based visual programming has become increasingly popular with students of all ages, as it makes programming structures much more intuitive. It consist of modular blocks that represent fundamental programming utilities (if else, while, for loops etc.) or even custom prototypes that describe more complex functionalities. Users can generate programming patterns and sequences by connecting the blocks together forming a program. It is common for block-based visual languages to have different connection points like puzzle pieces to point that each block can fit only with specific type of blocks, underling in that way the syntactic rules of the language.

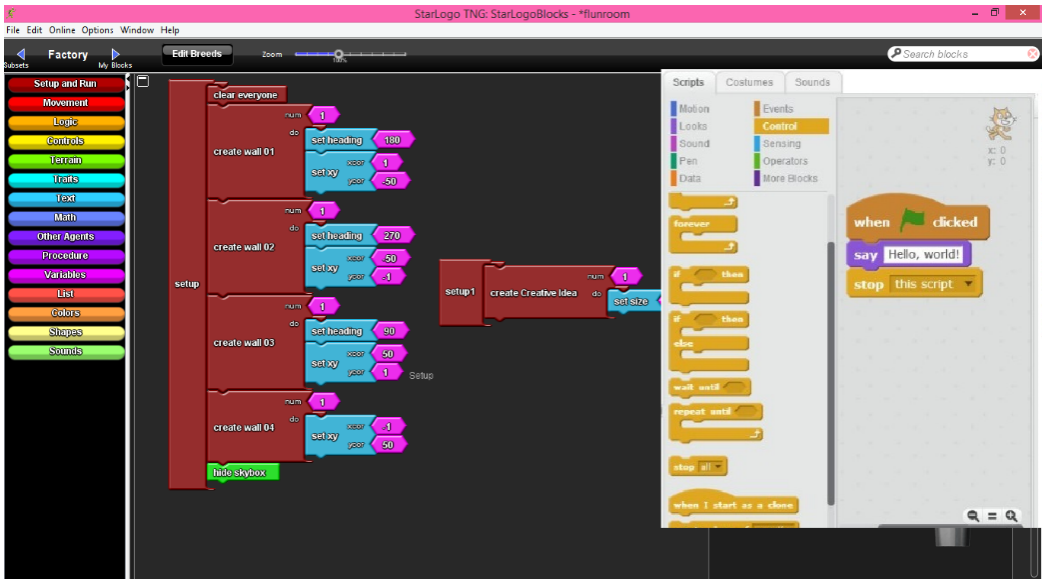


Figure 2.2: Examples of block-based programming languages (Left) StarLogo, (Right) Scratch

As an example, the OpenBlocks platform [47] proposes an extendable framework called that enables application developers to build their own graphical block programming systems by specifying a single XML file. Google's online visual scripting platform Blocky [43] uses interlocking, graphical blocks to represent code

concepts like variables, logical expressions, loops, and other basic programming patterns to export blocks to many programming languages like JavaScript, Python, PHP and Lua. Another approach from MIT is StarLogo [30], a client-based modeling and simulation software which facilitates the generation and understanding of simulations of complex systems. StarLogo utilize 3D graphics, sounds and a block based interface to facilitate as a programming tool for educational video games. Finally, another interesting approach is the Scratch [34] visual programming language and environment, which primarily targets ages 8 to 16 offering an authoring tool to support self-directed learning through tinkering and collaboration with peers.

To conclude, the majority of block-based languages were designed for younger ages leading to simple UIs and limited functionalities focused on programming fundamentals. They serve a great tool to learn basic programming, however if we isolate the block-based design, it does not scale up to more complex projects.

2.3.2 Node-based visual languages

Another design method of visual scripting systems represents programming elements using nodes instead of blocks. Modular programming can be visualized in that way as nodes can be generated and linked together with edges, forming complex patterns. **The principle of creating a complex structure out of basic elements is what makes visual scripting an ideal authoring tool for rapid prototyping**. Utilizing node-based programming, users can manipulate and link nodes that each perform a specific task and returns a corresponding output according to the node's input. The links between the nodes represent the flow of data from one node to another. The resulting structure looks like a directed graph that provides users with a visual overview of the data and program flow. By inspecting the data received and returned at each node, users can examine the functionality of their scripting nodes.

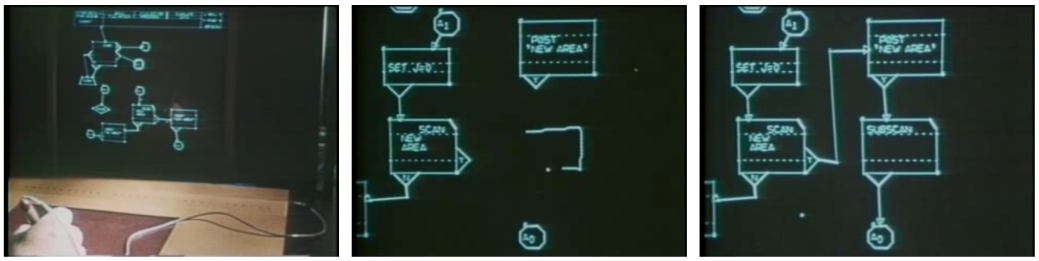


Figure 2.3: Pictures of GRaIL featuring modular programming.

GRaIL can be considered one of the first modular, node-based, visual scripting languages ever designed fifty years ago. Back to the present, node-based visual scripting tools are often designed to improve the user experience offering an easy to use system for various purposes. In is common for such systems to be released as a

supplementary tool to support existing platforms. As an example, Unreal engine supports the Blueprints Visual Scripting system as a game programming scripting system based on the concept of using a node-based interface to create gamified content from the Unreal Editor. With this system, even non-programmers can design a simple game. In addition, Blueprint-specific markup available in Unreal Engine's C++ implementation enables programmers to create baseline systems that can be extended by designers. Moving on to another game engine, Unity has recently released a visual scripting editor for building graphics shaders with programmable nodes within an intuitive UI. In Unity, shader programs are written in a variant of HLSL language (also called Cg) which is quite complex especially for users that are not familiar with computer graphics. This visual scripting editor offers an easy to use tool for writing shaders with limited programming knowledge, which is highly important since game engines are used by a variety of users with different technical skills and experience. On top of that, Unity recently announced a new visual scripting editor for game programming, not only for creating shaders, as a part of their standard s/w plugins.

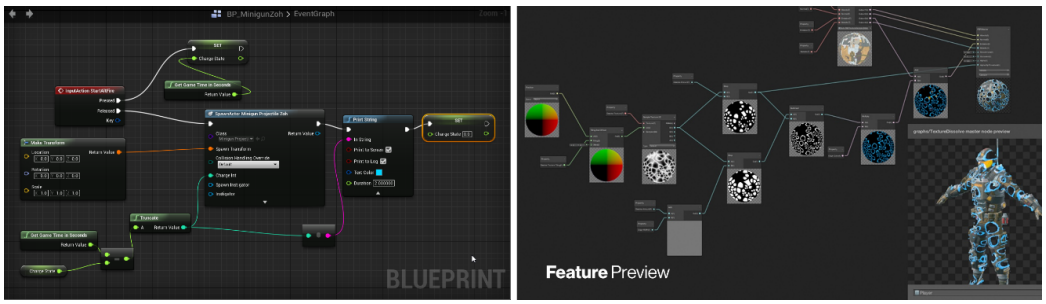


Figure 2.4: (Left) Unreal's Blueprint, (Right) Unity's shaders visual scripting.

Except from the shader visual scripting tool for Unity, additional tools are available from third parties as plugins at the asset store. Among them Flow Canvas, BOLT and Playmaker are the most popular ones offering many functionalities and features for developing gaming content. Each one of them provides a high-level code interface, enables decoupled systems and provides new self-contained nodes for designers with an easy and well-documented API. However, those tools were designed for general-purpose programming, thus they are not very friendly to modifications and prototyping.

Node-based systems can form a powerful tool when used correctly. They encapsulate complex systems, propriety and critical code to give end user or the programmer a better visualization of schemes, data structures or they even serve visual scripting capabilities like the system presented in this project. However, not all systems are suitable to support a visualization tool simply because there are too complex or the opposite; too simple. It is not good practice to increase complexity to systems that they work brilliant and do not need any modification. Keeping things simple is always the best solution. For example, figure 2.5 illustrates a chaos

of nodes and edges without any chance to understand the correlation of objects and logical links between them.

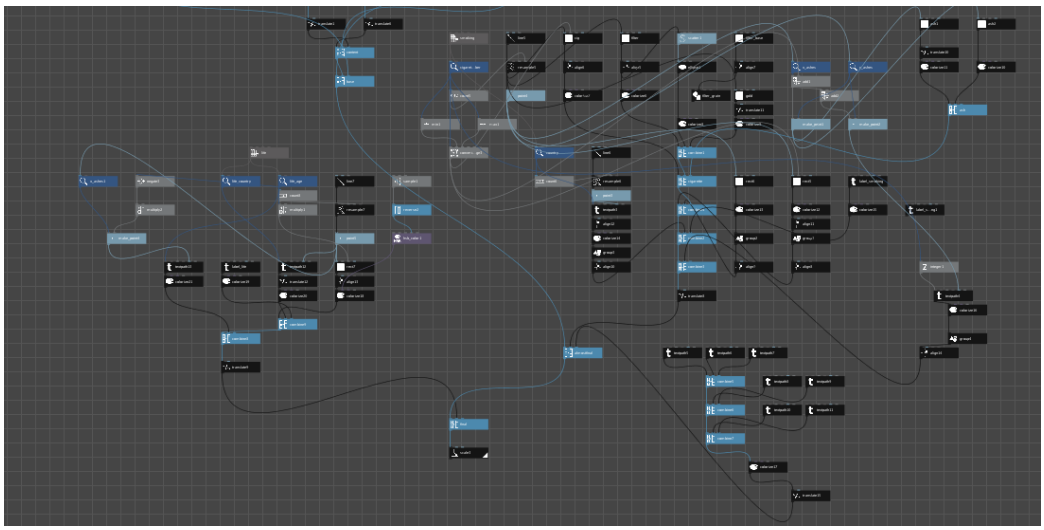


Figure 2.5: Example of unhealthy node-base visualization.

2.3.3 Unity Node Editor Base (UNEB).

The demand for visual scripting is high since modern game engines are widely available to designers, sales persons and non-programmers in general. For this project, the used base framework for visualizing nodes, edges and graphs was the open source plugin *Unity Node Editor Base (UNEB)*[2] . This framework provides basic node rendering and management functionalities within the Unity's editor. Main features of UNEB:

- Editor view with panning, zooming, and grid background.
- Save System using Scriptable Objects.
- Customizable GUI.
- Create, delete, and drag nodes and connections between them.

The modular scenegraph system was built on top of UNEB to inherit a graph engine system suitable for the development of a custom graph editor. UNEB has an active community of developers supporting and upgrading the platform via the Unity's forum and GitHub.

[2]https://unitylist.com/p/dqo/Unity-Node-Editor-Base

2.3.4 Editing directly from the VR environment

The development of authoring tools in virtual reality systems led to the integration of sophisticated functionalities from within the virtual environment. One of them is the implementation of VR Editors as an embedded feature for rapid creation of digital worlds directly from the virtual environment.

In SIGGRAPH 2017, Unity technologies presented EditorVR, an experimental scene editor which encapsulates all the Unity's features from within the virtual environment giving developers the ability to create a 3D scene while wearing a VR headset. During the live demo, Amir Ebrahimi and Matt Schoen from Unity Labs presented the building tools of EditorVR including features for initially laying out or modify a scene in VR, making adjustments to components using the Inspector workspace and building custom tools. The same year, Unreal Engine announced VR Mode, following similar principles. VR Mode enables to design and build worlds in a virtual reality environment using the full capabilities of the Unreal Editor Toolset combined with interaction models designed specifically for VR world building.

Except from game engines, VR editors started emerging into other software sectors like model editing. MARUI[3] is a plugin for Autodesk Maya that lets designers jump right into the virtual scene and perform modelling and animation tasks. MARUI 3 [35] claims that it not only allows designers to work comfortably with unlimited workspace and freedom of posture, but it can also can reduce the production costs up to 50%. Another noticeable project is RiftSketch [14], a live coding environment built for VR, which allows the development and design of 3D scenes from VR. RiftSketch proposes a hybrid XR system utilizing an external RGB camera and a leap motion sensor to record live footage from the programmer's hands while writing code and project this image into the virtual environment. Another software, eyecadVR [52] , proposes a VR editor for architecture design and scene management, a professional solution for architects to visualize and create their project while being immersed into the virtual world.

Figure 2.6: Examples of VR editors embedded in modern game engines (Left) Unity and (Right) Unreal.

[3]https://www.marui-plugin.com/

The currently available VR editors feature scene management capabilities with intuitive ways to build a scene directly from the virtual environment. **Their functionality is limited to moving virtual assets** from within the 3D scene claiming the immersion is beneficial for the world creation. **However, there is no other authoring tool in the market that offers a complete system for developing a behavioural VR experience including both the design and the programming aspect.** The proposed VR Editor offers scene management capabilities accompanied with the VR metaphor of the authoring platform where both users and programmers are able to generate training scenarios in a coding-free environment.

Expanding in areas beyond gamified VR applications, VR editors for software development and visualizations started emerging in the recent years. Primitive[4] is a startup company funded $1.1 million by Vive X fund, a venture capital accelerator from HTC, which last year (2018) released a promising demo video featuring 3D visualizations of source code that can be collaboratively explored and analysed in VR. During a presentation in San Francisco, Primitive founder John Voorhees pitched about the importance of software in our century underlying the difficulties in collaboration created by distributed teams who may not work in the same building or even country. Given the distributed nature of large-scale software development these days, much of the challenge is in figuring out how to keep everyone on the same page, he said. Primitive aims to the visualization of how complex object oriented code interacts with different modules within the same system. Users wearing VR headsets, can select any part of the code and explore the interconnections and data traversal between structures and classes. Primitive can also visualize how the code behaves when running and it is also possible to trace back complex multithreaded applications to understand how they work.

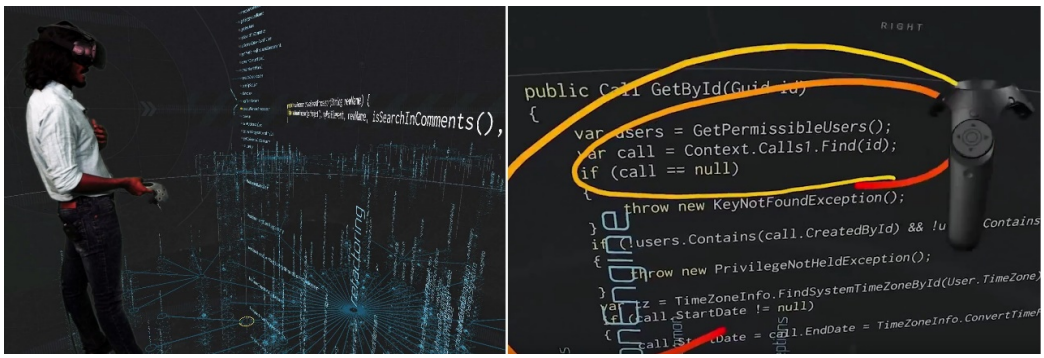


Figure 2.7: Primitive released demos of its VR code visualization program and plugins for IDEs.

It is common in game development to make small adjustments into an already finished product to improve user experience, (for example if an interactive object

[4]https://primitive.io/

is out of reach) or even under a client's suggestion that will make the application more convenient for them. In this situation, the standard way of completing this task is by tries and errors to achieve the desired object location. Focusing in VR applications, the setup of the virtual environment is a critical task since all the interactive objects need to be in specific positions to avoid game breaking bugs where an object is in a weird position or behind something else and thus out of reach of users. Those challenges can be solved with the use of a VR Editor tool, since developers are building and interacting with the game assets directly within the virtual environment using the controllers that ta application is meant to be played and not a mouse/keyboard on a screen replicating a 2D metaphor of a virtual reality experience. To make things more interesting, end users can modify the application on-the-go using the VR Editor without contacting the developer. This feature evolves the application into a highly adaptive environment where everyone can make changes and modify the content in their preferred way without any programming knowledge.

2.4 Dynamic programming languages

Dynamic programming languages define a class of high-level programming languages, which, at runtime, execute many common programming behaviours that static programming languages perform during compilation. These behaviours are mainly the addition of new code, extension of certain objects and modification of type systems. This project was developed using C# a classic dynamic language by Microsoft. The .NET Framework introduced the Code Document Object Model (CodeDOM) mechanism that enables developers to generate source code at run time, based on a compile object that represents the code to render. CodeDOM elements are linked to each other to form a tree data structure known as a CodeDOM graph. This structure represents the source code of our program. From the CodeDOM graph we can acquire realtime valuable information for our elements and types which otherwise was impossible. For the needs of the project, we utilized CodeDOM to auto-generate the Action scripts from the Visual Scripting and VR Editor tools. At this point, we will mention the differences between managed and unmanaged code since we will refer to them in the following chapters due to the reflection unit we integrated in our system:

Managed code is not compiled to machine code. Instead, it is compiled to an intermediate language, which is interpreted and executed by a separate service on the machine. It is operating within a secure framework, which handles memory issues and other runtime threads. Managed code is written in high-level languages run on top of .NET like C#, F#, etc.

Unmanaged code is compiled to machine code and therefore executed by the OS directly. It therefore has the ability to do powerful things Managed code does not. Applications that do not run under the control of the Common Language Runtime CLR are unmanaged, like C/C++.

2.5 Software Design Patterns for Visualization

In software engineering, design patterns describe how to solve recurring design problems to design flexible and reusable object-oriented software. Over the years, software architecture patterns and interfaces evolved to provide elegant solutions to specific challenges and tasks. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides broadly known as "Gang of Four" or "GoF" published the book "Design Patterns: Elements of Reusable Object-Oriented Software" in 1994 describing software design patterns. It has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice.

> *"Program to an 'interface', not an 'implementation'."*
>
> – Gang of Four, *1995*

Information visualization is an important aspect of data management systems and software tools developed to arrange a multitude of data. The visualization techniques are used to provide feedback on specific analysis problems. The software structure of such systems should be developed in a way to support the communication between different modules and form an architectural model capable to optimize data retrieval and management.

Nevertheless, why we need to apply design patterns in software development? Gamma et al. [13] described the importance and purpose of design patterns and ranked them into **creational**, **structural** and **behavioural**. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioural patterns characterize the ways in which classes or objects interact and distribute responsibility. Table 2.1 lists the three categories with example patterns.

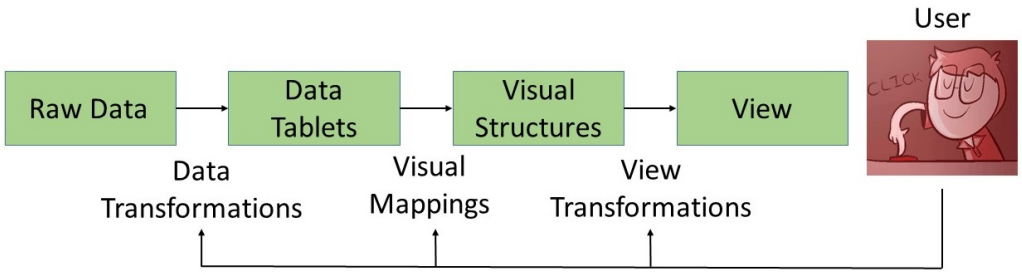


Figure 2.8: Information Visualization reference model.

A widely used reference model pattern in the area of information visualization described by Card et al. [8] and it is widely used by many visualization toolkits. The InfoVis reference model as he described it represents a pipeline of data transformations that begins with raw data and continues through different stages to finally come to visual representations and user interactions. The Raw data

Table 2.2: Design pattern space (Gamma et al.)

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chair ofResponsibility Command Iterator Mediator Memento Observer State Strategy Visitor

that is collected through analysis is transformed into data tables. Data tables are the building blocks within this model. In the next step, the visual structures are created depending on the kind of visual object we want to use to display the data. The visual abstraction will contain the definition for the properties such as shape, colour and position of the graphical object. The final step transforms the visual structures to what is called the "view" which user can interact with an interface and manipulate the extracted data. User has the freedom to modify at any intermediate step of the data, data tables or the visual abstraction process.

The visualization process has a strong correlation with design patterns and the literature contains various examples with different visual toolkits for data management. J. Heer and M. Agrawala presented a series of design patterns [23] for the domain of information visualization discussing their structure, context of use, and interrelations of patterns spanning data representation, graphics, and interaction. They also implemented the mentioned design patterns and data management techniques in the Prefuse toolkit [24], a software framework for creating dynamic visualizations of both structured and unstructured data. Prefuse, does not provide only InfoVis nodes that can be used like textboxes and buttons but provides a set of fine-grained building blocks for constructing tailored visualizations.

2.6 Rapid Prototyping

Code reusability and prototyping are two major principles in software architecture. The structure of software systems and the communication between its modules is described in an abstract way in terms of software design patterns. Software design patterns are reusable solutions for common design problems that often occur during software development. A strategically engineered system improves

product development in multiple ways and most important **makes the design process less sequential** than before.

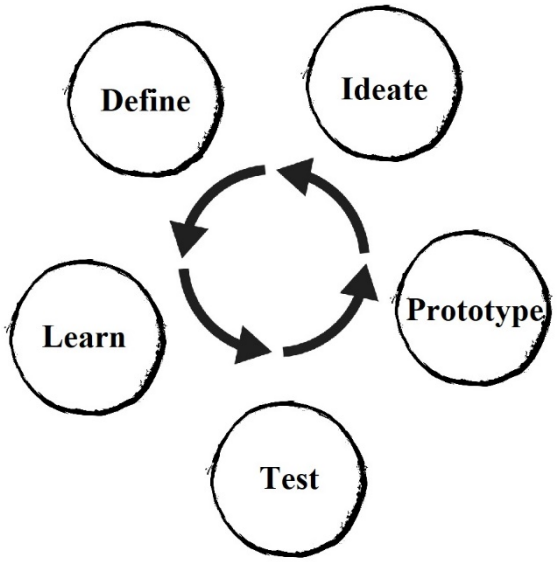


Figure 2.9: The Prototyping loop.

Prototyping hides the complexity of making new instances from the mother object. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when the object creation is a heavy process. Mark Giereth and Thomas Erlt [19] described three design patterns for rapid visualization prototyping: a) a mapping of object oriented models to relational data tables used in many visualization frameworks b) a script based approach for the configuration of visualization applications and c) performing online changes on the visual mapping by enhancing fine-grained mapping operators with scripting capabilities.

A virtual prototyping system that integrates VR with Rapid Prototyping to create virtual or digital prototypes to facilitate product development is described in [10]. Combining VR with Rapid Prototyping can result to a powerful tool for testing and evaluating new products and ideas before being employed in practical manufacturing, preventing costly mistakes, decreasing time to market, and meanwhile increasing worker safety. In his book J. Rix et al. [46] discussed the importance of Virtual Prototyping from the applications point of view. Among others, he underlined that developing virtual prototypes and integrate this technology to the product development process promises major advantages for the industrial process such as the reduction of time, saving cost and the increase of

quality. In addition, he states that rapid product development and virtual prototyping are fast becoming commodities of worldclass companies as a solution to maximize their effectiveness.

2.7 Interactive VR Environments

A virtual environment consist of digital information in the form of assets, avatars, objects and visual surroundings. To communicate and manipulate those digital assets, VR applications need interaction facilities to support this connection with the virtual world. The first generation of mainstream HMDs (Oculus DK1, 2) had limited interaction capabilities due to the lack of a robust input system. Most applications were using a controller connected to the computer or directly the keyboard. As a result, the majority of VR applications were forced to present short animated stories or virtual experiences with no interaction. However, since hand tracking became popular, Leap Motion[5] launched Orion, an update focused specifically for VR HMDs. It was a major change to the research development as VR applications increased their interaction capabilities dramatically. Users were able to handle and manipulate digital objects using their hands with natural gestures. Nathan Beattie et al. presented a platform [4], which employs the Oculus Rift Head Mounted Display (HMD) and Leap Motion Controller (LMC) to provide a low cost method enabling users to use their hands to dissect a mechanic model to manipulate and inspect individual components in realistic 3D. Another interesting approach [27] explored the feasibility of adapting the Leap Motion Controller to neurorehabilitation of elderly with subacute stroke. Following the launch of Orion SDK, Leap released several demo applications featuring the new features of Leap Motion Controller, among others a paint simulation, a game for interacting with particles and a cat's anatomy visualization.

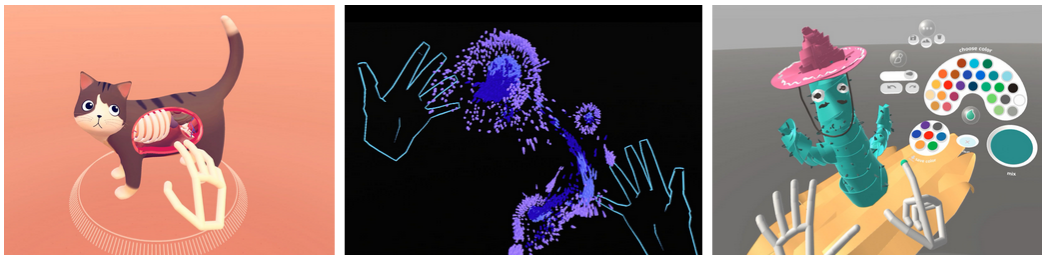


Figure 2.10: Three applications launched by Leap to support the Orion project.

Nonetheless, hand tracking had major stability issues. It was a common issue for applications using this technology to suffer from hand jittering or even hands were disappearing from the scene causing interaction malfunctions. The next generation of VR headsets (Oculus Rift, HTC VIVE) introduced specially designed

[5]https://www.leapmotion.com/

controllers for the devices. The controllers supported 6DOF due to the tracking technology from external cameras located in front of the user for standing mode or at the corners of the playing area for roomscale experiences. It was a gamechanger and the first time a robust input system integrated to VR offers great intractability and immersion. Following the launch of native controllers, VR companies also released games/demos that are still considering state-of-the-art for their interactive capabilities, originality and ease of use (The Lab[6] , Robo Recall[7] , Tilt Brush[8] , NVIDIA VR Funhouse[9] , Dead and Buried[10]).

Following the release of native VR controllers, new integrations for interacting with the virtual assets came out as Open Source Software Development Kits. A characteristic example of those integrations is NewtonVR[11] , developed by Keith Bradner and Nick Abel launched with the catchy slogan "Physics to the max!". It currently supports both Oculus SDK and SteamVR with a simple player controller. It allows users to pick up, throw, and use objects while taking advantage of PhysX so held objects do not go through each other or walls. The interaction-physics engine of NewtonVR is Velocity based which means that interaction is not relied on parenting objects to user's hand but the objects are connected to the hands according to their current velocity. This approach gives the sensation of a more natural movement than the parenting mechanism. NewtonVR features a plugin for Unity engine with an active community and a significant number of successful VR applications. After the success of NewtonVR, Keith Bradner moved to VALVE to upgrade the interaction and physics engine of SteamVR following the same principles from his previous projects.

Interaction on VR came a long way to stand at the point it is nowadays, but what comes next? VALVE recently launched the newest version of their upcoming "Knuckles" VR controllers, loaded with sensors. The company states they will make VR more immersive by bringing more of user's natural hand dexterity into the virtual world. In addition to being able to sense which buttons or sticks are being touched, similar to Oculus' Touch controllers, the handle of Knuckles has capacitive and force sensors inside which allow the controller to detect full finger movement and even grip strength. In addition, Moondust 3.0, an opensource collection of tech demos exploring the uses of Valve's Knuckles controllers, released on March 2019 offering to the community tools to develop content for Knuckles.

Each VR company focuses on different aspects of the same interaction challenge: Oculus is working on sophisticated controllers to increase the immersion through finger tracking and multisensory h/w, VIVE is paving the way towards room scale VR with trackers to reach the embodiment and support a robust system. At the same time, Oculus will soon release Oculus quest, an inside out,

[6]https://store.steampowered.com/app/450390/The_Lab/
[7]https://www.epicgames.com/roborecall/en-US/home
[8]https://www.tiltbrush.com/
[9]https://store.steampowered.com/app/468700/NVIDIA_VR_Funhouse/
[10]https://www.oculus.com/experiences/rift/1198491230176054/
[11]http://www.newtonvr.com/

untethered headset capable for free motion tracking without the need for cameras and extra sensors. It is a fast pace race for VR market dominance, the optimal stance from a developer is to make the most out of each solution and continue implementing on different platforms until a product becomes mainstream.

2.8 Our publications related to this work

In this section we present a short overview of our previous publications related to this work

- **Transforming medical education and training with VR using M.A.G.E.S. [41]:** In this work, we proposed a novel VR s/w system aiming to disrupt the healthcare training industry with the first Psychomotor Virtual Reality (VR) Surgical Training solution. We delivered an educational tool for orthopedic surgeries to enhance the learning procedure with gamification elements, advanced interactability and cooperative features in an immersive VR operating theater.

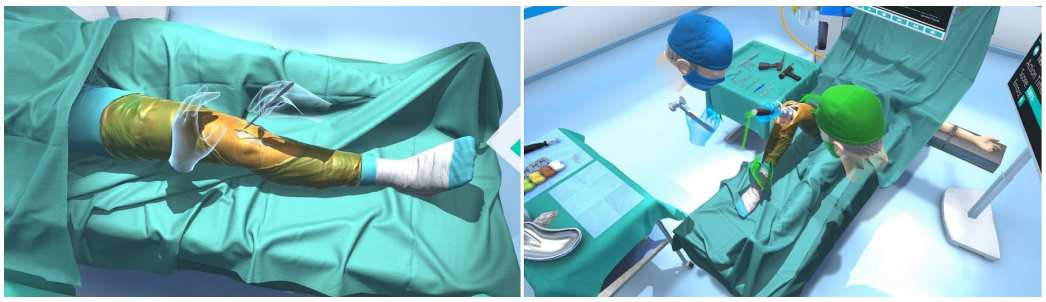


Figure 2.11: Different medical scenarios and functionalities of our system: a) Initial incision in Total Knee Arthroplasty simulation based on the medial parapatellar approach, b) Cooperative Total Knee Arthroplasty in which the main surgeon inserts the femoral implant.

- **Mixed reality serious games and gamification for smart education [55]:** In this project, we developed two mixed reality serious games featuring the palace of Knossos in Crete. The first, was an AR application using the Meta AR glasses: a holographic-AR, tethered headset, ancestor of Microsoft HoloLens, introduced novel interactive features with gesture manipulation and holographic projection. The second application featured a mobile VR application regarding a virtual exploration of the Knossos Palace. Both applications refer to the same content in a different approach, based on the used Medium (AR vs VR).

- **A Mobile, AR Inside-Out Positional tracking algorithm, (MARIOPOT), suitable for modern, affordable cardboard-style VR HMDs**

Figure 2.12: Various interaction mechanics implemented on Meta AR glasses.

[56]: MARIOPOT was an affordable, low-cost VR visualization h/w and s/w method, for heritage professionals to employ it for VR archaeological sites and Cultural Heritage applications. Taking advantage of the RGB camera sensor that each modern mobile device is equipped, we described a novel combination of inside-out AR tracking algorithms based on both marker and markerless tracking systems to provide the missing positional tracking for mobile HMDs.

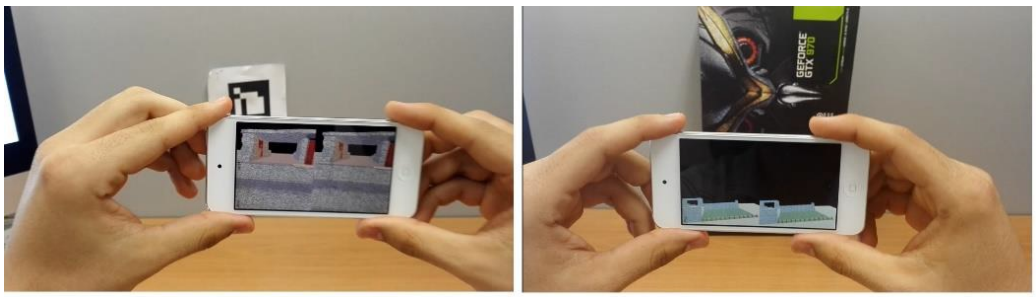


Figure 2.13: Positional tracking in both marker and markerless implementation.

- **Life-sized Group and Crowd simulation in Mobile AR [44]:** In this work, we created AR crowd and group simulations with life-sized augmented characters that simulate crowd behavior with real human locomotive motions and body gestures. We employed the glGA graphics framework along with RVO2 library for collision avoidance capabilities. To evaluate our platform, we developed two applications for crowd visualization based on mobile VR and markerless AR using MetaioSDK.

Figure 2.14: Life-sized AR crowd simulation on mobile device.

Figure 2.15: Our Computer Vision Lab in 2016. Mixing the virtual and the real world

Chapter 3

The training Scenegraph model

To achieve a goal whether it is the restoration of a statue, the repair of an engine's gearbox or a surgical procedure we need to follow a list of tasks/steps in a sequential order. We are referring to those steps as **Actions**. For instance, if we want to hang a painting on the wall we have to perform the following steps (Actions):

i. Mark the wall using a pen.

ii. Hammer a nail at the marked spot.

iii. Hang the painting on the wall.

Those are the three steps that someone needs to complete to hang a painting on the wall. Having those steps in mind, we create nodes, each one representing an Action.

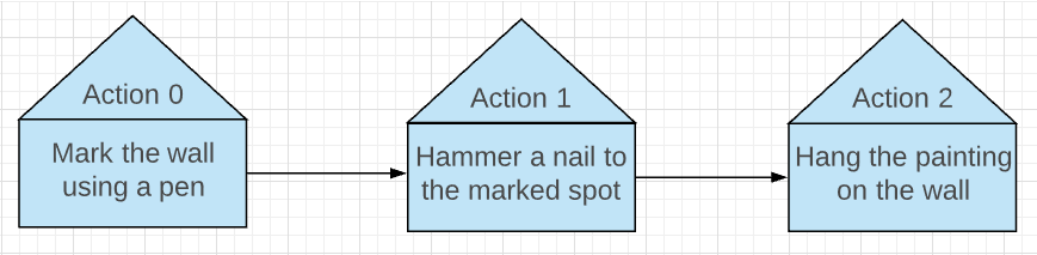


Figure 3.1: A basic sequential procedure from three distinct steps.

However, in complex applications there are dozens of Actions, in this case a sequential representation is not very convenient. For this reason, we implemented the training Scenegraph architecture. A training Scenegraph is a tree with three levels of depth. The root of the tree defines the operation/process, on depth 1 we have the Lesson nodes, depth 2 the Stage nodes and finally depth 3 the Action nodes.

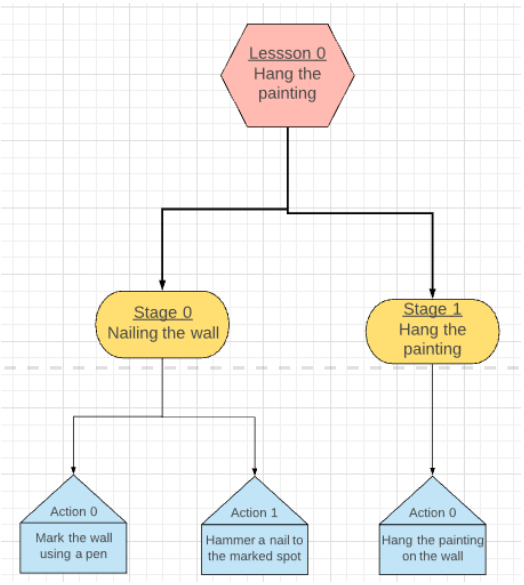


Figure 3.2: Our Lesson-Stage-Action scenegraph tree example

This data structure is referred as the scenegraph or LSA nodes from the initials of the three main structural components. The procedure runs only on Action nodes but we also use the other nodes in a tree format to merge parts of the simulating procedure.

For instance, we can present the above 3 Actions in a tree format as follows:

In this scenario, we decided to group the first two Actions in a Stage since both of them are referring to steps that are linked with the nail. The last action can be placed alone in a stage particularly for this cause.

After those optimizations, we can use this lesson in a more complex procedure along with other lessons to construct a bigger Scenegraph tree. However even though we have multiple types of nodes (Lesson, Stage, Actions), only the Action nodes have customizable behaviour. The operation runs only on Actions, the other nodes are for traversal and scene management. Scenegraph is implemented under the "Scene Graph" gameObject in unity scene. This gameObject will contain the Lesson, Stages and Actions. Scenegraph will manage, perform and run all the Actions as an educational pipeline.

3.1 The training Scenegraph Data structure

At this point, we have outlined a basic training scenario but it still needs implementation to be transformed into a dynamic representation within our application. The main principle of this project is to modify the same data structures (scenegraph, LSA behaviours) using three different editors (scripting, visual scripting and the VR editor). To achieve this, the shared scenegraph data is stored as an xml file for efficient maintenance and editing. However, an xml file is not easy to read especially when crowded with various information and data fields due to complex data structures. This was the reason the three developed tools offer extended functionalities, editing abilities and easy maintenance of the scenegraph structure even for complex training scenarios.

The figure 3.3 illustrates an example of a scenegraph xml. Its tree structure and the defined data fields to store information for the LSA nodes characterize each training scenario. From the image, it is visible that using an xml data structure is

```
<ArrayOfLessons>
  <Lessons>
    <Lesson_Name>Hang the painting</Lesson_Name>
    <Stages>
      <Stage_Name>Nailing the wall</Stage_Name>
      <Actions>
        <Action>Mark the wall using a pen</Action>
        <ParallelModules>
          <ActionModules>
            <ActionClassName>MarkWallAction</ActionClassName>
          </ActionModules>
        </ParallelModules>
        <ParallelModules>
          <ActionModules>
            <ActionClassName>InsertNailAction</ActionClassName>
          </ActionModules>
          <ActionModules>
            <ActionClassName>HammerNailAction</ActionClassName>
          </ActionModules>
        </ParallelModules>
      </Actions>
    </Stages>
  </Lessons>
</ArrayOfLessons>
```

Figure 3.3: Scenegraph xml example.

not easy to interpret complex training scenarios with multiple levels of information and different nodes. However, exploiting the proposed authoring tools there is no need to work directly with the xml file since the visual editors are far more reliable and easy to use. Below there is an explanation of the xml tags used in this script.

- **ArrayOfLessons:** Contains the list of assigned lessons.
- **Lessons:** Contains a Lesson.
- **Lesson_Name:** Name of Lesson.
- **Stages:** Contains a Stage.
- **Stage_Name:** Name of Stage. This string is only used to name the gameobject and nowhere else.
- **Actions:** Contains an Actions.
- **Action:** Name of Action. This string needs to address in a small sentence the purpose of current Action. (e.g. Unscrew the flashing bolt).
- **PrallelModules:** List of parallel modules. This feature enables the initialization of two or more Actions in parallel. The Actions contained in this list will run in parallel and users will be able to choose which one they would like to complete first. An example of parallel actions may be in a car service

training simulation where user would have the choice to examine first the engine OR the brake pads. We will explain parallel Actions and Alternative paths in a following section.

- **ActionModule:** Contains Action behaviours (Action Scripts). An Action is possible to have multiple Action scripts for user to complete before the Action is finished. Assume a complex step with multiple sub-tasks, in this case the developer may want to compress the Action into a single one (one Action Node) but expand its implementation to form multiple sub-actions. These sub-behaviours of the main Action are the Action Modules.

- **ActionClassName:** Contains the Class name of this Action (Action script's class name). Each Action has an Action script, which describes its behaviour. This script will automatically run by Scenegraph when the right time comes. In order to have a clear project we recommend storing all the Action scripts under the format LessonX/StageX/ActionX to match the Scenegraph Nodes.

In the following section, we will dive into the details of the Action node to analyse the different prototyped Actions and discuss IAction module and its functionalities.

3.2 IAction Interface

The Action object reflects a flexible structural module, capable to generate complex behaviours from basic ones. This is also the concept idea behind scenegraph; **provide developers with fundamental elements and tools to implement scenarios from basic principles.** Each Action script describes the behaviour in means of physical actions in the virtual environment. In technical details, each Action script implements the IAction interface, which defines the basic rules every Action should follow. This interface ensures that all Actions will have the same methods. The methods of IAction interface are explained in detail below.

- ParallelActionID: (Property) Stores the ID number of the parallel module the Action is registered. This information is used to specify the alternative path the user decided to complete.

- Initialize: (Method) This method is the first method to call when starting an Action. It is responsible to spawn all the necessary prefabs for the Action to run normally.

- Perform: (Method) What is the behaviour of the action when completed? This method cleans the current Action and ensures that everything not needed is deleted before the next Action starts. Also plays animations, and unity Actions to finalize the Action. Performing an Action means, we complete the Action and we proceed with the next one. When Perform is called, Scenegraph goes to the next Action and Initializes it.

- Undo: (Method) This method includes all the necessary calls to reset the Action. This includes the deletion of spawned prefabs and all the necessary animations and Unity Actions to set the Actions before it. Finally, Scenegraph jumps to the previous Action and Initializes it.
- Clear: (Method) Clears the scene from initialized objects and references of the Action. This method is also used from Perform and Undo methods to clear the currently active Action.

3.3 Action Prototypes

"A journey of a thousand miles begins with a single step."

– Confucius

At this point, we have described the basic interface each Action should implement to initialized and performed properly. With this interface, a developer can generate action scripts that behave in a common ruleset, following the scenegraph pipeline. However, this would not be very convenient since the Actions in this form have a lot of flexibility in terms of implementation. This is problematic in systems where the developer needs classification properties to run across different modules. To make things clear, in our case we need to develop VR training scenarios where users would be able to perform specific tasks. To make our system more efficient we have to limit the capabilities of the Action entity to target simple but commonly used behaviours/tasks in training scenarios. Modelling those behaviours, we will generate a pool of generic behavioural patterns and tasks from which we will develop scenarios that are more specific.

As an example, assume we want to recreate a car service scenario where the mechanic needs to unscrew the oil plug under the car to change the oil. Instead of implementing a complex task that will recreate this scenario into a single Action module, we can dismantle the process into smaller sections and implement those instead. In our case those steps will be 1) Insert the wrench to the oil plug, 2) Unscrew the oil plug, 3) Remove the oil cap. Those steps are more generic and reusable into other training scenarios than a single complex Action. We call those behaviours **Action Prototypes**.

For reusability purposes, we developed three different Action Prototypes to test the system with fundamental behaviours and to examine how we can dismantle a training process into basic tasks. Each Action prototype implements some specific methods according to the functionality we would like to support. For example, the "Insert wrench to oil plug" Action needs the wrench (a tool) and the oil plug (a 3D model) to be accomplished. Those assets need to be visible during the training scenario for user to take the tool and complete the action. In addition, when the Action is completed there is no need to keep the oil plug available so maybe there

is a need for deletion there. Those behaviours describe each prototype and we need to pay attention to the details to complete the fundamental prototypes since they have different implementation.

At this point, it is important to mention, the implementation of Action Prototypes was highly inspired by Game Programming Patterns [38] as **an alternative paradigm for design patterns specially design for VR experiences.** The immersion of virtual environments causes the implementation of programming patterns to fit into a more interactive way of thinking. The classic game design patterns tend to be deprecated since interactive environments like VR applications are focused to the connection of user with the virtual assets. For this reason, the design patterns developed in this project designed to match the needs for **interactivity, embodied cognition** and **physicality** of VR experiences. In the following sections, there is a brief explanation of the action prototypes and examples of usage.

3.3.1 Insert Action

Insert Action is referring to a specific type of Action that user has to insert an object to a specific position in order to complete it.

```
public class InsertWrenchToPlugAction : InsertAction
{
    public override void Initialize()
    {
        SetInsertPrefab("Lesson1/Stage1/Action0/WrenchInteract",
                        "Lesson1/Stage1/Action0/WrenchFinal");

        base.Initialize();
    }
}
```

Listing 3.1: Insert Action script example.

- **SetInsertPrefab(string arg1, string arg2)**
  This method sets the Action's insert prefabs that will be spawned on Initialize. To set an insert Action we need to spawn two different objects, the interactable item and the final prefab. The first argument is the path to the interactable prefab and the second the path to the final. The interactable object refers to the asset in the virtual environment where user needs to take it using the controllers and place it in the desired position indicated by the final prefab. The final prefab is the same object as the interactable but with disabled mesh renderers. In the position where the final prefab is located, a holographic visualization indicates the correct orientation of the prefab. An Insert Action script may have multiple prefabs for insertion; in this case, the developer needs to call SetInsertPrefab method with its arguments for

each insertion separately. To Perform the Action user needs to insert all the interactable prefabs correctly.

3.3.2 Remove Action

Remove Action describes a step of the procedure which user has to remove an object using his hands.

```
public class RemoveOilPlugAction : RemoveAction
{
    public override void Initialize()
    {
        SetRemovePrefab("Lesson2/Stage1/Action1/CupRemove");
        SetRemovePrefab("Lesson2/Stage1/Action1/OilPlugRemove");

        base.Initialize();
    }
}
```

Listing 3.2: Remove Action script example.

- **SetRemovePrefab(string arg1)**
  This method sets the Action's removable prefabs to initialize the Action behaviour. To set a Remove prefab we need a string which contains the path to the removable prefab. This method can be called many times in an Action Script. Each time SetRemovePrefab is called a new removable prefab is added into the remove prefabs List. To perform the Action user needs to remove all of them. In this scenario, user needs to remove the protective cup and the oil plug to complete the Action.

3.3.3 Use Action

Use Action refers to the step where user needs to take an object from the virtual scene and interact with it over a predefined area.

```
public class CleanOilSpillAction : UseAction
{
    public override void Initialize()
    {
        SetUsePrefab(Lesson3/Stage0/Action1/OilCollider",
"Lesson3/Stage0/Action1/Cloth");

        base.Initialize();
    }
}
```

Listing 3.3: Use Action script example.

- **SetUsePrefab(string arg1, string arg2)**
  This method sets the two prefabs the use Action needs to operate properly, the Use Prefab and the prefab collider. The Use Prefab is an object that is spawned into the virtual scene and works in a similar way as the interactable item in Insert Prefab with a small twist. In order to accomplish the Use Action you have to take the use Prefab and place it on top of the spawned use collider (first argument) for an amount of time. This prototype is useful in situations where user needs to take an object and bring it into contact with a second one. In this scenario, user needs to take a cloth and clean the oil spill from the workshop's floor. Another example may be the scrape of an old paint from the wall or the procedure of cleaning a dirty window.

3.3.4 Base Prototype

The Base Prototype does not represent a behaviour like the previous prototypes, is the base class where the other prototypes derive from. It contains common methods used across multiple prototypes for better organization and code optimization. Figure 3.4 illustrates an architectural diagram of Action Prototypes to visualize better their dependencies.

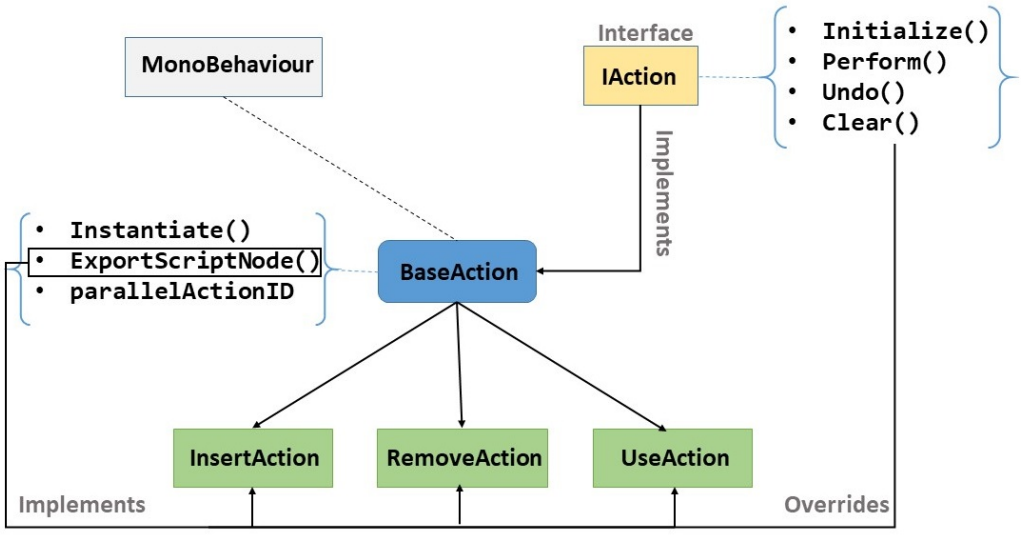


Figure 3.4: Action Prototypes Architecture diagram.

3.4 Alternative Paths

The Action prototypes mentioned above, function as a new design pattern for VR experiences, a modular building block to develop applications in combination with

the scenegraph architecture described. However, with the proposed scenegraph architecture we can develop VR experiences following a "static" pipeline of Actions. Scenegraph is not just a static tree, it's a dynamic graph. Since an educational pipeline can lead to multiple paths according to user's actions and decisions, Scenegraph does it so. Imagine our previous example with the car's oil change scenario. Most of the times the mechanical engineer follows the same sequential actions in the same order. A static Action graph (our scenegraph) can describe this specific procedure, following the same pattern over time. Nonetheless, this does not fit a dynamic experience where users can select what actions would like to complete first and in what order. Considering the oil change car service paradigm, the mechanic may like to check the engine for a potential malfunction instead of changing the oils that is a trivial procedure. In addition, certain actions or even wrong estimations and technical errors may deviate the original training scenario from its normal path. For example, in a paint restoration process if the technician does not pay attention to the correct consistency of used chemicals may cause damage to the painting. This damage should now be fixed causing the scenegraph to dynamically add more Actions to fix the damage.

We implemented these functionalities in such a way to **support real time decision making** and as a result, Scenegraph can change its structure (Nodes) as the training scenario goes on. Scenegraph currently supports the addition, deletion and alternation of Lessons, Stages and Action Nodes depending on the user's actions and decisions. Below there is a brief explanation on the process needed to create and use an alternative Path.

3.4.1 Alternative Path Generation

Alternative Actions are similar to the normal ones with the particularity that they store some additional data on the scenegraph xml. To begin with, Alternative paths are referring to scenegraph nodes (Lessons, Stages or Actions) that are not a part of the normal training pipeline and thus they are not generated with the normal flow of actions. To summon an alternative node, another Action node should call this trigger event to generate the Alternative node. As mentioned before, alternative nodes are similar to the classic ones and as a result, they are saved and generated from the same scenegraph xml.

The image 3.5 shows an example of an Alternative Lesson from the xml file. The xml format is the same with the Scenegraph storyboard except the Lesson_Name tag. There is some extra information on Lesson Name regarding the alternative lesson behaviour. At this Alternative lesson we have the Lesson Name: Oil Change ADD(1000-110). This string consists of four elements: the Lesson Name (Oil Change), a special keyword (ADD), the action which triggers the alternative path (1000) and the position within the scenegraph tree where the new action will spawn (110).

First, we have to define the keywords used for Alternative Paths. We currently use 3 keywords (ADD, DEL, RPL) and each one serves a different role for the

```
<Lessons>
  <Lesson_Name>Oil Change ADD(1000-110)</Lesson_Name>
  <Stages>
    <Stage_Name>Check Car Oil Level</Stage_Name>
    <Actions>
      <Action>Unplug the oil cup</Action>
      <ParallelModules>
        <ActionModules>
          <ActionClassName>InsertWrenchAction</ActionClassName>
        </ActionModules>
      </ParallelModules>
      <ParallelModules>
        <ActionModules>
          <ActionClassName>Check oil level</ActionClassName>
        </ActionModules>
      </ParallelModules>
    </Actions>
  </Stages>
</Lessons>
```

Figure 3.5: Alternative Lesson snippet from scenegraph xml.

Scenegraph manipulation.

ADD: Add a node BELOW another one in the Scenegraph

DEL: Delete a node in the Scenegraph

RPL: Replace node with another one in the Scenegraph

Following the mentioned keyword there is a parenthesis with some district values indicating the two LSA IDs separated with a dash. The first three numbers (1, 1, 0) specifies the action that will trigger this alternative path. The second values are depending on the keyword but for our current ADD example the second LSA IDs are referring to the Lesson that AFTER this one the alternative Lesson will be added. To sum up this alternative Lesson triggers on Lesson 0, Stage 1, Action 0 and adds the new alternative Lesson after Lesson 0 (so Oil Change will become Lesson 1).

The same example with the RPL keyword would be : Oil Change RPL(01000-000). This Alternative path will be triggered again on Lesson 0, Stage 1, Action 0, Parallel Module 0, Action Module 0 and REPLACE the Lesson 0 with the Oil Change Lesson (Oil Change will become Lesson 0).

Finally the DEL keyword would be Oil Change DEL(0—1—0-0—0—0). This Alternative path will be triggered again on Lesson 0, Stage 1, Action 0 and DELETE the Lesson 0.

Except from Lessons we can handle alternative Stages and Actions. Figure 3.6 presents an alternative Stage xml file.

For alternative stages, we need a dummy Lesson just to have the correct xml format. Since this xml handles alternative Stages, the keyword and the LSA IDs are written at Stage_Name tag.

```
<Lessons>
  <Lesson_Name>$</Lesson_Name>
  <Stages>
    <Stage_Name>Restore old painting ADD(2010-010)</Stage_Name>
    <Actions>
      <Action>Clear Paint from dust</Action>
      <ParallelModules>
        <ActionModules>
          <ActionClassName>UseClothAction</ActionClassName>
        </ActionModules>
      </ParallelModules>
    </Actions>
    <Actions>
      <Action>Re-paint highlighted Area</Action>
      <ParallelModules>
        <ActionModules>
          <ActionClassName>PaintSpotAction</ActionClassName>
        </ActionModules>
      </ParallelModules>
    </Actions>
  </Stages>
</Lessons>
```

Figure 3.6: Alternative Stage snippet from scenegraph xml.

Finally an example of Alternative Action:

```
<Lessons>
  <Lesson_Name>$</Lesson_Name>
  <Stages>
    <Stage_Name>$</Stage_Name>
    <Actions>
      <Action>Insert new gear into box RPL(1014-120)</Action>
      <ParallelModules>
        <ActionModules>
          <ActionClassName>InsertGearAction</ActionClassName>
        </ActionModules>
      </ParallelModules>
    </Actions>
  </Stages>
</Lessons>
```

Figure 3.7: Alternative Action snippet from scenegraph xml.

As previous, for Alternative Actions we define the keywords and the LSA IDs at Action tag and we need to include a Lesson and a Stage node for structure purposes.

The main purpose of Alternative paths is to make the training procedure more challenging for advanced trainees. This dynamic functionality of scenegraph is

used along with the parallel Actions mechanic where more than one Actions is initialized at the same time. In this way, if user decides to go with one approach instead of another one, the scenegraph automatically deviates from its normal state and transforms itself by triggering the alternative paths linked with the specific Action.

3.5 Fast and efficient scenegraph traversal

The selected data structure (scenegraph) illustrates a dynamic tree with active leaf nodes and the rest of nodes (Lessons, Stages) offering management and traversal capabilities. Training procedures are most of the times linear, meaning that one tasks follows another to complete the scenario. However, a linear data structure represented by a list would offer limited functionalities in terms of traversing the training scenario, moving back and forth the educational pipeline and selecting specific moments during the timeline the user would like to jump in immediately. For this reasons, we visualize and represent the training steps using a tree. The scenegraph traversing was optimized, utilizing different levels of abstraction (Lessons, Stages and Actions) and recursive algorithms to expend the capabilities of the traversing mechanism. Intermediate nodes (Lesson, Stages) offer great flexibility for alternative paths since they reduce their complexity especially when the developed scenario needs specific parts to run in different order or if a critical error occurred and scenegraph needs to add or replace specific nodes. Finally, the Jump Lesson functionality offers immediate transition to the selected Lesson, giving user the choice to skip unnecessary actions and focus only on specific parts of the training scenario.

3.6 Valley of Interactivity

After experimenting with various design patterns and interaction techniques for VR, we discovered an interesting pattern regarding the correlation of user experience and the interactivity of the VR application. Without a doubt, an immersive experience relies significantly on the implemented interactive capabilities that form the general user experience. As a result, to make an application more attractive in means of UX we need a more advanced interactive system. However, as we implement more complex interaction mechanics there is a point in timeline where the UX drops dramatically. At this point, the application is too advanced and complex for the user to understand and perform the various tasks with ease. We characterize this feature as heterogeneous behaviourism meaning that user's actions do not follow a deterministic pattern (same actions cause different behaviours) resulting in the inability to complete the implemented Actions due to their incomprehensible complexity.

In contrast, applications with limited interactivity follow a linear increase of their user experience. From applications where users are only observers, like

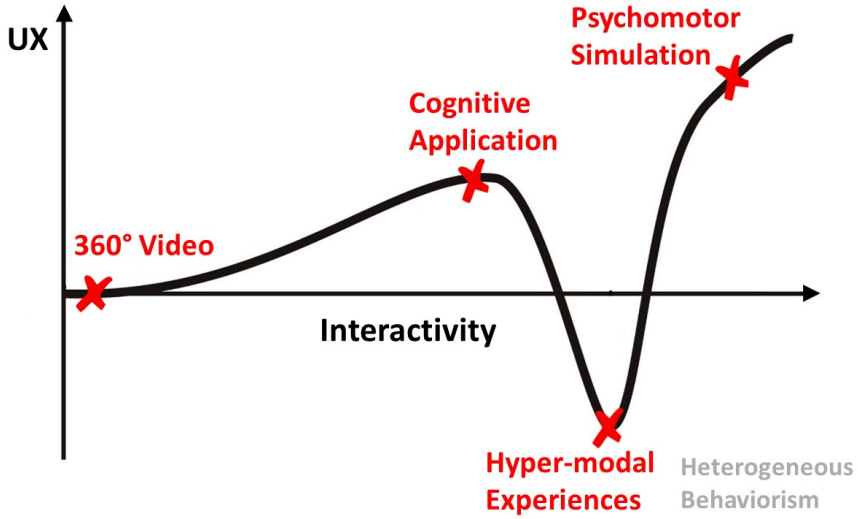


Figure 3.8: "Uncanny valley" of interaction. Correlation between UX and interaction in VR.

360°VR videos to cognitive applications, the interaction curve is linear and VR experiences easy to understand and accomplish. To overcome the effect mentioned before, applications need to drastically enhance their interactivity capabilities and offer users a more intuitive VR environment to understand how they are supposed to act in the virtual world. A commonly used method for applications to overpass the valley of interactivity is to augment the content with in-game tutorials or similar visual information in order to highlight the current objectives. Overpassing the valley of interactivity, applications are evolving rapidly to follow a psychomotor methodology integrating embodied cognition for the maximum user experience.

This "valley pattern" is encountered in robotics (and computer graphics) as the "Uncanny Valley" referring to the relationship between the degree of an object's resemblance to a human being and the emotional response to such an object. Similar to the uncanny valley of virtual characters, the graph 3.8 depicts the correlation of UX and interactivity classifying different application models according to their interactivity capabilities.

Chapter 4

Visual Scripting

Up to this point, we presented the basic principles of scenegraph architecture and the prototyped patterns for VR training simulations. This model is capable to generate applications from reusable fundamental elements (Actions) supporting basic insert, remove and use behaviours in VR. Utilizing this system, developers can implement new experiences using the classing scripting abilities of C# and Unity engine. However, what is the next step? What can be done to enhance the development process and speed up the content creation? The complexity of scenegraph xml may cause difficulties visualizing the LSA nodes especially for major training scenarios. Another point is the programming skills required do develop such experiences. Using the proposed architecture could be challenging for inexperienced programmers and a lot of errors may occur resulting in limited user experience and time consuming debugging sessions.

To eliminate the mentioned difficulties, **we introduce Visual Scripting as a visual authoring tool to manage, maintain and develop VR experiences** utilizing the training scenegraph and the Action modules. Visual Scripting encapsulates all the functionalities from the base model and at the same time offers high visualization capabilities, which are very effective especially on extended projects.

4.1 The Visual Scripting metaphor

The development of a visual scripting system as an assistive tool aimed to visualize the VR training scenario in a convenient way, if possible fit everything into one window. The simplicity of the tool was carefully measured when designing the features since the offered functionalities could be used from non-programmers. A coding-free platform offers a safe environment to work, reducing programming errors and unforeseen discrepancies between projects. From the beginning of the project, one of the main design principles was to strategically abstract the software building blocks into basic elements. The main idea behind this abstraction was the improvement of the visual scripting and VR Editor platforms. Fundamental elements construct a better visual environment than complicated ones.

Moving into the Visual Scripting metaphor, the scenegraph data structure already forms a tree. This design property led us to visualize the system as a node based editor with nodes linked together with edges forming logical segments and reusable parts. An example of a complete diagram representing a training scenario is illustrated in figure 4.1.

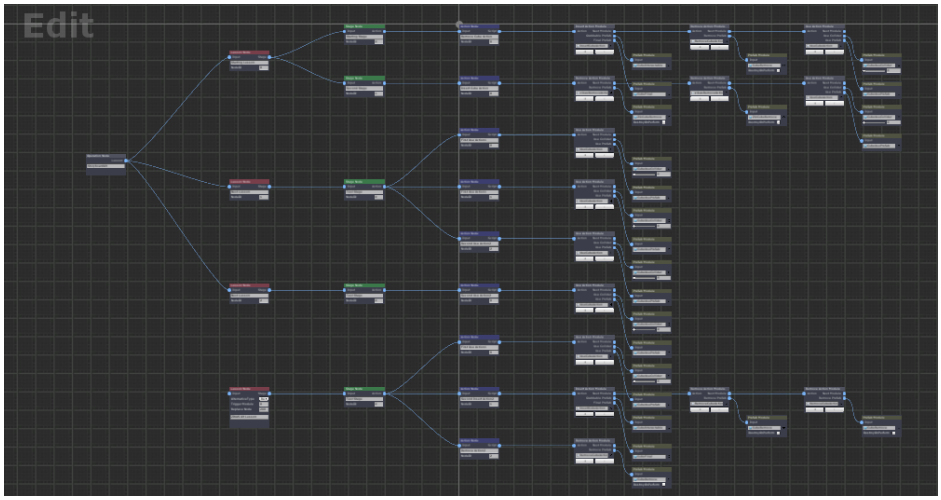


Figure 4.1: A training scenario visualized from the visual scripting editor.

The figure shows different types of nodes with different colours, special input fields and extra buttons. To begin with, all the nodes (except form the sub-tree in the bottom of the image) are linked into one node located at the left side of the image. This node represents the root node of the training scenario. Each simulation has one root node, which is used for better visualization and traversal purposes. This sub-tree represents an alternative Lesson and since alternative LSA nodes are not within the main pipeline of the scenario, they are not linked to the root node. Following the links to the root node, we identify the basic LSA nodes linked together: Lesson (red), Stage (Green) and Action Nodes (Blue) form the scenegraph structure indicating the logical parts of the training scenario.

Up to this point, we have constructed the main structural components of scenegraph. Early versions of the visual scripting system represented only this information and it was used to construct the scenegraph tree and nothing else. Users were able to construct the skeleton of the project from the visual scripting panel and then implement the behaviour of each Action programmatically through C# code. The advantages of this solution mainly focused on the easy maintenance of scenegraph structure that was hard to manage directly from the xml file. Being able to generate the scenegraph tree from a visual interface speed up the content creation and lead to the better visualization of the training scenario. However, the scenegraph architecture may expand significantly into larger tree structures, thus managing content from the xml file would be challenging. Visual Scripting

solves this problem, extending scenegraph capabilities into a simple coding free authoring tool.

4.2 Implementing Action Behaviour with Visual Scripting

One of the major goals of this project was to **implement a visual scripting system to replace coding of simple Action behaviours with intuitive and easy to use graphical patterns** like nodes, drop down menus, input fields and other clickable elements. In this way, the content creation evolves into a coding free process, encapsulating the system principles into equivalent visual metaphors giving programmers the ability to generate VR training content without high demand in software background. In the following sections, we will dismantle the Visual Scripting metaphor, explaining the features and implementation techniques.

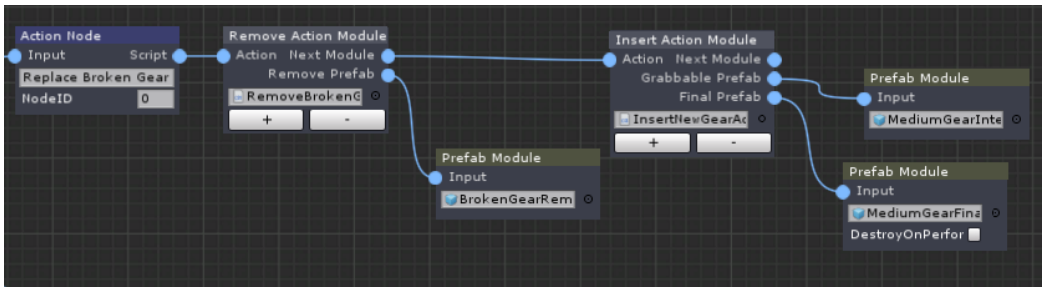


Figure 4.2: Action Node and script modules in Visual Scripting Editor.

The figure 4.2 illustrates an Action Node accompanied with its script modules and some prefab modules. As mentioned before, an Action represents a basic task in the training scenario. In this case, our training scenario is the restoration of an antique clock and the selected Action represents the replacement of a broken gear. The Action Node comes in pair with a node Description; it is a text field under the node name to inform user what is this Action about and what it needs to be done to accomplish this goal. In our scenario, users need to remove a broken gear and then insert a new one to make the clock working again. This Action contains two different logical parts: 1) Remove the broken gear and 2) Insert the new one into place. For this reason, we will split the action behaviour into two separate Action scripts, where user needs to accomplish first the removal of the broken gear and then the insertion of the new gear. Since our building blocks are following an abstraction over fundamental behaviours, is possible to implement this Action with multiple ways. For example instead of having two different Action scripts within the same Action module, a developer could split the single Action into two different Action Nodes thus each one will have a single Action Script. This modularity of the proposed system is one of the major points of scenegraph architecture. The

system abstraction makes the implementation methods generic enough to enable different implementations using the same basic tools.

In more detail, both Actions (Remove and Insert) are linked together forming a combined Action. The first sub-Action contains the Remove Action and the second one the Insert Action, user needs to complete both to proceed with the next one. We will take those two Actions as an example to examine the visual scripting metaphor in detail. To begin with the Remove sub Action, the Action module contains an input knob and two output knobs. The Remove Action module node contains an input knob, which is linked to the Action node, and an output knob named Next Module, which is linked to the next sub-Action, if any. Those knobs are shared to all other Action Modules as they form basic structural capabilities. The next element we identify is an output knob named Remove Prefab. This knob is linked to a Prefab Module with a reference to the prefab that user will remove to complete the Action. Every Action module is linked to such prefabs in order to keep track with the elements relative with each Action. Below the prefab knob, there is an input field with a file reference. This file is the unique script of this Action; it describes the behaviour of the Action following the IAction interface. It is automatic generated through the input from the scenegraph editor. We will analyse more details about those scripts in the following paragraphs. Finally, two UI buttons appear at the bottom side of the node indicating the feature of multiple object removals during the same Action. By pressing the "+" button another output knob named Remove Prefab will appear to link another prefab with this Remove Action. For example, if we had to remove multiple gears for this Action we would assign the same number of Remove Prefabs, and then later the generated script would instantiate the same amount of prefabs for user to interact. Figure 4.3 reflects a detailed diagram for the node components.

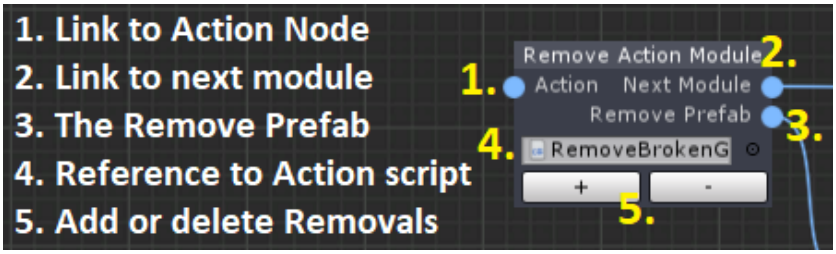


Figure 4.3: Remove Action Module Elements.

We will continue with the Insert Action represented by the insertion of the new gear into place. To implement this behaviour we need to construct a new Insert Action module and assign it as the sub-Action of the previous Remove Action module. The Insert Action has similar setup with the Remove Action except that we need different prefabs to link to the action module. From IAction we learned that an Inset Action needs an interactable prefab (for user to take) and a final position prefab (indicates the goal position of the interactable prefab). Those two

prefabs are linked into the Grabbable Prefab and Final Prefab output knobs. In this case, there is an extra option in the prefab module of the final prefab, a checkbox named DestroyOnPerform. This checkbox along with other options in the prefab modules, sets various parameters for the Action, in this case if we check the option, the final prefab will be destroyed as long as we completed the Action. Figure 4.4 illustrates the Insert Action module we created for this example.

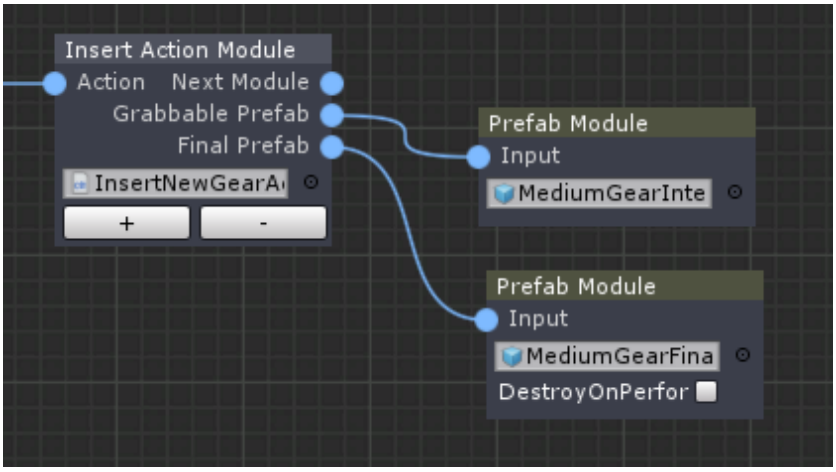


Figure 4.4: Example of Insert Action Module and linked Prefab Modules.

At this point, we created a complex Action (Removal and Insertion of object) using two basic Action modules. However, we need to generate a script from each Action module to export the developed behaviour from the scenegraph editor. This procedure is explained briefly in the next section.

4.3 Dynamic Action Script code Generation and Compilation

The proposed scenegraph architecture lies on a pipeline of Action scripts to replicate a training scenario in virtual reality. The simplest Action script contains only the overridden Initialize method, which defines the prefabs that would be instantiated when the scenegraph triggers its Initialize method (described in section 3.3). Visual scripting can generate realtime simple Action Scripts from the information provided from the visual input (Action and Prefab modules). After completing the visual construction of an Action using the scenegraph editor the next step is to generate the Action script to save the implemented behaviour in a C# code script.

In the previous example, we explained how to reconstruct the removal of a broken gear from the antique clock with visual scripting. We will continue with this example to follow the steps of the runtime script generation. Each Action module

features a method for generating the Action script according to the linked prefab nodes and the data provided. If the module was developed correctly following the main principles of scenegraph editor, (no missing fields etc.) generating the script is as simple as right clicking on the node and selecting the "Update Action Script" option. This method triggers the script generation process from the save manager of visual scripting and starts writing the code that implements the script. Visual Scripting identifies the Action type of each module and calls the abstract method implemented in each Action script to generate the Initialize method.

To write C# code runtime, we utilized CodeDOM, a build-in tool for .NET Framework that enables runtime code generation and compilation. The CodeDOM defines an object called a CodeCompileUnit, which can reference a CodeDOM object graph that models the source code to compile. A CodeCompileUnit has properties for storing references to attributes, namespaces, and assemblies. To generate the Action code, we have to instantiate the parameters and attributes with the desired values, and fed the CodeCompileUnit. The abstraction of Action Prototypes offers an elegant implementation to generate each script using a single prototyped method. However, to finalize the Action script, except from the Action Type (Insert, Remove or Use) we also need the implemented behaviour. Action Prototypes retrieve this information directly from the visual scripting editor through different nodes and links relative to the Action module. As an example, figure 4.5 illustrates an Insert Action linked with two prefab modules. To generate the Action script, the insert Action prototype will seek the references of the interactable and final prefab to find their path within the Unity project. The next step is to pass this information to CodeDOM and the compilation unit we created previously to set the Action's SetInsertPrefab method with the two strings containing the prefab paths as arguments.

```
using ModularScenegraph.Prototypes;

public class InsertNewGearAction : InsertAction
{

    public override void Initialize()
    {
        SetInsertPrefab("Lesson1/Stage1/Action0/MediumGearInteractable", "Lesson1/Stage1/Action0/MediumGearFinal");
        base.Initialize();
    }
}
```

Figure 4.5: The auto-generated Insert Action from CodeDOM.

At this point, CodeDOM generated the Action script and now we have to compile the script for Unity to recognize it. Thankfully, CodeDOM API provides all the necessary methods for runtime compilation and makes the process straightforward. However, Unity does not recommend runtime compilation and the engine itself warns you for possible errors during use since it conflicts with the engines native compilation pipeline. Another issue is the build in compiler of Unity, which features by default the Mono C# compiler for .NET 3.5 and the Roslyn for .NET

4.6. In this project, we used the native mono compiler but also experimented with Roslyn and its capabilities, although we aim to integrate Roslyn in an upcoming version of scenegraph architecture to benefit from the advanced runtime compilation features. Project Roslyn is a framework created by Microsoft to give a developer deep access to the compilation process. Utilizing Roslyn, a developer can parse a piece of source code, and see all the nodes and tokens in a tree structure. This would be very helpful for an upcoming version of scenegraph where the runtime editing of specific parts of the Action scripts would be possible instead of rewriting the whole script from the beginning.

After compiling the Action script, we can link it with a reference to the input field of the node from the visual scripting editor. Finally, we have to export the xml file from scenegraph editor that will save the node structure and important data for the Action scripts into a single xml file. This step concludes the process of generating an Action script using only the tools provided from the scenegraph editor and more importantly without writing a single line of code.

In this section, we used as an example a two-step process of replacing a broken gear from an antique clock. We dismantle the logical process into two different Actions, firstly the removal of the broken gear and secondly the insertion of the new gear into place. Figure 4.6 illustrates the three different stages of the training scenario.

Figure 4.6: (Left) Removal of broken gear, (middle) Insertion of new gear with the green holographic gear indicating the correct placement and (right) the restored clock mechanism.

4.4 Expanding auto generated scripts

Visual scripting generates a basic Action script that contains the Initialize method, the minimum requirement for an Action to run properly. This implementation provides a simple and elegant solution to deliver an interactive behaviour from a tree (scenegraph) into the VR metaphor. However, there are times where developers need to implement significantly complex Action behaviours to enhance the use experience with additional information and features for a better experience. For example, the last Action in the clock restoration describes the process of starting the clock to identify if everything went to plan and the user managed to revert the damage. In the real world, this process contains the jump-start of the clock by moving the pendulum from one side to the other to start the gear mechanism. A basic VR metaphor utilizing the scenegraph architecture can be achieved via an Insert Action where the intractable prefab will be an interactive pendulum in the start position and the final prefab will be the pendulum in the goal position. To accomplish the Action, user needs to insert the pendulum to the final position in order to start the gear mechanism.

From the visual scripting tool, the generated script will contain only the necessary information to complete the Action, which are the two mentioned objects. However, this implementation would be very bland and the user experience will drop from the simplicity. To make things more appealing, we will instantiate an animated pendulum to replicate the natural clock movement as well as a ticking sound effect as an audio feedback from the working clock. It is crucial to maintain higher realism in VR applications to enhance the immersion and the embodiment effect.

Prototyped Actions were developed using a particular software architecture capable to provide the basic gameplay facilities but also enhance those action according to developer's preferences. In our scenario, a straightforward solution to implement the animated pendulum would be to activate the animation along with the audio feedback as soon as the Action is completed. This methodology is reflected via the Perform method that triggers exactly when the intractable prefab collides with the final prefab, in other worlds when user inserts the pendulum on the correct position. The Perform method can be overridden directly from the Action script to include this additional behaviour. The same principle is applicable with all the other virtual methods defined in the IAction interface like Undo, Initialize etc. We plan to integrate the ability to modify all the virtual methods form the IAction interface directly form the visual scripting editor in future releases. This functionality will simplify the content creation and especially the customization of critical actions that deviate from the basic behaviour. The code snippet bellow overrides the Perform method to toggle the colliders of Antique clock.

```
public override void Perform()
{
    Destroy(GameObject.Find("BrokenGear"));
```

```
        Utilities.CustomObjectToggle.
        ToggleColliders("AntiqueClockFlip(Clone)/BackDoor/Colliders",
            false);

        base.Perform();
    }
```

Listing 4.1: Overridden Perform Method.

To conclude, visual scripting editor generates basic Action scripts according to the visual changes on the editor. For additional modifications, the best practice is to edit directly the exported script and override the declared IAction methods. In this way, we maintain simple scripts clean while complex ones can be adjusted upon request to fit the training scenario.

4.5 One storyboard file to rule them all

Earlier versions of scenegraph stored alternative nodes in a separate xml file resulting in four xml files in total, one for the regular path, and one for each one of the three alternative node types (Alternative Lessons, Stages and Actions). This implementation resulted into a complicated initialization procedure at the start of each training scenario since all four xml files needed to go through the storyboard importer. In addition, multiple storyboard files caused difficulty in updating specific parts of the simulation, since the data structure was spitted into four files.

Visual scripting introduced an editor window with many functionalities and visualization capabilities and along them, the need to upgrade the importing system of scenegraph. Visualizing alternative nodes within the same editor unlocked the development of a single xml file that contains every single LSA node. Alternative paths rendered in the visual scripting editor as normal LSA nodes however, in a separate cluster for better consistency. Both the storyboard importer and the exporter were updated to read and export different types of nodes from the same process (reading and writing to xml files).

To conclude, unifying the procedure of generating the different storyboard files simplified the maintenance of the system, reducing the data files into a single file. We need to mention that the single xml file grew in size since it contains every node of the training scenario. However, this will not cause any significant issues as with visual scripting the procedure of generating nodes directly from the text editor became deprecated. In this version of scenegraph, xml files are used only for saving purposes to store the storyboard after modifying it from the scenegraph editor.

4.6 Reflection for runtime data retrieval

Reflection is the ability of managed code to read its own metadata that describes assemblies, modules and types. Essentially, it allows code to inspect other code

within the same system. A program reflects on itself when it extracts metadata from its assemblies, then uses it to modify its own behaviour or inform the user. Reflection is a powerful tool to exploit in systems where we need this additional data from assemblies.

In this project, **we utilize reflection to gather data from Action scripts and visualize their content using the scripting editor**. There are two main processes our editor implements: 1) Transforming the xml file into nodes and 2) Generate the xml file from nodes. The second one uses information from the editor to export the scenegraph in an xml format. All the needed data is within the editor screen in a visualized graph. However, the process of converting the xml file into nodes requires a more advanced approach due to the limited data stored in the scenegraph xml file. As mentioned before, the xml file stores the name of each node and the class names of the corresponding Action scripts along with some additional information on the parallel and combined Actions. There is no information about the type of the Action script (Insert, Remove and Use) or the used prefabs. The Action visualization requires data like the arguments of specific functions called from the Initialize method (e.g. SetInsertPrefab or SetRemovePrefab) which are written in the script (text format). The question is how to retrieve data from a script in an elegant way?

One of our first thoughts was to develop a parser that will take as input the Action script and extract the important data needed to visualize the Action modules. This idea was quickly abandoned as the implementation would be very complex and not sophisticated enough to handle context variations between different Action types. The solution needs to support an abstract system for future improvements and expansion of current Action library, thus a parser would not be the ideal methodology to support this system. Focusing on the features of C# a .NET languages we started to design a system which handles the assemble files to retrieve as information as needed to construct the visual nodes in scripting editor.

Reflector is the name we gave to this module; it uses Reflection into managed code to read the assemblies in our favor. The process begins with the compilation of the script to generate the intermediate language from C# code. After compiling the script, we have now access directly to runtime types and methods. To implement an algorithm, which works in all scripts, regardless the Action type, we declared once more an abstract method at Base Prototype and implemented it in all types of Actions (Insert, Remove and Use). The abstract method ExportScriptNode is responsible to construct the Action module along with prefab nodes and anything else that is declared within the Script. In this way, each Action type implements its own custom way to instantiate the Action module along with its components since each one has its own unique characteristics. For example, Insert Action needs to instantiate two nodes in total, one for the interactable prefab and one for the final position. On the other hand, Remove Action needs to instantiate only one object. Those variations lead to the design pattern of abstraction for a versatile implementation that follows each Action's characteristics. This abstract method is invoked using InvokeMember function from Type member

along with a few arguments regarding node editor parameters (Listing 4.2).

```
Type type = assembly.GetType();
object instance = Activator.CreateInstance(type);
className = type;
actionType = type.BaseType;

object[] parametersArray = new object[] {parentNode,
    actionScript, parallelID };
(Node)actionType.InvokeMember("ExportScriptNode",
    BindingFlags.InvokeMethod | BindingFlags.Instance |
    BindingFlags.Public, null, instance, parametersArray);
```

Listing 4.2: Using reflection to invoke a method directly from assemblies.

It is worthwhile to mention an optimization we implemented to speed up the compilation times. For training scenarios with many Actions and nodes, the compilation time of all the scripts was noticeable. For this reason, we designed a cashing algorithm of compiled assemblies to keep track of the previous ones and avoid compiling the same script more than once.

Chapter 5

VR Editor

5.1 The Initial idea – Extending system capabilities

The visual scripting system enhanced the usability and effectiveness of the scenegraph system to generate gamified training scenarios through a coding-free platform. The impact on content creation was very strong due to the additional tools and features that introduced in the editor. However, visual scripting lacks on one specific and rather important part: the ability to design on-the-go behaviours and scenarios directly within the virtual environment. This feature will improve the design capabilities while offering an intuitive way to modify and update existing applications.

The implementation of VR Editor was designed as an authoring tool on top of the scenegraph architecture, utilizing the developed features of the system and extending the visualization and interaction capabilities. This interactive tool will reduce even more the time needed to produce training scenarios due to the runtime modification features and the advanced visualization it provides. In the process of developing a training scenario, it is common to modify key assets multiple times to reach a convenient position to instantiate in the virtual environment. In addition, certain programming behaviours are better designed directly from VR instead of the editor due to the different perspective of the used medium; in this case the VR headset and the controllers. In the following section, we will discuss the importance of the medium in which the application will deploy and its correlation with the applied interaction and design methodologies.

5.2 Medium-oriented design principles

Designing the VR editor was a critical task since it should be well balanced between **interaction**, **intuition** and **functionality** to actually improve the development process instead of make it more complex. Below lies a brief explanation of the three basic principles of application design based on the applied medium.

- **Interaction:** Describes how the user can interact with a system. A system

is interactive if there is always an interesting way to achieve a goal using modern technological solutions related to the medium. From gesture based to voice activated commands each medium has its own characteristics and interaction principles that should be carefully taken in consideration at the design phase to avoid backtracking due to complex interaction behaviours and methodologies. What makes a system interactive? For the purposes of this project, we redefine interactivity into a broader definition more closely related to the deployed medium (VR/AR headset, body tracker, desktop etc.). For example, a desktop-based application lacks of interactivity since the only available input comes from the keyboard and mouse. However, multimodal applications that integrate different features and behaviours provide an advanced interactive mechanism to enhance the user experience.

- **Intuition:** Reflects the impact of interaction. A system is characterized intuitive you can get from point A to point B without looking the documentation. In other words describes how the user will achieve his end goal using the provided interaction techniques. Usability is very closely related to interaction as the latter is capable to provide the necessary tools that will characterize and application easy to use or not. A system is user friendly when all of its components are well defined, the outputs follow a deterministic pattern according to the inputs and finally the interface is design according to the medium of interaction (controllers, hand gestures, voice commands etc.). However, there are systems that are complex enough to achieve a wide variety of tasks (3D editors like Maya, 3DS Max) but at the same time their components follow a design pattern simple enough to get acquainted with, on frequent use. The design process should include revisions and rehearsals to make the final product more intuitive and reach a satisfactory level of simplicity.

- **Functionality:** Describes what can be achieved with the proposed system. Functionality is the related to the seriousness of the application in means of context and impact. For example, a game may be interactive and interesting to play however it does not have great impact on the user, an exception of this rule are serious games. Functionality describes the reason of existence of a system. Why we should use this software? What will be the feedback and the results related to our work? Functionality has a serious impact on complexity due to the additional information and data management.

"Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains."

– Steve Jobs

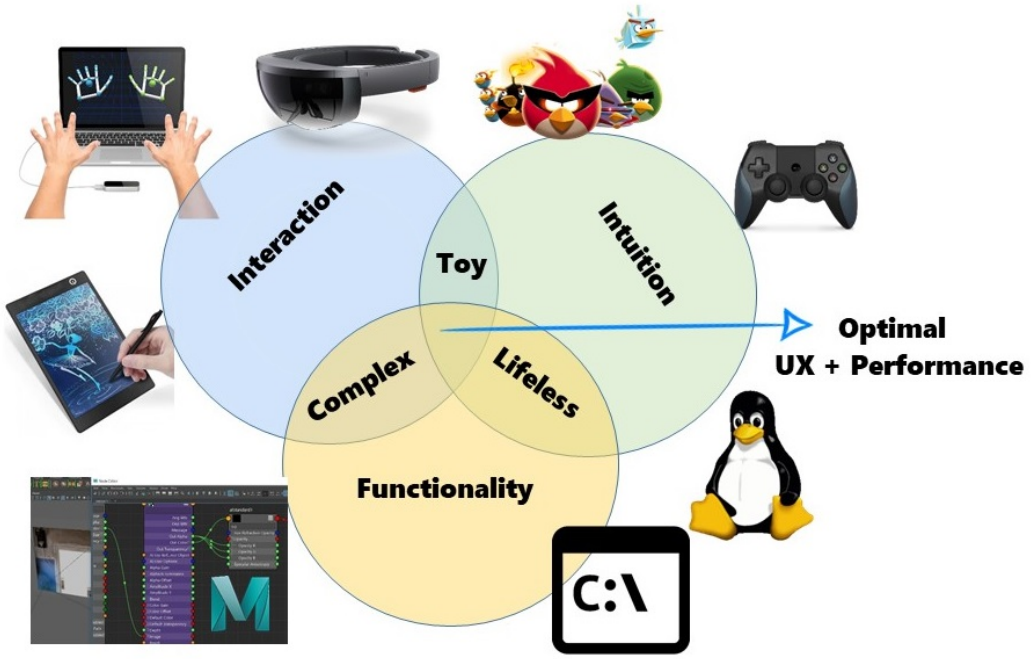


Figure 5.1: Medium-oriented design principles.

Figure 5.1 illustrates the three basic design principles related to the medium along with example applications and their correlation. From this, we can extract precious information regarding the impact of each principle especially on scenarios where only two of them are implemented but not the third one.

Interaction and Intuition: Applications where there is plenty of interaction capabilities along with intuition are characterized as toys since the functionality component is missing. Those applications are easy to use offering an intuitive user experience and a variety of enhanced interactions but their purpose is only related to entertainment neglecting the educational aspect.

Intuition and Functionality: In this section of the diagram, applications are intuitive and they have also advanced functionality, they characterized as lifeless due to the lack of interaction. Applications in this category provide useful output and they are intuitive enough but users most of the times are not happy to use them. A classic example of a lifeless applications may be the command prompt of a system where provides high functionality but its interaction capabilities are limited to keyboard input only. In this category, we can also include certain operating systems like UNIX/Linux where the functionality capabilities are considerably higher than other systems (Windows) and they are intuitive enough due to the advanced UI but their default interaction features are limited.

At this point, we need to make clear that limited interaction does not necessary have a negative impact since it depends on the raison d'être of each application.

This conclusion applies to all three design principles, underling that is not mandatory to include all of them in each application due to the diversity in context and purpose of the end product.

Functionality and Interaction: In this category, applications are highly interactive and functional at the same time but they lack of intuition, thus they characterized as complex. They support a wide range of input capabilities but the lack of intuition does not allow users to use them without experiencing difficulties in understanding how they supposed to perform specific tasks. An example of complex applications with the mention characteristics are the 3D editing programs (Maya, 3DS Max etc.) where they offer a huge amount of parametrization but it is fairly difficult to find on your own a specific feature without looking the documentation or a tutorial. At the same time, the use of a stylus to design 3D models can enhance the interactivity offering a diversity of inputs to the system. In the same category we can include prototype headsets and other gadgets like Microsoft HoloLens 1 and Leap Motion Sensor where is difficult to understand what is you are supposed to do without any guidance from an expert. For example, Microsoft HoloLens applications are not intuitive and this realization is not based on software design principles but the hardware itself due to the lack of robustness.

To conclude, the sweet spot of the diagram combines all three principles, balancing between user experience and performance. Although difficult to implement the perfect scenario, paying attention to the hardware where the application will be deployed is the key element to design the best-case scenario. In the following section we will discuss the how the design principles are affected in different realities (VR/AR/MR) and how we can approach the optimal result.

5.3 The VR metaphor

During the implementation phase, the design of VR Editor was refactored multiple times to achieve an optimal scenario. Previous prototypes of the VR platform were not convenient enough to support the implemented scenegraph features, thus backtracking was inevitable. The main question in this phase was how to design the metaphor from the 2D visual scripting system that runs on a desktop screen to the VR Editor where needs to be rendered in the virtual environment.

5.3.1 Design 1: Interactive Tablet

The initial idea was to construct a similar system with visual nodes in a tablet format where user would have to press buttons on the touch screen using his virtual hands. The implementation continued until a point of realization that this format would simply not fit into a virtual environment. Using the tablet was not intuitive at all, resulting in poor user experience since pressing buttons using the tip of your virtual finger was not convenient. However, many mechanics built for this version of VR Editor passed through the following ones due to their abstraction and generic implementation. Figure 5.2 illustrates of the first prototype of VR

Editor, featuring on the left side the list of LSA nodes and on the right buttons for the insertion of new nodes. The save icon at the bottom panel would generate the scenegraph xml and the bin icon would discard the selected node.

Figure 5.2: First prototype of VR Editor (Non-intuitive, difficult to use, lifeless)

Below there is a list with the positive and negative aspects of this design.

✓ Common design with visual scripting tool.

✓ Scales up to complex scenegraph trees.

✗ Lifeless, not interactive to extend VR capabilities.

✗ Not intuitive, uses same functionalities as a 2D desktop application.

The realization of what went wrong with this design proved to be critical for the continuation of the project. The initial tablet design was not suitable for a virtual reality application since it was clearly derived from a desktop UI design. Working with nodes, drag and drop icons and simple input fields was ideal for a 2D desktop environment but this was not the case for a VR application that uses controllers as inputs instead of mouse and keyboard. **The design of an application should rely on the deployed medium.** An idea cannot be converted to another platform without serious modifications. This has been already mentioned in the previous subsection where we discussed the importance of the medium. In this scenario, VR Editor was designed as a 2D screen on a 3D environment and proven completely incompatible due to the differences in realities. Immersive VR applications should rely on interactive features, not in 2D User Interfaces; the latter was designed for 2D screens and does not support any advanced mechanic to manipulate menus and buttons from within the virtual environment.

In a previous project [55], we presented the differences in the implementation of a serious game for cultural heritage in VR and AR. In this situation, we face a common dilemma where design choices should overpass the medium discrepancies to support a generic design pattern. Of course, we are not obligated to implement the same features in for visual scripting and VR Editor but since they share common functionalities, they should rely on common principles. As an example, it would not be efficient to design a node-based VR Editor on a virtual tablet due to the lack of interaction; however, there are other designs to consider overpassing this issue.

5.3.2 Design 2: Library and interaction with books

From the first prototype, we learned that we should utilize the capabilities of the deployed medium to extend the functionalities of our application. Considering what will be the design of the VR Editor we thought an appealing metaphor, which mainly focuses on interaction with virtual elements. The idea came up after brainstorming on a possible metaphor of Actions to the real world. In a training scenario, the trainee needs a source of information to study the correct steps he need to follow to complete a task. Transforming Actions into physical books from within the virtual environment was the main concept behind the second design of VR Editor. Scenegraph was represented as a physical library with bookshelves being the Lessons, books the Actions and finally Stages were bookstops, merging them into groups. To enable physics based interaction we utilized NewtonVR to give user the ability to handle books using the controllers. Visual scripting connects each LSA node with lines representing their relation; however, in a virtual environment we have the available tools to construct a physical relation between objects. The VR metaphor for generating an Action in this case would be to take a book and place it in the desired bookshelf paying attention to its order and position regarding the book-stops.

Figure 5.3: The library metaphor: (Left) bookshelves and sorted Action books, (Middle) interaction with action books (Right) pull lever to save and export scenegraph xml.

The next phase was to customize each book to replicate the Action generation process similar to visual scripting. This implementation was not finished due to problems we faced in with this VR metaphor. The main idea was to construct a stand where user can place the book on top and browse the pages to customize further its content. This interactive customization for each book made the script generation a very complex procedure. The whole process of generating the theoretical entity of scenegraph through a practical and intuitive way was interesting and eye catching, excluding of course the complexity of generating the Action behaviours. We also faced other problems regarding the physics of the books since it was not very easy to arrange them in a bookshelf without dropping some of them.

To conclude, we summarize the main advantages and difficulties of the Library metaphor:

- ✓ Advanced interactive features forming an intuitive way to manage scenegraph.
- ✓ Utilizes capabilities of VR.
- ✗ Does not extend to complex scenegraph trees. Library would have a significant size to store all the Actions resulting in inconvenience and complexity.
- ✗ Difficulties and struggle to handle books due to collisions between them on the shelves.

5.3.3 Design 3: Floppy disks and analog controls

From the library design, we learned the importance of interactive assets within a virtual environment. The system should **assimilate interactive capabilities to transform previously static behaviours into interesting and attractive challenges, utilizing the available modalities.** The library metaphor introduced a new tool to our development arsenal, the idea of interactive 3D assets as parametrized Action modules.

The third and final approach utilizes this feature into a simpler and more efficient design to finalize our road towards the VR metaphor. The main concept behind this implementation focuses around floppy disks and a personal computer. We switched books with floppy disks since they are easier to handle and they give a vintage essence to the application. We replaced the library with a personal computer with analog controls and physical buttons as main inputs. The figure 5.4 illustrates the design of the VR editor along with its various components and floppy disks. To make things clear we will begin by mentioning the analogies between the original scenegraph architecture and the VR Editor. The LSA nodes are represented by floppy drives on the left side of the screen. Action scripts are floppy disks, each one holds the script behaviour that defines the prefabs relative to the Action. There are three types of floppy disk separated with a unique colour; blue disks represent Use Actions, Red disk the Remove Actions and black disks

the Insert Actions. This colour identification was made to visually distinguish different types of Action scripts.

Figure 5.4: The third and final design of VR Editor was influenced from vintage TVs.

Under the floppy drives, we installed an analog knob (influenced from old TV screens) where user can traverse the LSA drives by twisting it left and right. The selected drive is highlighted with an orange colored screen while the rest of the drives remain light green. For example in the picture 5.4, the user has selected the third from the end drive. Right next to the knob, there are two more buttons for addition and deletion of floppy drives. Finally, on the top side of the left panel a square save button generates the scenegraph.

The right panel contains the properties of the selected drive. On the top side, we see the name of the scenegraph node along with its type (Lesson, Stage or Action). The first two node types (Lessons and Stages) have only available the parametrization of their name, whereas Action nodes provide additional features to setup their script behaviour. It is important to remember the analogy: floppy disks == Action scripts. To implement a new script for an action we have to take a floppy disk and insert it in the corresponding floppy drive. This will automatically load a new script according to the type of floppy disk. In a similar way, ejecting a floppy disk from the drive, detaches the script from the Action. Scenegraph architecture supports multiple script nodes into a single Action node, for example the 5.4 figure has two scripts integrated, a remove and an insert script and they are both illustrated as floppy disks icons under the Action name field. Finally yet importantly, one of the main features of VR Editor is the Action customization behaviour. The figure 5.4, illustrates an insert Action behaviour with miniature objects reflecting the interactable and final prefabs. Next to the prefab miniature object lies a save button to generate the selected Action Script following a similar

implementation with visual scripting. We will present more details about this mechanic in the following section.

To conclude, we summarize the main advantages and difficulties of the floppy disk metaphor:

- ✓ Advanced interactive functionalities, featuring virtual floppy disks mechanic.
- ✓ Utilizes capabilities of VR, forming an intuitive and simple design.
- ✓ Scales up to complex scenegraph trees.
- ✓ Powerful mechanic to generate Action scripts on the go.
- ✗ The floppy drive vintage concept does not fit to every training scenario.

Those where the main design components of the third and final VR Editor. In this section, we presented all three approaches, analyzing their advantages, miscalculations and wrong decisions. It is critical in the design phase to backtrack and learn from previous implementations to improve the following versions and reach the optimal result. In the next section, we will present key features of VR Editor and how we managed to overcome specific challenges.

5.4 Training Scenegraph generation through VR Editor

The training Scenegraph has a tree structure, however in a VR environment it is challenging to represent a huge graph, which meant to be rendered on a 2D screen. For this reason, we designed the floppy disks and drive metaphor. In this scenario, we faced a major challenge on how to construct a scalable floppy drive system able to fit multiple drives from complex scenegraph trees. The drives structure has a finite maximum number of floppy drives (8) that can be rendered at the same time. The solution lied on the analog knob, as the main input to traverse through the LSA entities. User can traverse through the drives by twisting the knob and when the second to last selected node is reached; all the drives are translating one-step up. Using the same principle, once the upper limit is reached, the drives are translating one-step down. With this mechanic, our system can support infinite amount of LSA nodes.

Up to this point, we implemented the traversal mechanic to warp around the different modules. The next step is to give user the ability to add or remove LSA nodes at his favor to construct a personalized scenegraph in a similar process as on visual scripting editor. For this reason we designed the addition and deletion buttons that will manage this issue. By pressing the green addition button, a new driver will appear right below the selected one. The new driver will be empty, with no name nor type. The figure bellow illustrates the process of adding and removing floppy drives.

Figure 5.5: (Left) Starting state, (Middle) we are pressing the addition button, (Right) a new empty floppy driver appeared.

The next step is to modify the generated LSA node to our needs; this process is described briefly in the next section. To finalize and generate the scenegraph xml we have to press the save button located above the drive stack, this will trigger the xml generation process that will gather data from all drives, action modules and scripts to summarize the results into a single xml file. To export the scenegraph hierarchy we need the LSA node names along with the corresponding class names of scripts. We retrieve the node names from the drivers list, which always remains up to date for this reason. Specifically for Action drives, except from their names (description), we also need to retrieve their script class name. This information is saved inside the floppy disks we have inserted in the driver. This implementation links everything together, simplifies the generation of scenegraph and visualizes the training scenario through an intuitive and interactive way.

5.5 Generate Actions and parametrization on-the-go

The functionality with the higher impact on the VR Editor is by far the ability to modify and parametrize Actions on the go. All the other features (scenegraph generation, node addition, deletion etc.) are integrated to visual scripting and can also been achieved through native C# code. However, the ability to **modify or even generate new gamified behaviour directly from the virtual environment** is a new feature supported only in VR Editor. This was also the main reason that lead us to implement the VR Editor as an immersive authoring tool from within the virtual environment, to support a coding free development tool and give user the ability to modify existing scenarios while playing.

We begin the parametrization process from the point we left it at the previous section, with a new empty driver (Action node). To make things more interesting we intend to make an Action script. The first task we need to accomplish is the distinction of its type. In visual scripting, we were selecting the node type along with its initialization from a drop down menu. We integrated this functionality for VR Editor in a similar way from the properties panel where we can select the LSA type and set it to Action. After that, we need to fill the script name by pressing

the corresponding name input tab to enable the virtual keyboard so we can type the node's name/description.

Figure 5.6: Modifying the Action description using the virtual keyboard.

The next step is be the most important as well as the most interactive of all, the script generation from the VR Editor. In this process, we will utilize the floppy disks to implement new Action behaviours but also modify existing ones. As an example, we will develop the process of replacing a broken gear from our clock mechanism with a new one. This is a two-step process as we have to implement a Remove Action to take out the broken gear and then and Insert Action to insert the new gear in its place. We will begin by inserting a Remove Action floppy disk (red coloured) into the Action drive and then an Insert Action floppy disk (black coloured), and this is it. The system will register the insertion of floppy disks and two empty Action scripts will appear on properties screen ready for modification. Figure 5.7 shows the process of loading two empty scripts to our Action module.

Figure 5.7: (Left) Inserting the Remove Action floppy disk and (Right) the Insert Action script.

Moving on the script generation pipeline, the next step is to set the parameters

for each action accordingly. Remember that each Action has an initialize method (implemented from IAction interface) which contains key method calls for the initialization of the Action such as the instantiation of the Action's objects. For this reason, we reserved a special place within a cage where a reference to the corresponding prefab will appear. Design-wise, the "miniature" object inside the cage serves the purpose of instant visualization without the need to traverse the project's resources according to the name.

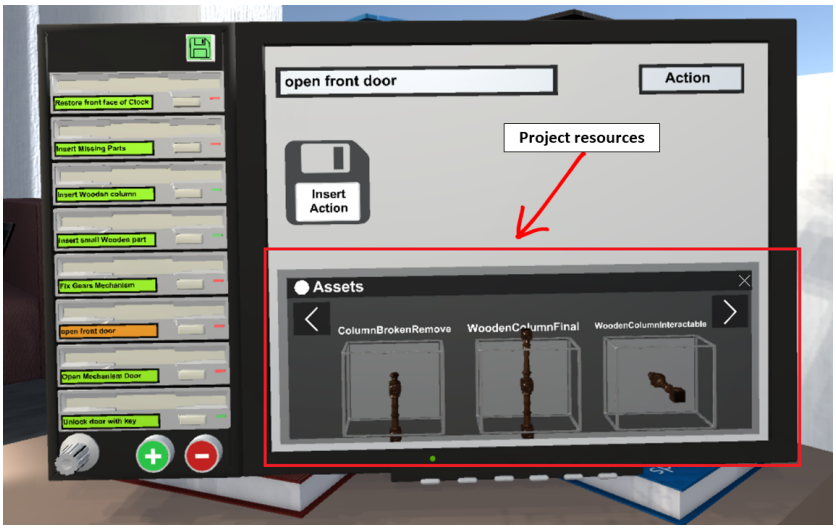


Figure 5.8: User can select a prefab from the resources menu directly from the VR Editor to generate the Action script.

If the script is freshly generated it will not have an object attached to the prefab positions, thus we have to generate one to complete the Action script. To implement this functionality we introduced a file manager module for VR Editor, influenced from the Unity's demo announcement for their custom VR editor. The embedded file manager gives user the ability to traverse through the project's saved models and select the one needed for each case. In this case, we have to link the Remove Prefab object to generate the action script and take out the old and rusty gear from our antique clock. Manipulating the position of instantiated 3D assets is meaningful to the content creation as it is common to place an object in a specific location from the 2D computer monitor but from the VR perspective, the object may not fit well.

The next and final step is to generate the action script using the VR Editor. This process works in the same way as the generation of the script from the visual scripting tool since we used similar abstract mechanics for easier code maintenance and reusability of important patterns. After we trigger the command to generate the selected Action script, VR Editor seeks to find all the modified elements and generates a new script in our favour with our customizations. Once again,

Action Prototypes played important role to the generation of script since their encapsulation capabilities made the implementation clear and straightforward.

To conclude, VR Editor proposed a new interactive design not only to manipulate 3D assets to set the virtual scene but also to generate gamified behaviours without writing a single line of code. This feature had a massive impact on content creation, as a script generator and data visualization at the same time. **With this tool, users are not just observers, the can modify the training scenarios on the go, implement new ideas, correct wrong Action behaviours or even generate whole missing parts at their favour without specialized programming knowledge.** The game turned into a one-man show!

Chapter 6

Results: Antique clock training restoration

For the purposes of this dissertation, we developed a training scenario to evaluate the scenegraph architecture as an authoring tool and test its functionalities and components. The main concept behind the scenario was the restoration of an antique pendulum clock. We chose a manual labor as it reflects better the capabilities of an interactive VR system. Before we decide to move on with this scenario, we also thought other possible alternatives like the repair of a car engine or the restoration of a cultural heritage object (painting or statuette). However, we have to underline that the developed application is not the final product as this project was not about the content creation but the development of the authoring tool (scenegraph system) with which we are able to generate gamified training scenarios. In the following sections, we will present the main components of the application and further customizations beyond the proposed scenegraph tools.

6.1 The training scenegraph

To restore the antique clock user first needs to replace some missing parts (wooden columns) at the front face and then turn the clock to examine the gear mechanism. However, there is a door at the backside of the clock, protecting the mechanism. To open the small wooden door, user needs to take a key and unlock the door to have access at the gears. After accessing the gears mechanism, he will face a broken gear and he will be asked to replace it with the new one. This task concludes the work at the backside and now he needs to turn the clock back to its starting position and open front glass door, which does not require a key this time. The next step is to insert the missing hour and minute hand and finally jump-start the pendulum to hear the clock ticking again. The mentioned steps are illustrated using the visual scripting editor in the picture 6.3 .

With the first sight, we can identify that the training scenario has a moderate

Figure 6.1: Replace wooden column action. The green holographic material highlights the correct placement of the object.

Figure 6.2: Replace broken gear action. To accomplish this Remove Action user needs to take the broken gear out of the clock mechanism.

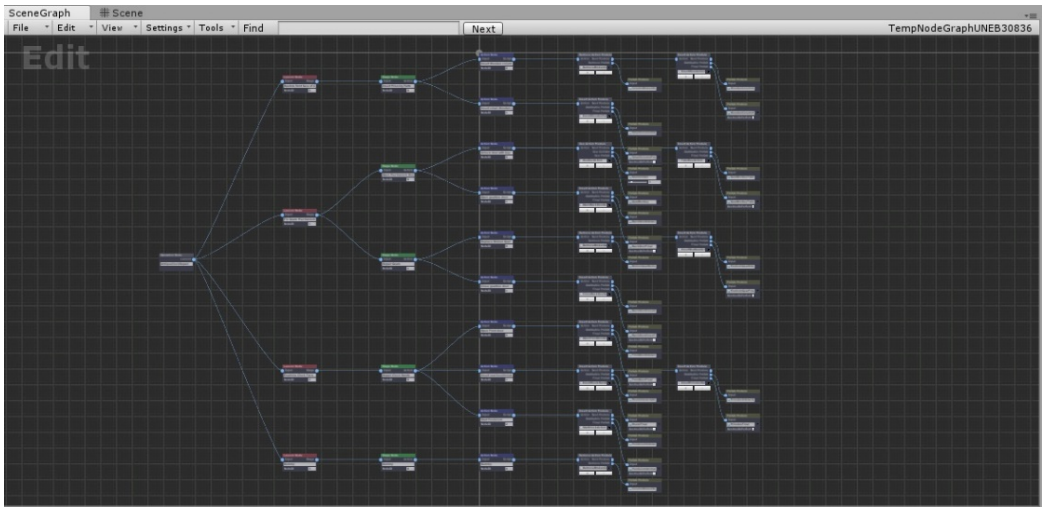


Figure 6.3: The scenegraph tree containing the LSA nodes of the clock restoration.

number of nodes and various tasks to accomplish. However, through the visualization from the visual scripting editor we can clearly identify the LSA nodes from their unique colors and practical tree-like structure. The majority of the Actions were implemented using the visual scripting tool to construct the main skeleton of the project with simple behaviours. After implementing the basic Actions, we utilized VR Editor for further customization on specific behaviours that had a poor user experience due to inconvenient positioning of the key 3D assets. VR Editor served well to customize the Actions and achieve a better experience. The last tool in the pipeline is the actual C# scripting for final touches like animation management, sound effects and any other business, which are not supported by Prototypes.

6.2 Extending action behaviour

Scenegraph architecture along with Action Prototypes form a powerful tool for the following reason. Visual Scripting and VR Editor are located at the high-level hierarchy of the system to manage the layers below that maintain the core functionalities and the engine of the training scenarios. They encapsulate fundamental behaviours into tools available to developers and at the same time, they allow further parametrization to enhance the game experience. To make things more clear we will take as example the last Action of the clock restoration simulation where user needs to move the pendulum at a higher angle to jump-start the gear mechanism. A simple version of this behaviour would be a scenario where user lifts the pendulum with an Insert Action and then the operation ends. This Action can be easily generated using the Visual Scripting tool or the VR Editor directly from

the virtual environment; and that would be it. The Insert Action is finished, the training scenario is completed and everyone is happy. Everyone except from the trainee. There is nothing wrong with this implementation since this would be the process in real life as well. However, we can implement a better scenario than this one. Remember, lifeless applications are the ones with advanced functionality and intuition (we have those features directly from the scenegraph architecture) but with limited interaction and no embodied cognition. The solution to our problems lies on the abstraction of Action Prototypes.

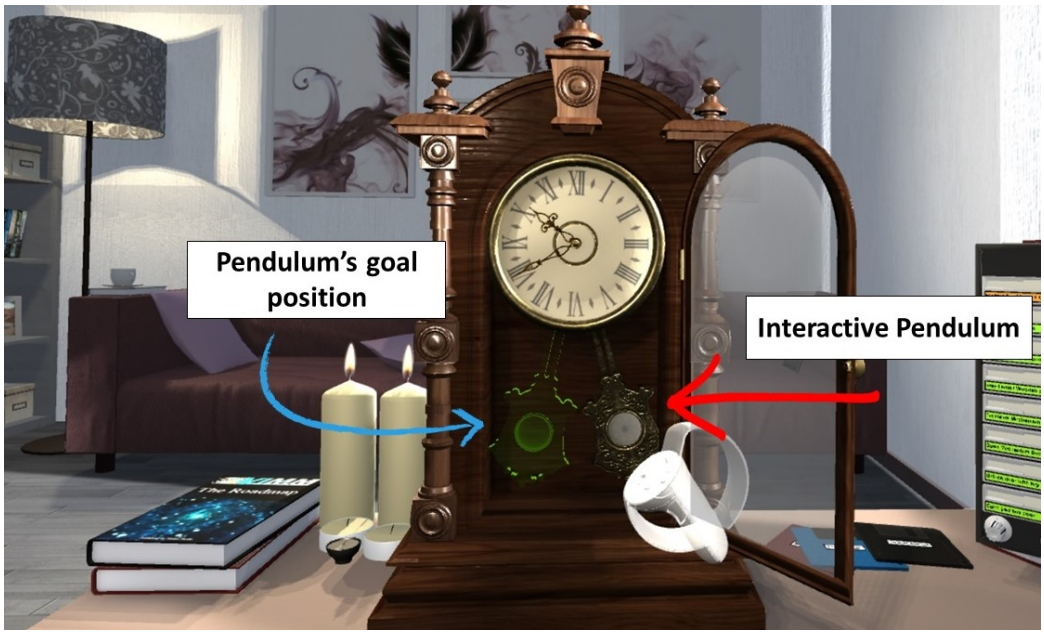


Figure 6.4: The insert pendulum Action. On the right is the hologram indicating the goal position as user tries to jumpstart the clock.

That was a major challenge for the development of the project. Our idea was to build a strong base layer using Action Prototypes to support basic Action behaviour and on top of that provide implement additional authoring tools to speed up the content creation an minimize debugging. Back to our example, we wanted to make the jumpstart of the clock more interacting and realistic. The code snippet 6.1 is the Insert Action script, the visual scripting tool auto-generated from the input we gave to the editor.

```
public class StartClockAction : InsertAction
{
    public override void Initialize()
    {
        SetInsertPrefab("Lesson2/Stage0/Action2/PenInteractable",
```

```
                        "Lesson2/Stage0/Action2/PendAnimatedFinal");
        base.Initialize();
    }
}
```

Listing 6.1: Auto-generated Insert Action script.

It seems simple but it is also very condense in data. This is all we need, a single line of code inside Initialize method is able to generate the gamified behaviour we want to lift the pendulum and complete the Action. Nonetheless, we can improve the Action with additional features from Action Prototypes. The first thing to consider is to add an animated pendulum right after we completed the Action to visualize the clock mechanism. For trigger additional behaviour right after the completion of an Action, we utilize the Perform method declared in IAction interface. In this case, we have to override the Perform method to instantiate the animated pendulum and replace the static one. To increase the immersion, we will add a ticking sound effect in the same way we instantiated the animated pendulum. For the sound effect, we will utilize the embedded Voice Actor, the module that manages all sounds in our application. The code snippet 6.2 shows the updated version of the Action script after overriding the Perform method.

```
public class StartClockAction : InsertAction
{
    public override void Initialize()
    {
        SetInsertPrefab("Lesson2/Stage0/Action2/PendInteractable",
                        "Lesson2/Stage0/Action2/PendAnimatedFinal");

        base.Initialize();
    }


    public override void Perform()
    {
        pendulum.GetComponent<Animator>().enabled = true;
        pendulum.GetComponent<ClockSounds>().PlaySound();

        base.Perform();
    }
}
```

Listing 6.2: Parametrized Action script with overridden Perform method to support Audio clips.

To summarize, the generated training scenarios from visual scripting and VR Editor, provide basic behaviours to match the project needs. For additional customization, the developer should implement the enhanced features following the

IAction interface and Action Prototypes by modifying the C# code generated from the mentioned tools.

6.3 Alternative paths in practice

Alternative paths offer user the ability to choose among different dilemmas on how to proceed with critical parts of the training scenario. In-game decisions affect the upcoming Actions due to dynamic changes in the scenegraph tree. As mentioned before, scenegraph is a dynamic tree and can modify itself runtime according to the user's Actions and decisions. For this reason, we utilize the Alternative Path mechanic to provide a dynamic training scenario, challenging even for the experts. In this section, we will discuss the Alternative Action we implemented for the antique clock restoration.

The Alternative scenario is the following: At the point we have replaced the broken gear, we are facing with the dilemma of a) Oil the gears mechanism using the oil tin or b) Close the door and move on. The correct path is the first one as the gears are jammed and they will not turn unless we add some lubricant. However, if user follows the second path and complete the Action, which closes the back door, the Alternative Path will trigger right away. Following the second path, when he/she will reach the end of the training scenario, the clock will not start due to the jammed gears. At this point, he/she needs to go back to examine the gears mechanism and insert oil to fix the issue. After that, the clock will start running normally. The diagram 6.5 illustrates the two available paths. The red colored path represents the Alternative Path and the green the correct one.

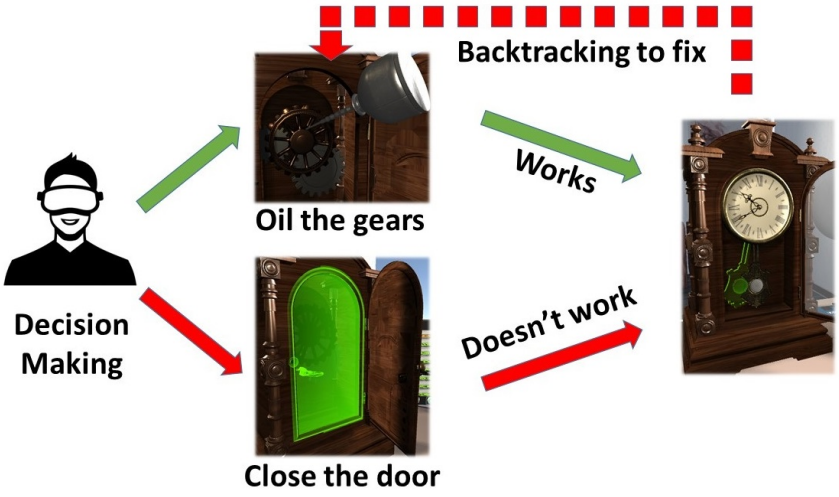


Figure 6.5: The decision-making timeline.

In more detail, to implement this decision making feature we need to set the scenegraph properly to trigger the Alternative path the right time. We will begin by generating the decision making Action through the Visual Scripting editor. The Action that describes better the gear's oiling is a Use Action since we need an object to interact with another one for a predefined amount of time. Figure 6.6 illustrates the decision-making Action forming a parallel Action, splitting into two segments, the first one is the OilGearAction (Normal Path) and the second one the CloseBackDoorAction (Alternative Path). Both Actions are initialized simultaneously to choose which path to follow. If user completes the Use Action by oiling the gears, the scenegraph will continue normally. However, by closing the clock's door, the Alternative Path will trigger causing the scenegraph to adapt reflecting the wrong decision.

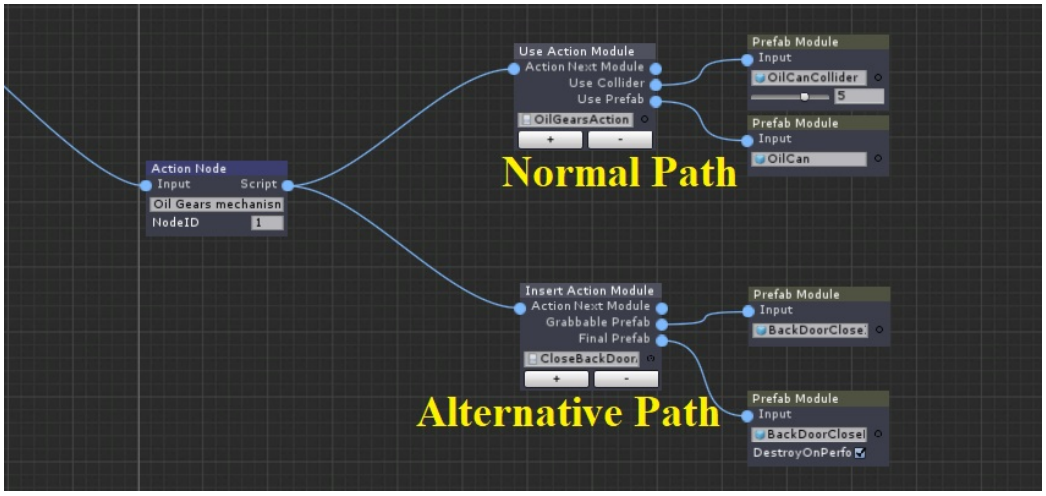


Figure 6.6: Visualizing in scenegraph a decision making Action with a two directional node.

To finalize the decision-making process, we need an Alternative Action Node that will be added to scenegraph if the user follows the wrong path. In this situation, we will use an Alternative Stage since we have only three additional Actions to insert. This Stage contains the backtracking to lubricate the gears mechanism. Figure 6.7 shows the Alternative Node in the scenegraph editor. As it seems, it is not connected to the rest of the tree because Alternative Nodes are not a part of the main pipeline. To work properly we need to set related parameters on the Stage Node to link it with the close door Action. **In this way, if user proceeds without lubricating the gears, the Alternative Path will trigger adding the Stage we just implemented to scenegraph forcing the user to back-track and correct his wrong decision.**

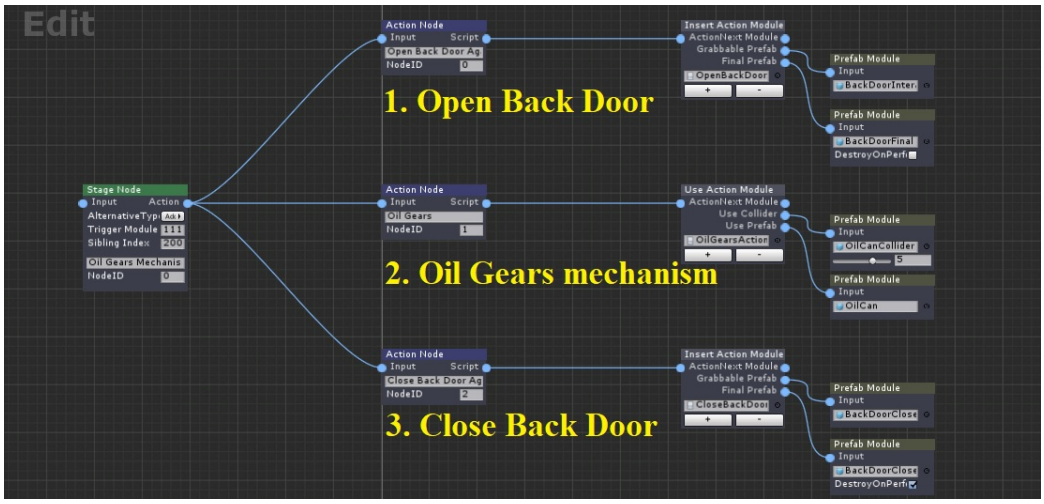


Figure 6.7: The Alternative Stage in Visual Scripting Editor.

Chapter 7

Evaluation

7.1 Self-evaluation

7.1.1 Training Scenegraph architecture

The training Scenegraph represents in a practical and easily visualized way a training scenario. We chose to follow a tree data structure instead of a linear (list, array or queue) to extend its capabilities into a dynamic, modular entity. Scenegraph encapsulates the necessary data for a training scenario and stores it xml format for further modifications. In more detail, scenegraph contains data relative to each LSA node such as names and action description as well as references to the Action scripts and additional information for parallel Actions.

The training scenario consists of several tasks user needs to accomplish to finish the simulation. A general observer is responsible to manage those Action behaviours, decide which one to instantiate and for how long each one will be available. This operation observer encapsulates all the needed to Initialize, Perform and Undo each Action along the training scenario continuum.

The main advantage in scenegraph architecture lies on the visual separation of specific logical parts in the training scenario. If instead of Lesson, Stage and Action nodes we supported only the latter ones, the whole structure would be simpler due to reduction in the number of nodes but we would not have a mechanic to separate Actions into groups. Another advantage we get by utilizing the scenegraph tree is the convenient runtime pruning of the tree in alternative Actions. Alternative paths modify the training scenegraph tree by adding and deleting parts of the training scenario to deviate the normal path if user decides to follow a different approach or due to critical errors. In addition, the tree structure improves Action traversal utilizing recursive path finding mechanics fully compatible with scenegraph trees.

On the other hand, this system may not perform well in complex alternative paths. The backtracking mechanic of scenegraph does not support nested alternative paths due to the limited number of data we gather to jump from one alternative path to another. However, it supports multiple parallel actions that

can trigger unlimited number of alternative paths giving user the freedom to pave his own way through the training scenario.

To summarize, scenegraph offers a great tool for visualizing, traversing and managing training scenarios. Below there is a list with the main advantages and disadvantages of scenegraph design pattern.

- ✓ Advanced visualization, ideal for complex training scenarios.
- ✓ Logical separation of Actions into context groups.
- ✓ Scales up to support complex trees.
- ✗ Currently does not support nested Alternative Paths.

7.1.2 Action Prototypes

In this project, we introduced Action Prototypes as a novel design pattern for VR experiences. The main concept is to prototype commonly used patterns that derive from real life behaviours (insertion of objects into predefined positions, use of tools etc.) to generate reusable properties and apply them in VR. Action Prototypes implement the IAction interface to encapsulate and extend its functionalities. Until now, the process of developing a VR experience was a straightforward pipeline employed by many developers and designers to implement interactive applications from the beginning using a game engine. The game has now changed. Developers can utilize Action Prototypes to implement interacive behaviours following the scenegraph pipeline. The enhanced functionalities and tools supported in our system simplify the content creation while reducing the needed developer.

For the needs of the project, we created three Action Porotypes, the Insert, Remove and Use Action. However, due to the scalability of this system, developers can extend the library of Prototypes to implement new behaviours according to their training scenarios. This is one of the main features of the system; it is **scalable** and **adaptive** to any new training scenario with minimal changes. For example, to create a new prototyped behaviour we have to inherit the BasePrototype class and implement the virtual methods of IAction and this is it, our new prototype is fully compatible with the rest of the architecture.

Action Prototypes share common mechanics due to the inheritance from BasePrototype. For this reason, Action scripts have limited vulnerability to bugs and errors from the side of architecture. Of course, developers have their responsibility to program their Action according to the prescribed standards but they only manage the front end of Actions (the scripts) that is responsible for the unique behaviours. In contrast, content developers do not need to make any changes to the back end that runs the system. Updates to critical parts of the code are always needed but only under supervision and in case of a system update (addition of new Prototypes). In this way, the core engine of the system stays intact, leaving only the front end available to parametrization from content developers.

To conclude, Action Prototypes enhance the system with scalability, reusability and produce a safe core engine for a stable development. They simplify the content creation providing developers with a starting point to work on to avoid rewriting the same long code sequences all over again. A few lines of code are needed to generate gamified creations since the rest of the system runs back end. Below we collected the most important points that characterize Action Prototypes.

- ✓ Speeds up the VR content creation through Prototyped behaviours.
- ✓ Reusability, scalability and adaptation to new scenarios.
- ✓ Protects core engine from bugs while leaves freedom to implement customized behaviours at the front end.
- ✗ Supports only Action scripts that are linked with a Prototype.

7.1.3 Visual Scripting editor

On top of our training scenegraph, we built a Visual Scripting editor to visualize the training scenarios in a consolidated map with linked nodes and parametrized fields. Following a holistic approach, we went even further, extending the Visual Scripting tool to generate Action scripts via the editor. Utilizing this tool, developers can set specific parameters from menus and drop down UIs in the editor and the system will auto generate the code which reflects the Action behaviour. Visual scripting is one of the strongest points in this dissertation since it is the connected tissue with all the supported and prototyped components. One tool to rule them all. Until now, Unity3D engine does not support a native visual scripting system, despite the fact that Unreal engine does. However, in the next update, (2019.3) Unity will launch a Visual Scripting tool embedded in the engine. We are glad to finish ours a few months before Unity's official launch.

In more detail, Visual Scripting tool is the front-end system that encapsulates Action Prototypes. It solved a major problem of scenegraph architecture, the complex xml files. It was difficult to maintain training scenarios bigger than four or five Lessons due to the length of their xml file. After identifying that the previous model does not scale up, we developed visual scripting with initial task to visualize the scenegraph xml and provide tools to update the tree using an editor rather than the xml file. Back then, Visual Scripting seemed like an intermediate step between the scenegraph and the Action Prototypes and then everything linked together prefect. We designed the second version of Visual Scripting editor to auto generate Action scripts directly from the scenegraph tree and the editor tools.

Visual Scripting transformed our system into an SDK platform due to the versatility of its components. Being able to generate scripts without writing a single line of code is truly meaningful for content creation platforms since it boosts the production, reduces bugs from untested code and widens the target group of the platform. Our platform does not require strong programming skills or computer graphics specialization just because of the proposed Visual Scripting editor.

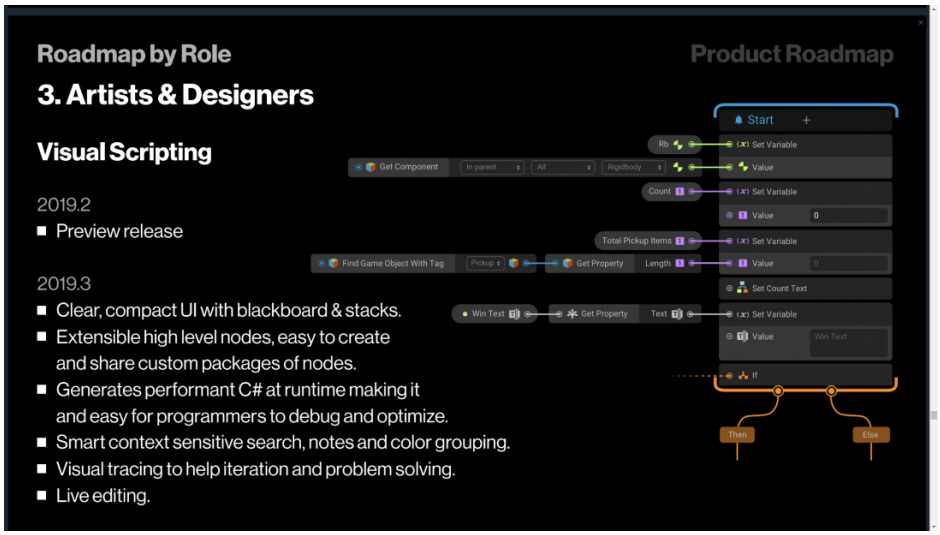


Figure 7.1: Unity's official Visual Scripting editor release note from the product roadmap. Available on the next update.

Unexperienced developers or even artists and designers with minimal algorithmic knowledge are able to produce training scenarios without writing scripting code, just by moving nodes and selecting data from dropdown menus in the editor. In addition, due to the abstraction of scenegraph architecture, developers can extend the auto-generated scripts by adding customized behaviours according to their needs. This is an important aspect of this project since it reflects the scalability of the system from simple auto generated VR scenarios to complex behaviours generated by the visual scripting and then parametrized by the developer.

To summarize, Visual Scripting introduced great features in our system and had great impact on the content creation and management of VR application. Below we state the main key points of this authoring tool.

- ✓ Advanced visualization and management features.
- ✓ Generate VR experiences without writing a single line of code.
- ✓ Developers can extend auto-generated scripts with additional features to fit each application.
- ✗ Not the optimal visual representation for Alternative Action Nodes.

7.1.4 VR Editor

Up to this point, we can generate automated scripts, manage and visualize the scenegraph tree with the Visual Scripting editor. However, we went beyond that overpassing our expectation for this project since the development of a VR Editor

was not in our initial plans. Editing and modifying the training scenarios from within the virtual environment imparts great flexibility in the development process. Utilizing VR Editor, not only developers but also experienced users can take part in the content creation, modifying existing Actions or recreating complete training scenarios at once. To the best of our knowledge, there is no similar system in the market that promises generation of VR experiences from the virtual environment.

Figure 7.2: VR Editor unlocked the developing while playing feature.

What makes VR Editor a great tool? The answer lies on its ability to encapsulate the functionalities of visual scripting while enabling object visualization related to the training scenario. When developing a VR experience its always important to consider that the medium in which the application is meant to be played is a highly interactive and immersive system. This why a VR Editor tool was proven beneficial for VR development since it gives developers the ability to generate experiences while immersed in the virtual world and not from a 2D screen writing code. This tool is a game changer for VR development.

Another interesting feature of VR Editor is the implemented Assets Library. Instead of visualizing the imported 3D assets and models from a desktop screen, both developers and especially artists can visualize their work directly where it should be, in the virtual world. This ability is considerably important to avoid backtracking and multiple updates due to 3D asset misplacement that occurs very often with conventional ways of editing.

In addition, VR Editor makes the development process easier and more interactive than classic coding. It is well advised to change some habits from time to time and a VR Editor is the perfect alternative to C# coding in this case. Of course, it is not so powerful and versatile being at a higher level of architecture in the front-end of our system but it is surely a unique way to implement new ideas and scenarios via this immersive tool.

To conclude, VR Editor introduced an interactive and immersive way to recreate Action behaviours through the scenegraph architecture. Below we summarize the basic points of the system.

- ✓ Interactive and immersive visual creation of gamified training scenarios.
- ✓ Visualize 3D assets and objects directly from the virtual environment.
- ✓ Generate Action behaviours on the go without advanced programming skills of computer graphics background.
- ✗ Currently does not support visualization of Alternative Paths and Parallel Actions.

7.2 User-based qualitative evaluation

We organized a user-based qualitative evaluation experiment to examine and quantify the effectiveness and impact of our system but to also gather useful data and comments for future updates. In more detail, the experiment conducted from 12 participants (10 male, 2 female), half of them were familiar with computer programming and VR whereas the other half did not have programming background or any familiarity with VR. We split our participants in this way to examine the differences between programmers and non-programmers and how this variable will affect their interaction with our system. In the begin of the experiment, participants rate themselves on their familiarity with VR and their programming skills (figure 7.3).

The experiment was separated in four sessions to evaluate the different components and functionalities of our system. The first part was to run a VR training scenario generated from our system. For this reason, we developed a VR training scenario regarding the restoration of an antique clock where users asked to complete interactive tasks to make the clock working again. The next session, evaluates the capabilities of Visual Scripting, participants were asked to generate a Use and a Remove Action into the VR scenario. After that, they had to utilize VR Editor to change the positions of certain interactive objects and generate an Insert Action directly from the virtual environment. Finally we had an open discussion with the participants to give us valuable feedback for all the implemented tools.

We begin with the evaluation of the generated training scenario. In the following tables, the completion time is measured in mm:ss format, the evaluation of mechanics in 0-10 range and the Help metric illustrates amount of times where participants asked for hints don't knowing how to proceed further. From table 7.1 we acquire that non-programmers faced more difficulties on completing the scenario since their average completion time and times asked for help are higher. In contrast they have higher rates in the quality of the overall experience and the educational value of the application.

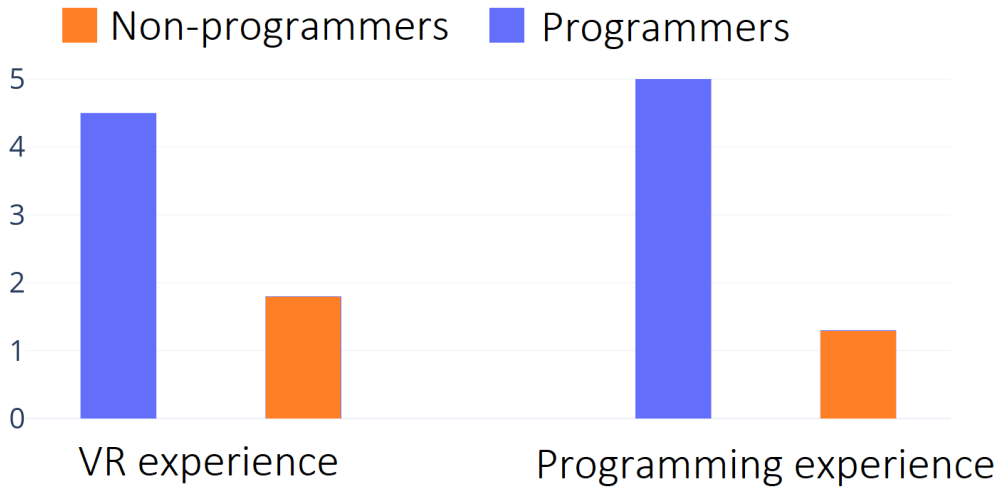


Figure 7.3: Data acquired from the qualitative user evaluation regarding their experience (range 0-5).

Table 7.1: Evaluation of training scenario

	Programmers	Non-programmers
Completion Time	2:14	3:53
Help	0.8	1.6
Experience quality	7.0	8.5
Educational value	7.6	8.8

We continue with the evaluation of Visual Scripting tool. In this phase, we explained all the possibilities and functionalities of Visual Scripting and how to generate scripts automatically. After this brief explanation, users were asked to implement a new Use and Remove Action utilizing the scenegraph editor. From table 7.2 we identify that both programmers and non programmers asked more often for help in the generation of Action scripts than the training scenario. In addition, it is clear, regarding the implementation of Actions, programmers were more familiar with the tools and managed to complete the tasks five minutes earlier. An interesting realization is the fact that non-programmers rated the overall experience higher than the programmers although they gave lower scores on the implementation of Action Scripts.

Table 7.2: Evaluation of Visual Scripting

	Programmers	Non-programmers
Completion Time	12:45	17:53
Help	1.3	2.0
Use Action	8.5	6.6
Remove Action	8.8	7.1
Overall Experience	7.2	8.6

The final evaluation was for VR Editor. We introduced the new tool to the participants with a brief in the beginning to ask them later to implement two tasks. Table 7.3 shows some interesting results from the evaluation of VR Editor. First of all, participants asked for help in a higher rate that the Visual Scripting tool. In addition, non-programmers completed the tasks four minutes earlier using the VR Editor than the Visual scripting tool. The overall experience has similar values with the one from the Visual Scripting tool.

Table 7.3: Evaluation of VR Editor

	Programmers	Non-programmers
Completion Time	12:15	13:21
Help	1.6	3.0
Object Reposition	7.3	7.1
Insert Action	7.1	6.9
Overall Experience	7.4	8.5

The next session was an open discussion about our system to get feedback from the participants. Some of them suggested to implement additional features on the Visual Scripting editor focusing more in the User Experience that functionality.

Other participants commented on the difficulty they had to understand the interaction with some modules from VR Editor since there was not a tutorial while playing the application. In addition, some programmers complained they could not assimilate the generation of Actions through VR Editor due to the overload of information in a single window. A positive feedback we received from some non-programmers was the feeling of accomplishment when the managed to generate a simple VR application without knowing programming. Finally, some programmers mentioned they enjoyed the process of Action generation from the Visual Scripting editor due to the visualization of the complete scenario in a single window.

Chapter 8

Conclusions

8.1 Summary

The main purpose of this Master's thesis was to design an authoring tool based on VR design patterns to generate gamified training scenarios. To the best of our knowledge, there is no such system available in the research or the industry that utilizes visual scripting and VR Editor Tools to auto generate simulation-based VR gamified training. In this section, we will summarize the achievements and main novelties of the proposed system.

The training scenegraph is better described as an authoring tool in the form of a dynamic tree, which generates gamified training scenarios. On top of the proposed architecture, we developed three different ways to populate prototyped behaviours as interactive and immersive behaviours. Those tools are the **a) Visual Scripting**, the **b) VR Editor** and finally the **c) C# scripting through prototyped patterns.** Each one serves a different role in the pipeline of content creation by overcoming specific challenges and reducing dramatically the time and manpower needed to generate training scenarios. The Action Prototypes define the main structural element of the scenegraph system offering an abstract layer to develop unique Actions via the proposed tools.

8.2 Comparison of implemented tools

In this section, we will compare the three different ways we implemented to create Action behaviours. Those authoring tools are the Visual Scripting, the VR Editor and the classic C# coding along with Action Prototypes. Which is the best tool for the creation of gamified training scenarios? The answer follows one of the most popular clichés in computer science bibliography. Of course,no one; each one has its own characteristics to solve unique programming challenges and fit in specific applications.

In more detail, C# coding along with Action Prototypes was designed to offer great flexibility and abstraction in programming Action behaviours. Developers

are free to extend this tool according to their needs and implement various interactive Action behaviours. This is the reason why we are saying this is architecturally the lowest level, front-end module of our system. Moving hierarchically upwards, we have the visual scripting editor, to visually represent and modify the training scenarios. This system is great to use to construct the initial skeleton of the project and then generate simple Action scripts. However, its capabilities are currently limited, mainly focused on fundamental Action scripts with minimal alternations between them. It is well advised to use visual scripting along with Prototypes and C# coding for further modification and customization. Finally, on top of the architecture lies the VR Editor, offering advanced visualization of 3D assets as well as creation of gamified scripts on the go while playing the training scenario. In a similar way as visual scripting, VR Editor generates simple Action scripts, thus C# coding can be used to extend their interaction for a more realistic finish.

To conclude, each tool has its own advantages solving specific challenges on the development pipeline. The best-case scenario is to utilize each one in the development areas where has the strongest functionalities to take full advantage of the tool's capabilities. Table 8.1 summarizes key aspects from all three authoring tools according tools, comparing them in different scenarios.

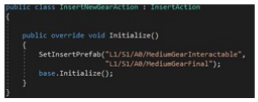
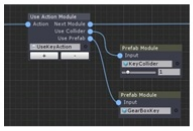
	 **C# code, Prototypes**	 **Visual Scripting**	 **VR Editor**
Advanced Visualization	✗	✓	✓
Script auto generation	✗	✓	✓
Custom/Parametrized Action scripts	✓	✗	✗
Coding free	✗	✓	✓
Virtual scene management	✗	✗	✓

Table 8.1: Comparison of implemented authoring tools.

8.3 Future Work

In the future, we aim to include computational vision and machine learning capabilities to extend the authoring capabilities of our system. We would like to capture the movements of the trainer directly from the virtual environment and

use this information as a basic example for the trainee to follow. Another idea is to collect this data through video from a real life scenario by monitoring the trainer and afterwards processing the data using machine learning to extract important key points for our training scenario.

We also aim to support the ability to edit Action scripts from visual scripting and VR Editor. Currently, both tools can only generate the whole content of a script in a new .cs file meaning that any previous customization will be overridden. In the next version, we aim to publish a new reflection system that would update only specific parts of a script leaving the rest of the code intact. This would be essential since most of the time we edit small parts of a script by replacing prefab paths or include new customizations to Perform and Update methods.

Another optimization we aim for is the encryption of scenegraph xml for security reasons. The xml file represents the core data structure of a training scenario and should be treated carefully. In addition, both Visual scripting and VR Editor illustrate the scenegraph tree in detail, thus editing the scenegraph directly form the xml file is deprecated. Moreover, we would like to create Unity packages containing the encrypted xml file along with the project resources (3D models) and scripts to compress training scenarios into one single file. This methodology will separate the platform from the content leading to the simplification of distribution and exploitation of the developed applications.

Regarding the Visual scripting and VR Editor tools, an extension for the near future is the implementation of a real time visualization system like a workflow that would be exploited to monitor and debug the data traveling from one object to another. Currently the system visualize in a simple way both scenegraph and Actions but there is no visual information for realtime events like in which step we are currently located or important parameters of an Action.

Future research should further develop Action Prototypes to increase their number of Prototypes (currently: Insert, Use and Remove). More Prototypes meaning more development flexibility and versatility on Action development. In addition, we would like to implement nested alternative paths since currently they are not supported. Alternative Paths are the key to a dynamic training scenario with different supported approaches and multiple variations, thus we would like to upgrade this feature in next versions. Future research could examine the expansion of our training scenarios with additional VR content regarding the restoration of Cultural Heritage artifacts to study the impact of our system from a multidisciplinary approach.

Finally, we aim to extend our qualitative evaluation by conducting a professional usability evaluation in cooperation with the Human Computer Interaction (HCI) Laboratory of ICS-FORTH. Our updated evaluation will be based on standard methods [37], [48], [50] to efficiently quantify and measure the user experience and functionalities of our system.

Bibliography

[1] M. Abrash. What VR could, should and almost certainly will be within two years. *VALVE*, 2014.

[2] Asmaa Alsumait and Zahraa S. Almusawi. Creative and innovative e-learning using interactive storytelling. *International Journal of Pervasive Computing and Communications*, 9(3):209–226, 2013.

[3] Margaux Lhommet Andrew Feng, Ari Shapiro and Stacy Marsella. Embodied autonomous agents. pages 347–361, 2014.

[4] Nathan Beattie, B Horan, and Sophie Mckenzie. Taking the LEAP with the oculus HMD and CAD - plucking at thin air? *Procedia Technology*, 20:149–154, 12 2015.

[5] Stéphane Bouchard, Stéphanie Dumoulin, Geneviéve Robillard, Tanya Guitard, Evelyne Klinger, Héléne Forget, Claudie Loranger, and Francois Xavier Roucaut. Virtual reality compared with in vivo exposure in the treatment of social anxiety disorder: A three-arm randomised controlled trial. *The British journal of psychiatry : the journal of mental science*, 210, 12 2016.

[6] R. Bouville, V. Gouranton, T. Boggini, F. Nouviale, and B. Arnaldi. five : High-level components for developing collaborative and interactive virtual environments. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 33–40, March 2015.

[7] Cédric Buche, Ronan Querrec, Éric Maffre, Pierre Chevaillier, and Pierre De Loor. MASCARET: multiagent system for virtual environment for training. In *VRIC 03*, pages 159–164, France, 2003.

[8] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[9] Marcello Carrozzino and Massimo Bergamasco. Beyond virtual museums: Experiencing immersive virtual reality in real museums. *JOURNAL OF CULTURAL HERITAGE*, 11 - 4:452–458, 10 2010.

[10] S.H. Choi and H.H. Cheung. A versatile virtual prototyping system for rapid product development. *Computers in Industry*, 59(5):477 – 488, 2008.

[11] Guillaume Claude, Valérie Gouranton, Rozenn Berthelot, and Bruno Arnaldi. Short paper: seven, a sensor effector based scenarios model for driving collaborative virtual environment. 12 2014.

[12] Desi Dwistratanti Sumadio and Dayang Awang Rambli. Preliminary evaluation on user acceptance of the augmented reality use for education. *Computer Engineering and Applications, International Conference on*, 2:461–465, 03 2010.

[13] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[14] Anthony Elliott, Brian Peiris, and Chris Parnin. Virtual reality in software engineering: Affordances, applications, and challenges. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 547–550, Piscataway, NJ, USA, 2015. IEEE Press.

[15] J. F. Heafner Ellis, T. O. and W. L. Sibley. The grail project: An experiment in man-machine communications. *RAND Corporation*, pages RM–5999–ARPA, 06 1969.

[16] Anthony Gallagher, E Matt Ritter, Howard Champion, Gerald Higgins, Marvin Fried, Gerald Moses, C Smith, and Richard Satava. Virtual reality simulation for the operating room: Proficiency-based training as a paradigm shift in surgical skills training. *Annals of Surgery*, 241:364–372, 02 2005.

[17] Pedro Gamito, Jorge Oliveira, Carla Coelho, Diogo Morais, Paulo Lopes, José Pacheco, Rodrigo Brito, Fabio Soares, Nuno Santos, and Ana Filipa Barata. Cognitive training on stroke patients via virtual reality-based serious games. *Disability and Rehabilitation*, 39(4):385–388, 2017. PMID: 25739412.

[18] Franck Ganier, Charlotte Hoareau, and Jacques Tisseau. Evaluation of procedural learning transfer from a virtual environment to a real situation: a case study on tank maintenance training. *Ergonomics*, 57(6). PMID: 24678862.

[19] Mark Giereth and Thomas Ertl. Design patterns for rapid visualization prototyping. pages 569–574, 08 2008.

[20] Scott Greenwald, Alexander Kulik, André Kunert, Stephan Beck, Bernd Froehlich, Sue Cobb, Sarah Parsons, Nigel Newbutt, Christine Gouveia, Claire Cook, Anne Snyder, Scott Payne, Jennifer Holland, Shawn Buessing, Gabriel Fields, Wiley Corning, Victoria Lee, Lei Xia, and Pattie Maes. Technology and applications for collaborative learning in virtual reality. In *CSCL*, 2017.

[21] S. Göbel, L. Salvatore, and R. Konrad. Storytec: A digital storytelling platform for the authoring and experiencing of interactive and non-linear stories. In *2008 International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution*, pages 103–110, Nov 2008.

[22] J. Hamari, J. Koivisto, and H. Sarsa. Does gamification work? – a literature review of empirical studies on gamification. In *2014 47th Hawaii International Conference on System Sciences*, pages 3025–3034, Jan 2014.

[23] Jeffrey Heer and Maneesh Agrawala. Software design patterns for information visualization. *IEEE transactions on visualization and computer graphics*, 12:853–60, 09 2006.

[24] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 421–430, New York, NY, USA, 2005. ACM.

[25] Jessica Hooper, Eleftherios Tsiridis, James E. Feng, Ran Schwarzkopf, Daniel Waren, William J. Long, Lazaros Poultsides, William Macaulay, George Papagiannakis, Eustathios Kenanidis, Eduardo D. Rodriguez, James Slover, Kenneth A. Egol, Donna P. Phillips, Scott Friedlander, and Michael Collins. Virtual reality simulation facilitates resident training in total hip arthroplasty: A randomized controlled trial. *The Journal of Arthroplasty*, 2019.

[26] I-Chun Hung, Lung-I Lin, Wei-Chieh Fang, and Nian-Shing Chen. Learning with the body: An embodiment-based learning strategy enhances performance of comprehending fundamental optics. *Interacting with Computers*, 26:360–371, 07 2014.

[27] Marco Iosa, Giovanni Morone, Augusto Fusco, Marcello Castagnoli, Francesca Romana Fusco, Luca Pratesi, and Stefano Paolucci. Leap motion controlled videogame-based therapy for rehabilitation of elderly patients with subacute stroke: a feasibility pilot study. *Topics in Stroke Rehabilitation*, 22:1074935714Z.000, 02 2015.

[28] Stavros Kateros, Stylianos Georgiou, Margarita Papaefthymiou, George Papagiannakis, and Michalis Tsioumas. A comparison of gamified, immersive VR curation methods for enhanced presence and human-computer interaction in digital humanities. *International Journal of Heritage in the Digital Era*, 4(2):221–233, 2015.

[29] Konstantina Kilteni, Raphaela Groten, and Mel Slater. The sense of embodiment in virtual reality. *Presence Teleoperators & amp Virtual Environments*, 21, 11 2012.

[30] Eric Klopfer, Susan Yoon, and Tricia Um. Teaching complex dynamic systems to young students with starlogo. *Journal of Computers in Mathematics and Science Teaching*, 24(2):157–178, April 2005.

[31] Konstantinos Ilias Kotis. Artist - a real-time low-effort multi-entity interaction system for creating reusable and optimized mr experiences. *Research Ideas and Outcomes*, 5:e36464, 2019.

[32] Vincent Lanquepin, Domitile Lourdeaux, Camille Barot, Kevin Carpentier, Margot Lhommet, and Kahina Amokrane. Humans: a human models based artificial environments software platform. 03 2013.

[33] Mary Margaret Lusk and Robert K. Atkinson. Animated pedagogical agents: does their degree of embodiment impact learning from static or animated worked examples? *Applied Cognitive psychology*, 21:747–764, 09 2007.

[34] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.

[35] MARUI. MARUI 3 plugin for autodesk maya, 2018.

[36] Teresa Monahan, Gavin McArdle, and Michela Bertolotto. Virtual reality for collaborative e-learning. *Computers & Education*, 50(4):1339 – 1353, 2008.

[37] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[38] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[39] Xueni Pan and Antonia Hamilton. Why and how to use virtual reality to study human social interaction: The challenges of exploring a new research landscape. *British Journal of Psychology*, 109, 03 2018.

[40] Margarita Papaefthymiou, Marios Evangelos Kanakis, Efstratios Geronikolakis, Argyrios Nochos, Paul Zikas, and George Papagiannakis. Rapid reconstruction and simulation of real characters in mixed reality environments. In *ITN-DCH*, 2017.

[41] George Papagiannakis, Nick Lydatakis, Steve Kateros, Stelios Georgiou, and Paul Zikas. Transforming medical education and training with VR using M.A.G.E.S. In *SIGGRAPH Asia 2018 Posters*, SA '18, pages 83:1–83:2, New York, NY, USA, 2018. ACM.

[42] George Papagiannakis, Panos Trahanias, Eustathios Kenanidis, and Eleftherios Tsiridis. Psychomotor surgical training in virtual reality. *Master Case Series & Techniques: Adult Hip*, pages 827–830, 07 2017.

[43] E. Pasternak, R. Fenichel, and A. N. Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 21–24, Oct 2017.

[44] Zikas Paul, Papaefthymiou Margarita, Mpaxlitzanakis Vasilis, and Papagiannakis George. Life-sized group and crowd simulation in mobile AR. In *Proceedings of the 29th International Conference on Computer Animation and Social Agents*, CASA '16, pages 79–82, New York, NY, USA, 2016. ACM.

[45] Nadine Pfeiffer-Leßmann and Thies Pfeiffer. Exprotovar: A lightweight tool for experience-focused prototyping of augmented reality applications using virtual reality. In Constantine Stephanidis, editor, *HCI International 2018 – Posters' Extended Abstracts*, pages 311–318, Cham, 2018. Springer International Publishing.

[46] Joachim Rix, Stefan Haas, and José Teixeira. *Virtual Prototyping: Virtual environments and the product design process*. 01 1995.

[47] Ricarose Roque. Openblocks : an extendable framework for graphical block programming systems. 05 2008.

[48] Jeffrey Rubin and Dana Chisnell. *Handbook of Usability TestingXXX: Howto Plan, Design, and Conduct Effective Tests*. Wiley Publishing, 2 edition, 2008.

[49] Maria V. Sanchez-Vives and Mel Slater. From presence to consciousness through virtual reality. *Nature Reviews Neuroscience*, 6(4):332–339, 2005.

[50] Jeff Sauro and James R. Lewis. Chapter 2 - quantifying user research. In Jeff Sauro and James R. Lewis, editors, *Quantifying the User Experience*, pages 9 – 18. Morgan Kaufmann, Boston, 2012.

[51] Ajith Sowndararajan, Rongrong Wang, and Doug A. Bowman. Quantifying the benefits of immersion for procedural training. In *Proceedings of the 2008 Workshop on Immersive Projection Technologies/Emerging Display Technologiges*, IPT/EDT '08, pages 2:1–2:4, New York, NY, USA, 2008. ACM.

[52] Digital Atom Srl. EyecadVR virtual reality architecture software, 2018.

[53] Evropi Stefanidi, Dimitrios Arampatzis, Asterios Leonidis, and George Papagiannakis. *BricklAyeR: A Platform for Building Rules for AmI Environments in AR*, pages 417–423. 06 2019.

[54] Sergi Villagrasa, David Fonseca, and Jaume Durán. Teaching case: Applying gamification techniques and virtual reality for learning building engineering 3D arts. In *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM '14, pages 171–177, New York, NY, USA, 2014. ACM.

[55] P Zikas, V. Bachlitzanakis, M. Papaefthymiou, S. Kateros, S. Georgiou, N. Lydatakis, and G. Papagiannakis. Mixed reality serious games for smart education. In *European Conference on Games Based Learning 2016*. ECGBL'16, 2016.

[56] Paul Zikas, Vasileios Bachlitzanakis, Margarita Papaefthymiou, and George Papagiannakis. A mobile, AR inside-out positional tracking algorithm, (MARIOPOT), suitable for modern, affordable cardboard-style VR HMDs. In *EuroMed*, 2016.

Appendices

Appendix A

Action Prototypes

```
namespace ModularScenegraph.Scenegraph
{
    /// <summary>
    /// This Interface needs to be implemented for every
       Action
    /// Describes the functionalities the Actions should have
    /// </summary>
    public interface IAction
    {

        int ParallelActionID;

        /// <summary>
        /// Go to Next Action
        /// Completes the current Action by finalizing and
           cleaning
        /// Destroys prefabs, holograms
        /// Also plays animations to set the next one
        /// </summary>
        void Perform();

        /// <summary>
        /// Go to Previous Action
        /// Resets current Action by finalizing and cleaning
        /// Destroys prefabs, holograms
        /// Plays Undo animations
        /// </summary>
        void Undo();

        /// <summary>
        /// Clears Action, Destroys everything
        /// </summary>
        void Clear();
```

```
        /// <summary>
        /// Initialize current Action by spawning the
           necessary prefabs
        /// Sets each Action properties to run correctly
        /// </summary>
        void Initialize();
    }

}
```

Listing A.1: IAction Interface

```
namespace ModularScenegraph.Prototypes
{
    public class InsertAction : BaseAction
    {
        public override void Initialize()
        {
            base.Initialize();

            // Spawn all prefabs
            foreach (ActionComponents component in
               actionComponentsList)
            {
                SpawnInsertPrefab(component);
            }
        }
        //Sets Interactable and Final Prefabs
        public void SetInsertPrefab(string intPrefabPath,
           string finPrefabPath,
                                    string parent = null)
        {
            if (performed)
            {
                Clear();
            }
            ActionComponents action = new
               ActionComponents(intPrefabPath,
                                           finPrefabPath,
                                              parent);
            actionComponentsList.Add(action);
        }
        //...
    }
}
```

Listing A.2: Insert Action (part of script)

Appendix B

Visual Scripting

```
internal void GenerateXmlFromNodes(SaveOp saveOption)
{
    List<Lessons> storyboard = new List<Lessons>();
    List<Node> nodeList = _window.graph.nodes;
    List<Node> alternativeNodeList = new List<Node>();
    foreach (Node node in nodeList)
    {
        //Find Operation Node
        Node operationNode = Node.GetOperationNode();
        //Link Operation Node
        UpdateNodeLists(operationNode);
        storyboard = StoryboardCompressor(operationNode);

        //Export Alternative Nodes
        foreach (Node altParentNode in alternativeNodeList)
        {
            Node updatedNode =
               UpdateNodeLists(altParentNode);
            storyboard.AddRange(StoryboardCompressor(updatedNode));
        }

        if (saveOption == SaveOp.SaveAs)
        {
            importedPath = SaveFileAs("xml",
               "Assets/Resources/Storyboard");
        }
        //Exports storyboard to xml file
        StoryBoardExporter exporter =
                new StoryBoardExporter(importedPath, ref
                   storyboard);

}
```

Listing B.1: Export visual nodes to xml file

```
private List<Lessons> StoryboardCompressor(Node parentNode)
{
    List<Lessons> storyboard = new List<Lessons>();
    int currentLessonID = 0;
    //For normal nodes
    if (parentNode.GetType().Equals(typeof(OperationNode)))
    {
        foreach (Node lessonNode in
           parentNode.GetChildLSANodeList())
        {
            Lessons lesson = new Lessons();
            //Traverse Stages
            foreach (StageNode stageNode in
               lessonNode.GetChildLSANodeList())
            {
                Stages stage = new Stages();
                //Traverse Actions
                foreach (ActionNode actionNode in
                   stageNode.GetChildLSANodeList())
                {
                    //Retrieves data from Action Node to
                       construct xml file
                    \textbf{stage =
                       ExportActionNodeXMl(stage,
                       actionNode);}
                }
                stage.stageName = ExportNodeName(stageNode);
                lesson.allStages.Add(stage);
            }

            lesson.lessonName = ExportNodeName(lessonNode);
            storyboard.Add(lesson);

            currentLessonID++;

        }
    }
}
```

Listing B.2: Compresses Storyboard and Retrieves data from Action Node

Appendix C

Reflection

```
public static Assembly Compile(string source)
{
    var provider = new Microsoft.CSharp.CSharpCodeProvider();
    var param = new CompilerParameters();

    // Add ALL of the assembly references
    foreach (var assembly in
        AppDomain.CurrentDomain.GetAssemblies())
    {
        param.ReferencedAssemblies.Add(assembly.Location);
    }

    // Generate a dll in memory
    param.GenerateExecutable = false;
    param.GenerateInMemory = true;

    // Compile the source
    var result = provider.CompileAssemblyFromSource(param,
        source);

    if (result.Errors.Count > 0)
    {
        foreach (CompilerError error in result.Errors)
        {
            msg.AppendFormat("Error",error.ErrorNumber,
                error.ErrorText);
        }throw new Exception(msg.ToString());
    }

    // Return the assembly
    return result.CompiledAssembly;
}
```

Listing C.1: Runtime script compilation

```
public Node ReflectScript(Node parentNode, System.Object
    actionScript, int parallelID)
{
    Assembly assembly;
    script = actionScript.ToString();
    //Keep record of assemblies to reduce compiling time!
    if (assemblyRecord.ContainsKey(script))
    {
        assembly = assemblyRecord[script];
    }
    else
    {
        assembly = Compile(script);
        assemblyRecord.Add(script, assembly);
    }

    Node returnNode = null;
    //Traverses types within script and tries to invoke the
       method
    //ExportScriptNode
    foreach (Type type in assembly.GetTypes())
    {
        object instance = Activator.CreateInstance(type);
        className = type;
        actionType = type.BaseType;

        //Constructs arguments to pass along with the
           invocation
        object[] parametersArray = new object[] {parentNode,
           actionScript, parallelID };
        //Finally invokes the method ExportScriptNode
        returnNode =
           (Node)actionType.InvokeMember("ExportScriptNode",
           BindingFlags.InvokeMethod |BindingFlags.Instance
           |BindingFlags.Public, null, instance,
           parametersArray);
    }
    return returnNode;
}
```

Listing C.2: Reflect script for data retrieval