Flow and automata programming inspired approach to human-readable VR training content logic modeling: View-Unit-Task-Step Unified Training Structure

Frans Simpura

Helsinki December 1, 2019

Master's thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author
Frans Simpura

Työn nimi — Arbetets titel — Title
Flow and automata programming inspired approach to human-readable VR training content logic modeling: View-Unit-Task-Step Unified Training Structure

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	December 1, 2019	87 pages

Tiivistelmä — Referat — Abstract

This thesis introduces, demonstrates, and evaluates a custom VR training content logic modeling approach, the VUTS method we have created. We inspect the content creation needs from the point of view of the occupational safety training focused VR platform, Virtuario^{TM}, developed at The Finnish Institute of Occupational Health.

We review flow-based programming and Statecharts as comparison points to our approach and analyze techniques of secondary notation for their suitability for our needs. To define and evaluate our created hierarchical, visually representable flow approach, we use methods of design science to first define what we expect from an artifact that enables scalable, modular, and visualizable VR training content logic modeling: We test the artifact against requirements dictated by the pedagogical substance, software architecture development, and sustainable VR training content creation.

We define the requirements for the artifact and test our developed approach against them by constructing real-life training scenarios utilizing the approach. We evaluate how this approach fares as the artifact to satisfy our set requirements. This thesis shows how our approach satisfies all the set requirements, is implementable within Unity3D game development platform, and is suitable for the content creation needs of Virtuario^{TM}.

ACM Computing Classification System (CCS): **Data flow languages**, **State systems**, **Virtual worlds training simulations**

Avainsanat — Nyckelord — Keywords
design science, flow-based programming, state machines, virtual reality

Säilytyspaikka — Förvaringsställe — Where deposited
—

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Virtual reality technologies and virtual reality applications have seen increasing popularity within the last few years. The solutions have become more affordable and the overall supply has increased in terms of different devices and volume sold. This has made virtual reality-based training more viable option among various institutions seeking new solutions for training their workforce. A certain level of entertainment is expected even when educational content is in question: Finding sustainable ways for developing such content can be a challenge as aspects of both gamification and substance driven goals set their own requirements. The end-product must be pedagogically accurate to meet given learning goals while simultaneously offering enough appeal to carry its player through the material for them to enjoy coming back for more.

A game development platform, such as Unity3D, can be used to develop virtual reality ready, customizable content that meets the needs of a corporate-grade virtual reality training application. The programming can utilize an imperative approach of writing the lines of codes in a code editor of choice, and then judging the reusability of the created code, but we can seek out other ways of modeling content, such as virtual reality training scenarios. There are ways to make the content creation follow a clear structure without the need of writing new lines of code for implementing a vast library of new content – once the tools for content creation are available and implemented.

When building this kind of content, a visual representation can provide additional benefits over a textual format, such as providing tools for easily inspecting different abstraction levels. With a suitable approach, we can convert a textual content logic decription into a visual model and revert it back to text.

The Finnish Institute of Occupational Health has developed a domain-specific approach, the VUTS method, to model virtual reality training content for the needs of occupational safety training without the need of writing any lines of codes to successfully carry out content creation. This thesis introduces, demonstrates, and evaluates this approach that combines different techniques similar to other known approaches by sharing some of its features with flow-based programming and state machines. The approach is being used for developing all the content The Finnish Institute of Occupational Health offers to their clients making large-scale content creation viable even for small teams.

Objectives

This thesis is the *post-review* of the *already designed and implemented* approach and aims to show how this VUTS method fares as a domain-specific approach. We will utilize methods of design science for defining the requirements for our approach and evaluate it based on them. We aim to compare preexisting solutions to our approach and demonstrate how our approach features similar techniques that are also present in well-know models. Our approach aims to offer something new and beneficial for various parties having virtual reality or other loosely story-based content creation or research needs similar to ours. The scope of this thesis will not present a formal definition for our approach and will have a pragmatic take on demonstrating the capabilities of our approach for our needs we will define in detail.

This thesis aims to show how it is possible to utilize said approach for efficient modeling of large-scale corporate-grade virtual reality training content. As for our pragmatic *development objective*, we also want to integrate the approach into Unity3D development environment to enable and sustain our virtual reality training content creation. We present how this approach aims to tackle the potential issues by utilizing a visual, modular and customizable way of modeling VR content logic. Finally, we evaluate it by first demonstrating its suitability for constructing content utilizing the approach and then by reflecting how the set requirements compare to the solution offered. We look answers to the research questions of

RQ1: What benefits can a domain-specific approach offer over using general flow and state-based approaches?

RQ2: What benefits can we achieve by depicting VR content logic as a hierarchical, human-readable, and visual flow?

We ultimately aim to propose an approach general enough for constructing virtual reality training logic in a simple to understand, yet expressively satisfying form. We also propose a way of visualizing this kind of content logic so that the need for textual cues can be reduced compared to a textual description and replace them with visual elements that can aid depicting different abstraction levels of the same content. Our focus is naturally inclined towards finding a solution that primarily caters the needs of The Finnish Institute of Occupational Health but we believe our approach in its general form can also benefit other parties in their research and development.

About this thesis

At the time of writing this thesis, the author works as the software architect and the technical lead of a virtual reality training platform project development of The Finnish Institute of Occupational Health and was responsible for developing the technical implementation of its preceding virtual reality research project as well. The virtual reality training platform specific implementation of the approach in its entirety is intellectual property of The Finnish Institute of Occupational Health and specifically created by the author for its needs.

Thesis structure

In this thesis, we first cover the basics regarding virtual reality and the virtual reality training platform developed by The Finnish Institute of Occupational Health and also define the necessary vocabulary. In Chapter 3 we introduce the research methodology and define the requirements for our approach to meet. Base literature is reviewed and analyzed as a comparison point to compare our approach to preexisting approaches in Chapter 4 before we introduce our approach and explain its inner workings in Chapter 5. The approach is tested by using it to model real-life training scenarios in Chapter 6 and the results are analyzed in Chapter 7. Finally, in Chapter 8 we discuss the results, the possible needs for extending our approach, and its potential to be further developed.

2 Virtual reality and Virtuario™ training platform

In this chapter we will briefly cover what virtual reality is and how it is utilized in the virtual reality training platform of The Finnish Institute of Occupational Health. Some of the general terms are also defined at the end of this chapter.

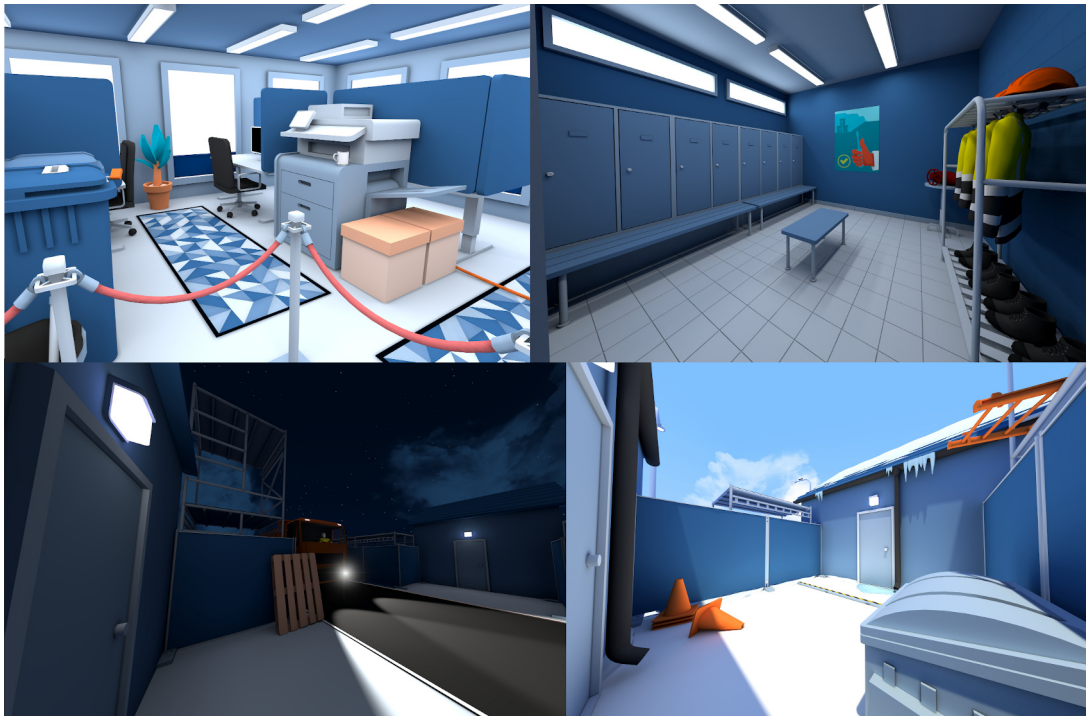

Figure 2.1: A compilation screenshot of various 3D modeled environments used in virtual reality training platform Virtuario™ for teaching occupational safety themes.

2.1 Virtual reality as a platform

Virtual reality (*VR* in short) and its use in education and training applications have seen increasing use within the last few years in different fields (Van Wyk and De Villiers, 2009; Dockx et al., 2016; OMER et al., 2018; Sacks et al., 2013; Brown and Green, 2016; Parsons and Cobb, 2011). As the technology has progressed, the VR solutions have gotten more and more approachable by the consumers and enterprise customers alike (Barnes, 2016).

In many VR solutions the fundamental idea is the same: the user wears a headset and holds one or two controllers or other sensors and the headset itself or external tracking stations track the position of both the headset and the controllers. This

information is used to synchronize the stereoscopic 3D image by the lens-like screens of the headset to match the position of the headset and the controllers within the space (Mazuryk and Gervautz, 1996; Cipresso et al., 2018; Silva et al., 2003). This allows the player to walk or otherwise move in the physical space and experience the synchronized movement in the virtual environment. As their position is tracked, the sensors can be made visible to the player, such as showing the held controllers as 3D models, in the virtual world through the headset in addition to matching the player's orientation in the physical world within the rendered virtual environment. Different VR headsets, depending on their type, can require connecting them to a computer for handling the actual processing, such HTC Vive Pro (HTC, 2019), or be stand-alone, such as Oculus Quest (Oculus, 2019), and contain all the necessary processing units and an operating system required to run VR applications without external processing.

The aim of VR solutions can be to create different environments (such as seen in Figure 2.1) for the player to "be part of" aiming to simulate believable – even life-like – experiences. These experiences can combine both the aspects of educational goals and gamification, making the experience game-like through challenges, goals and achievements (Huotari and Hamari, 2012). *Presence*, how fully the player feels being in the virtual reality (Barfield and Furness, 1995), can be used as a factor to track successfulness of created VR applications. To achieve presence by the user we are dependent on the *immersion* created by the technological capabilities of the hardware used (Mestre et al., 2006). If an educational VR application is question, internalizing the pedagogical goals is evidently another factor to consider on top of the generated and perceived believability, the lifelikeness of the virtual environment.

2.2 The VR research at The Finnish Institute of Occupational Health

In 2018 The Finnish Institute of Occupational Health launched the MoSaC research project for studying how virtual reality-based occupational safety training would compare to lecture-based training (Nykänen et al., 2019) focusing on construction site safety themed content. The content included working at heights, working with machinery, learning about the correct safety equipment for a given job as well as some general safety themes regarding visibility and observing the environment for potential safety hazards. For making possible to depict such content in VR, we were required to design and implement a virtual reality training application. We

rationalized the best way to achieve pedagogical accuracy was to build it utilizing only the in-house know-how even if it required building a technical implementation from scratch.

The environments and the training logic were to be built around scripts provided by the safety specialists. The scripts were written in co-operation with affiliate businesses that could benefit from participating in the MoSaC study and involved themes relevant to their area of expertise. The scripts were provided to the *development team*, the members of the project in charge of the technical implementation external to the design process, to be turned into interactive VR training programs. These scripts were presented in a textual, non-technical, human-readable format with a story-like structure starting from a clear starting point and leading to the end. The scripts had different locations to cover and a story-line like progression with player-centric choices and limited number of outcomes depending on those choices. While all the written scripts were seemingly similar in their flow-centric representation, they were overly ambiguous in terms of their rate of progression, sequentiality, and branching.

The idea of fitting the script format into a generalized, visually modelable approach became apparent to ease the work of both the designers and the implementors. We also wanted to minimize the interpretation errors being made when reading the script independent of the person reading it. To further test this idea against practice, the writer of the scripts was asked to provide a visual flow chart representation of their choosing based of the scripts for the development team to analyze their suitability for a generic approach.

All the scripts in this freehand flowchart representation ultimately had a forward-going format with minor side-tracks of the story that then would lead back to the main track, the intended "correct path" of the story. They were, as-is, drawable on a whiteboard as clear flowcharts but had too many events for a "boxes and arrows" approach to draw out every point within the story as its own state and pointing out the transitions between each state. This kind of "state explosion" made us think of a better, more compact way of representing the script without sacrificing any information so that the converted script could be used as a base for the implementation of the wanted VR training. We were interested in finding a solution that would minimize the need of drawing an excessive amount of arrows or other connection between perceived events and also make the direction of progression easier to read.

Another apparent issue was the different transitions between states being ambiguous and the idea of "when" an event or the transition should occur was not clearly expressed within the chart. On the other hand, some of the transitions were more implicit in their nature and could always be turned into similar "chains of action" that wouldn't require pointing out every transitional arrow between them. Having a "mindmap" representation (like in Figure 2.2) of the script structure wouldn't provide enough information about the flow while being needlessly "airy" in its format.

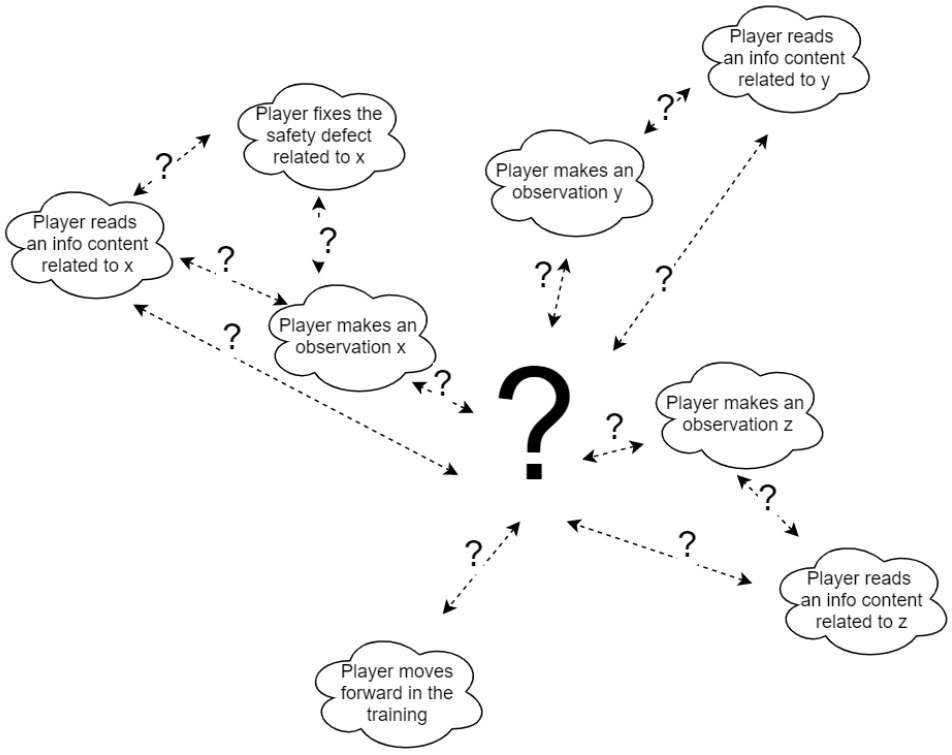

Figure 2.2: A script could be depicted in a "mindmap" format, but deducting the logical transitions would be troublesome and leave room for ambiguity. Each "association line" raises questions about the logic of the "flow" for the player to follow.

We came up with a way of placing similar sequential events together, minimizing the need of explicitly having to depict "redundant" implicit transitions. By modeling a structure that groups similar behavior together in a compact manner within higher level groups, we were able to depict the script in a hierarchical, flowchart-like format omitting the visual arrow-relying representation of the lower level transitions. One core idea was to be able to depict the resulting flow by drawing it using a marker on a flipchart or a whiteboard and so that the we could simplify lower-level details when necessary to present modeled logic to non-technical members of the team.

This solution, a high-level flow modeling approach – inspired by a whiteboard design style editing and the format of human-written story scripts – is what we call *VUTS*. The VUTS method aims to make high-level "non-tech" design of the contents more feasible and to encourage reusability of the existing functionality and ultimately streamline the agile design periods to reach a point where previously programming time constrained content creation will no longer be an issue. We require a solution general enough that provides a way to easily model and implement new training content the imminent future needs require. The goal of our approach is not only to present a standardized way of creating content, but also to guide us developing the content in a sustainable way by not making the logic needlessly complex for a simpler workflow.

MoSaC can be considered our starting point for our virtual reality content production that initially required creating four different training programs by the needs of the research. As the on-going research displayed encouraging results, we saw greater potential in developing similar training content in greater scale. The know-how gathered when undertaking the development phase of the MoSaC research could easily be translated into building a dedicated product for teaching and training occupational safety and improving safety culture using VR. Soon after we started developing a virtual reality training platform for offering similar content for the enterprise market.

2.3 Virtuario™ training environment

As the technical development phase of MoSaC came to a closure, we saw optimism towards utilizing the know-how for larger scale VR content production and using the created approach to enable competitive content creation rate. For this purpose, we started developing a VR safety training platform for large-scale content distribution and launched it as a product, *Virtuario™*.

Virtuario™ is a virtual reality training environment[1] by The Finnish Institute of Occupational Health that consists of various occupational safety related training content modules. Specializing in making occupational safety related training widely available, Virtuario™ provides a library of occupational safety training content for the user to choose from. At the core of each and every training flow, we utilize the VUTS method for designing and modeling them.

[1] www.ttl.fi/Virtuario, the name 'Virtuario' is a play of words of the words 'virtual' and a Finnish suffix 'io' which can be used to denote a space.

Each training scenario has the same fundamental idea: The player uses the virtual reality system in a physical space with enough moving room. The different, changing virtual sceneries we create each utilize four-by-four-meter areas that – when matching with similar dimenions of the physical space around the player – allows the player to freely move around and experience the virtual world as if the player was part of it instead of the physical space surrounding them. The player is given occupational safety related tasks and goals in a form of a flowing "story" that guides the player through a scripted flow of events. The flow guides the player from the start to the finish for the player to learn about a given occupational safety related theme or to perform a more specific task, such as operating a machinery safely and correctly. The interactionality, how the player acts with the environment, is simplified to rely on pointing and clicking behaviour using two handheld controllers that are the only peripherals the player requires in addition to the headset they use to view the virtual world. The player can always move and look around freely and set their own pace to move forward within the story.

Every content module, consisting of separate smaller training scenarios, shares vastly similar features, such as having visually simplified environments, incorporating objects to be interacted with, and displaying informational floating text windows with spoken audio once the objects are activated, as seen in Figure 2.3.

Commonly the flow of a Virtuario™ training is follows a pattern: The player starts standing on a specific position related to the VR space and the environment is loaded around them in a "fade in" manner[2]. A motivational text box is displayed and read out loud to give the player a brief background story. Some kind of task is usually presented right after reading the first text, such as finding an object, moving to a specific location within the space or – in the worst case somewhat unavoidably – reading through more text boxes. The player is usually being sequentially presented more tasks that fit the story and the substance of the training: in our case tightly tied around the themes of occupational safety. The tasks given can be as simple as locating a correct object within the environment (as in finding and pointing it with the controller) or more gradual, such as finding all the suitable objects listed explicitly or by the given target count, or completing a given work order appropriately by following the correct order of its sub-tasks. After the player has

[2]As a design principle, we avoid anything instant happening in the virtual world because of the novelty of VR. Being in virtual reality is exhilarating itself and sudden chances can – in our experience – confuse or even scaring the players. For every action, somekind of an "easing function" is therefore applied.

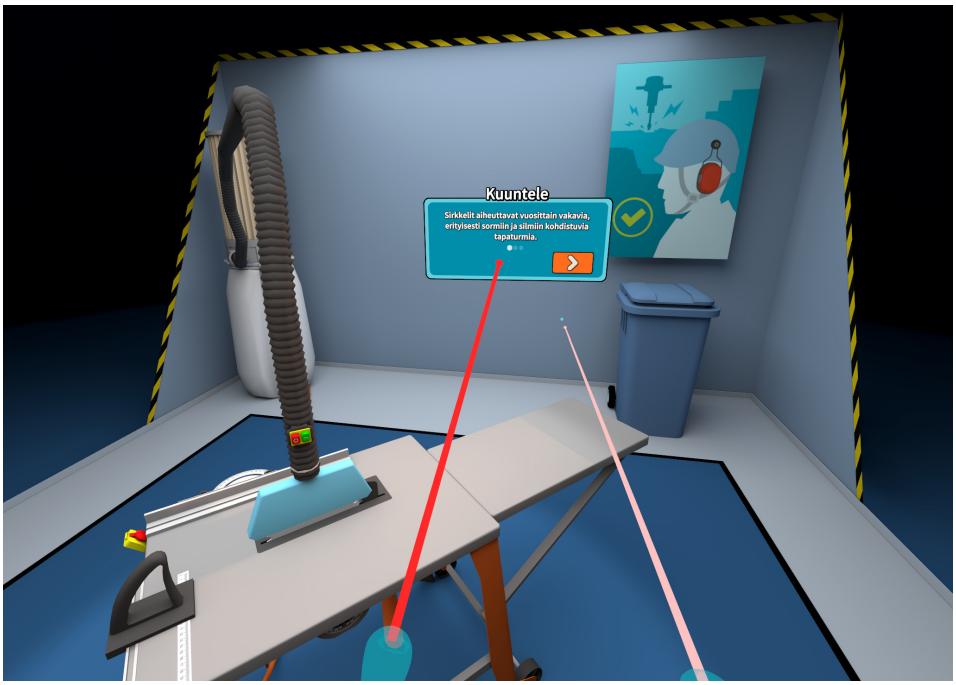

Figure 2.3: A screenshot of one of the Virtuario™ environments for teaching table saw safety visible from the player's viewpoint and one of the two controllers pointing at a floating info window. We aim to keep the environment minimalistic or simplified to draw focus on the pedagogical content while still making the scene appealing.

made a choice, a form of an audiovisual cue is typically presented followed by a congratulary text that states the actions the player just did and verbalizes the rationale for that action. If there is a chance for picking a "wrong" answer, such as trying to activate an object before carrying out necessary safety precautions, the player is informed about the correct procedures and asked to try again.

The player can never "fail" or get stuck within the training. Each wrong answer only opens up a different route that leads back to the main track. One of our pedagogical goals is to enforce positive learning by creating a clear link between the positive actions and their encouraging reactions (Bandura and Walters, 1977) and promoting this by making the player complete a path through the correct actions. This idea, at a high level, can be drawn as forward-going flow, as seen in Figure 2.4.

Once the required tasks are done, the player is told about the next part of the training program and once acknowledging that – usually by clicking a button within a floating text box – the environment around the player fades out and back in again with a new scenery and the whole process continues until all the parts of the program have been completed.

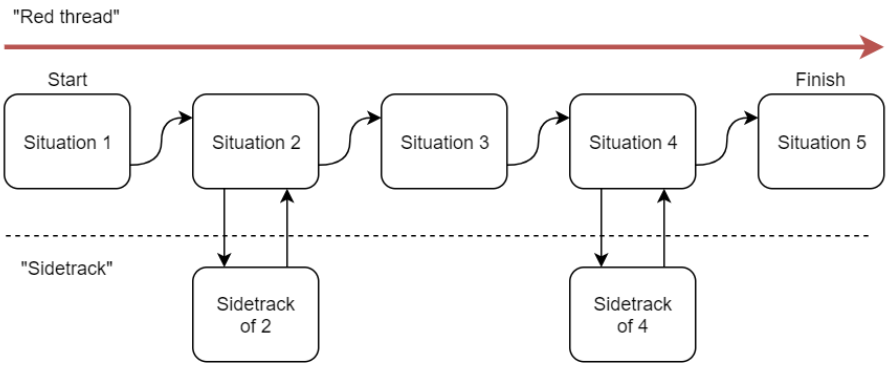

Figure 2.4: A story that goes forward as a "red thread" but lets the player to make choices that allow them to try again in order to move forward within the story flow.

Each training content is fundamentally linear: The player is given a task that has to be completed with restricted freedom on choosing the "path" in which to progress. This is heavily due to the nature of safety trainings, as each procedure to be trained is precisely defined and – challengingly enough – have little to no room for being creative when following them. To make the training more interesting, we allow the player to find the correct ways of doing things. We enforce that all the necessary actions are completed but the player may have power over choosing their *order* when it does not have any implications on the correctness of given task or internalizing the pedagogically relevant goals. The target is visible in the environment, but the player is not forced to focus on it or to be otherwise guided. It is part of an illusion of a choice: Finding something is usually due to the object being the only interactive object within the environment at that time. The player can still perceive this as having multiple "parallel" paths to choose from, but in reality there is a clear "red thread of action" that they follow at each given moment. This ensures the player can't get lost in the flow – as long as the objectives are made clear. The learning can benefit from engaging the player in the story and by offering challenges; relying on immersion alone might not be enough to support learning (Hamari et al., 2016).

For interacting with and within the environment, the player uses one of the two handheld controllers and points and clicks the objects with a virtual laser pointer. Virtuario™ also tracks the spatial information of the headset to determine where in the environment the player is any given time to for example warn the player of being too close to a hazardous substance or heavy machinery. Even though virtual reality supports a multitude of different methods of object manipulation (Bowman and Hodges, 1997; Kwon, 2019) such as grabbing, throwing, and turning, we ultimately waived of all the object manipulation methods other than the pointing and clicking

and location-based interaction, such as walking into an area[3]. The pedagogical core of Virtuario™ is not about teaching to mimic the real life physical interactions, but to notice and identify common safety deficits and learn about them. The pointing and clicking is what we found to be the most intuitive, as it allows to interact with both close up and far away targets, and to be the most easy to comprehend by the players, as guessing the trajectory to perform a more specific movement is eliminated. The pointing and clicking functionality can be seen in Figure 2.5.

Figure 2.5: A player points at an object in the environment using a virtual laser beam of a controller and the object is highlighted until clicked at.

The original hardware used for running the product was the Vive Pro (HTC, 2019) on Microsoft Windows, but was later replaced with a standalone Android-based headset Oculus Quest (Oculus, 2019) to sustain the growing demand and ease the use by a more mobile approach. Considering the limited computational capabilities of the standalone VR headsets compared to the processing power of dedicated hardware in modern desktop computers (Cuervo et al., 2018), Virtuario™ specializes in highlighting the role of the player within an environment as a proactive safety inspector instead of focusing on dynamic objects and leaving the player a role of a reactive bystander trying navigate within the audio-visually dominating dynamic

[3]For some special cases the content includes simple gestures, such as looking at or raising an arm, as these kind of activities are often required for safe construction site movement, for example. When this kind of actions are expected, by our records, the player has to be *clearly instructed* and encouraged to perform these kind of more "physical" actions.

environment. The player is given time to think and reflect on their actions: There is no pressing time limit to drive the player to make hasty decicions. Given the yet novel nature of VR, Virtuario™ aims to offer an easy-to-comprehend, yet fun and pedagogically competent platform by utilizing the tools VR provides in moderation.[4]

2.4 Core concepts and definitions

Virtuario™ consists of various interactive training materials, *content* in general. Different pieces of content, when put together in a meaningful way, form *(training) programs*[5] – be it training courses, demonstrational walkthroughs, or virtual reality promoting experiences, each containing their own environments and player completable goals within. For designing and describing what each content contains – in terms of events, objects and environment – we use the term *story script* or merely *script* for their non-technical description. The *player* is the end-user using the system and completing the programs. The player is guided through programs by a *flow* which is a visualizable, interconnected structure, containing a finite sequence of points which can be said to be active depending on where in the flow the player is located. The flow can be *completed* or *finished* by the player, since in our case a script has an end point. The player's decisions may result in *branching* of the flow – as formally defined (Knuth, 1997) – which in turn affects how the player is guided through the script.

For convenience, we use *content creation* as an umbrella term for designing a script and creating content based on the script, and the team working on this kind of work *content creators*. The people programming on an abstract framework to add concrete functionality to serve the goals of the design are called *implementors*, turning the expected scripted behavior work in VR, derived from the term of "implementing" an interface, like in C# programming language[6]. These two teams may or may not be the same people, depending on the nature of the project and the size of the teams.

---

[4] A noteworthy example to this is the notorious aspect of virtual movement as virtual reality sickness inducing (LaViola Jr, 2000), that being movement that is not synced to the player's actual physical movement, such as moving the player in the VR environment while the player is standing still. In Virtuario™ all the virtual movement is absent in order to reduce the VR sickness.

[5] The name 'program' in this context is not to be confused with a computer program per se. The program in the context of Virtuario™ loosely refers to a "training program" of sort, a compilation of shorter pieces of content forming a whole.

[6] "Interfaces (C# Programming Guide)": `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/`

3 Research methods and requirements

Our aim is to find a *sufficient*, *suitable*, and *working* solution that can facilitate our workload on both implementing and designing our required content and that can also be used in the future to sustain our potentially growing content creation needs. For *sufficiency*, we accept the potential flaws in the approach as long as it is *suitable* for solving our problem set and meeting our requirements. It has to prove itself as a *working* solution – by not merely constituting a theoretical curiosity – being implementable as stated by our research goal.

3.1 Design science

We must define what we expect from the solution. For this, we approach this problem from the design science perspective: The nature of the product and its development goals require a specialized solution that can take advantage of a multitude of proven solutions, combined into the most efficient form from the viewpoint of the project, as an aspirational goal, to be then constructed.

We are looking to implement, improve and evaluate *an artifact* (Vaishnavi and Kuechler, 2004) to satisfy the different categories of requirements. We hypothesize our approach provides a sufficient, suitable, and working solution: For this purpose, we inspect the VUTS method as our artifact for constructing visually depictable VR training content. The artifact has to be *implemented via programming* and *usable within a development platform* to *create visualizable VR training logic for the needs of The Finnish Institute of Occupational Health.* For defining such an artifact, we define a broader, detailed set of requirements. We want to know whether or not our VUTS method fares as such artifact that meets the given requirements.

The "problem solving" mentality of design science (Hevner et al., 2004) is what guides us in evaluating the artifact: We want the artifact to answer the problems and requirements Virtuario™ project made apparent concerning the sustainable content creation and requiring a visual representation to depict the content logic. We want to identify the problem areas and test the artifact against them to ultimately answer all of them in a satisfactory manner.

Considering the core high-level requirements of the ease of designing necessary VR logic and being able to inspect it in a tangible, graphical format, we look to find an answer to the research questions by presenting a method for depicting such flow.

Our goal is to evaluate the successfulness of the artifact by *manipulation* (Mettler et al., 2014), as we later present case examples of actual client projects and argue how such an approach has proved itself beneficial by satisfying our requirements.

We are tempted to conclude that our approach is a solution to our content creation needs but without defining the requirements and evaluating the VUTS method against them, we cannot yet judge its performance.

3.2 Requirements for the artifact

For creating interactive content for the needs of Virtuario™, the artifact must satisfy the development criteria set by the development team, the stakeholders and the end-product that utilize the content created with it. Below, we introduce the categories and summarize their items. The requirement can be divided into three categories of

Substance The requirements of the end-product produced by the artifact to fulfill substance-driven standards set by The Finnish Institute of Occupational Health.

Architecture The requirements of the artifact to meet the architectural needs defined by the team in charge of implementing and maintaining the system.

Content creation The requirements of the artifact to meet the content creation-centric needs and to serve different stakeholders that inspect the derived product from their viewpoints.

Thus, the artifact we want to construct carries its values also into the derived product – the training content. Being a training tool of occupational safety and a product of a large scale software development project, Virtuario™ adheres to these substance, architecture and content creation centered standards for its pedagogical and technical implementation. Each program integrated into Virtuario™ content library must follow the same productional quality standards in both functional and non-functional requirements.

The mentioned needs are defined as the essential principles for Virtuario™ in both pedagogical and technical sense, dictated by the stakeholders of the project. These needs translate into the expected features of the artifact. We introduce each category and describe their requirements.

3.2.1 Substance

The pedagogical principles are mainly driven by the substance know-how of the Finnish Institute of Occupational Health and set the standard for the end-product. These standards define how the training content produced by the artifact behaves.

S_1**: Learning reproducibility** Each player engaging the content by using the system must face the same reproducible experience. The intended pedagogy-driven core material incorporated within the flow should be taught in its entirety. The pedagogical equivalence between each run as well as between different players has been a crucial part of the system from the beginning. If the training flow is to be incorporated with a random factor, it is to be well-defined and controlled so that the pedagical quality will not degrade.

S_2**: Player-driven flow progression** The material is to be presented in a controlled manner, in such way that the player remains in charge of the pace of the flow by having the direct control over the timing and order of the events happening within. The player is ultimately the one in control of the experience and should hold that power through the flow to set its pace. The artifact must provide a way of depicting completely linear phases within the story that can aid the player to focus on a single line-of-action at a time.

S_3**: Training through a distinct path** The flow should branch according to the meaningful choices the player makes, yet ultimately result in the reproducibility by guiding the player "back on track" after a mid-flow choice was made. The artifact should support depicting "story branches" that will lead the player back where the branching occur to make sure the main storyline is ultimately followed.

S_4**: Observable parallellism** The player is presented a clear course of actions, but simultaneously the training system should be allowed to work "in the background" by, for example, tracking the player's movement within areas of interest while the player is performing another task. There is a need for such "controlled parallelity" that allows building for multi-action scenarios but helps ensuring that the outcome doesn't prove itself too tricky for the non-tech-savvy player to grasp.

S_5: **Interactionality** The player should be able to make meaningful choices during the flow. These choices should result in consequences that hold a clear causality in the eyes of the player by having an impact on the outcome of the flow.

S_6: **Minimal learning curve** Once the player is familiar with a set of core concepts, these concepts should apply for the rest of the experience (Linehan et al., 2014). Other concepts introduced later on should have a clear resemblance to the concepts taught earlier on.

S_7: **Story-drivenness** The format of the content produced by the artifact should follow a dramatic structure and form meaningful parts within the flow. There should be a clear pacing within the flow and the artifact should encourage creating such content.

S_8: **Analytics** Each meaningful choice performed by the player should leave a fingerprint within the system. The artifact should make easy for collecting such information and making the path of the flow clear.

3.2.2 Architecture

The architectural needs define the structure of the artifact at an implementation level. Apart from the obvious general software project needs such as correctness, we list the core requirements that are especially relevant to considering the needs of Virtuario^{TM} as a platform, intended to distinguish the artifact from other solutions that would axiomatically aim to fulfill such needs in all cases.

A_1: **Simplicity** To ensure correctness and ease of maintaining and content design, we want the artifact to remain simple in terms of technical complexity, being able to be visualized, designed on and documented. This means focusing the its architectural design to best serve our specific content creation needs without resorting to solutions that incorporate an excessive amount of needless functionality and expressive power that will not serve our content creation needs.

A_2: **Expressiveness** The conceptual breadth of the created content still calls for expressive power of the artifact to allow us to implement the required content

without resorting to external solutions for our content logic modeling. We also require the artifact to not pose restrictions on the content creation visions while still being high enough level to provide a development leverage over programming the content logic by hand.

A_3**: Modularity** The framework for building such content has to be built around creating subset functionality that can easily be interchanged as demand for content change arises. The system should be built around its separate components not being dependent on the changes to their sibling or parent components, and should therefore ease creating variants (Huang and Kusiak, 1998) of the content with ease.

A_4**: Reusability** Our ultimate goal is to create a design platform that would allow content creation without resorting to implementing such content at programming level: Implemented functionality for the use of the artifact should generate building blocks (Mili et al., 1995) that can then be further utilized in building following new content.

A_5**: Customizability** As part of the content change demands, the artifact requires a way of agile customization of the existing functionality be it at substance or functional level. The functionality inserted within the flow should have a minimal cost for making both trivial and non-trivial changes. Customizing the content should not require excessive manipulation of the data outside the means of the artifact.

A_6**: Integration** We want the artifact to support communicating with the our virtual reality functionality we require implemented on Unity3D using C#. Similarly, the artifact should itself be implementable by C# as an extension to our existing system, the rest of virtual reality training platform it must work in co-operation with.

A_7**: Scalability** We require the artifact to allow for manageable development and content creation in terms on time being spent on using the system. The artifact should therefore offer benefits over more traditional approaches when being built for our specific domain: The artifact should enable future-proof, en masse content creation.

3.2.3 Content creation

The design side of the principles define the optimal workflow for the development team of Virtuario^{TM} for streamlined content design and creation.

C_1**: Guidelining** Considering the complexity bounds set by the simplicity and expressiveness principles, we want the artifact to double as a passive mentoring aid that ensures certain design level values are being respected: The designed content should not be overly complex to follow or explain. In other words, we need the artifact to guide us during the design phase by keeping visible the implicit standards for common level of complexity shared between created contents of similar script length. The artifact should encourage creating simple to understand (yet effective) content and should make it obvious for the designer to note the designed flow becoming too complex for the player to follow.

C_2**: Visualizability of the structure** The content output created by the artifact should have a tangible form that can be reproduced as a visual presentation for the client, design team or product owners alike. The design of a flow should not depend on a written or a technical format and should have clear visual representational levels for different stakeholders to understand. The resulting flow should be drawable by hand on a flipchart and electronically on a digital medium.

C_3**: Cognitive ergonomics** We want to minimize the human error as part of the content creation process. The artifact should aim for ease of use not only for the end-user using the product, but also for the developer creating such content by utilizing clear visual cues and approaching the problem at hand in subproblems in form as minor as possible. The artifact should allow for catering different design roles for different members within a team. Different point-of-views of designers, such as being able to approach the design problems from top-down or bottom-up, should be considered.

C_4**: Bookkeeping** Lastly, the artifact should help us measure the time spent on different aspects of content creation by allowing setting clear milestones and making them visible for outer inspection.

4 Theoretical base

Based on our defined requirements, we are interested in a flow-like representation of the training logic. We also want to track the state of the player within that flow and have it offer a visual representation for the needs of a design team to visually "script it" and to define a clear outline of a training as a path to be completed by the player.

For this purpose our constructed artifact closely resembles a flow-based structure that combines the aspects of state machines and can be depicted in a concise, tangible, human-readable form. We set our theoretical comparison point around the pre-existing knowledge on *flow-based programming* to provide us insight over how the structural aspects of the flow is depicted in a dedicated paradigm. To be able to depict the state, our approach can be compared to *automata-based programming* by one of its applications, *Statecharts*. To test and further improve the visual readability of our approach, *secondary notation* can be used to evaluate the use of transitions notation and color-based syntax highlighting.

We aim to obtain knowledge on how these pre-defined theories as the *theoretical comparison point* make our approach fare and what we can learn to further develop our artifact. The selected approaches can not be justified as defining any kind of *extensive scope* for tackling problems similar to those set by our requirements, but can be seen to contain similar features while catering a wider scope expressiveness. These preexisting, generic solutions can provide insight over what kind of tools are available and what possible drawbacks their use in our case would present. Ideally, the theoretical base would as-is provide us the proven methods to satisfy our set requirements.

In this chapter, we first introduce the flow-based programming in Sub-chapter 4.1, the Statecharts in Sub-chapter 4.2, and secondary notation in Sub-chapter 4.3. Lastly, in Sub-chapter 4.4., we review what we can gather from these approaches compared to our approach.

4.1 Flow-based programming

As our approach has its roots in something that we intuitively would call a "flow", we set the *flow-based programming* as our theoretical base for testing our artifact. For this purpose, we review and analyse this paradigm as defined in the book "Flow-

based programming" (Morrison, 2010). Each definition and technique mentioned in this Sub-chapter refers to this book unless otherwise cited and the diagrams are adapted from the examples of the book.

Flow-based programming (FBP in short) is a programming paradigm that depicts the structure of a code as a directed graph. FBP defines two core subjects for depicting the structure: *components* and *connections*.

Components are defined as "black box" processes that define a piece of functionality that works on *input* data and provides a result as *output* data. Components encapsulate functionality within and allowing for various implementations to be built independent to the external structure as a whole.

Connections are used to connect components by their inputs and outputs to form the flow, determining which data is provided for each component. At the ends of each connection is a *port* that is either an input or an output, depending on the direction of the flow.

Each component can produce one or multiple *output* values that can then be supplemented to other components via connections. These supplemented values are considered *input* values of those components and can be similarly used to produce output values to pass forward. This configuration therefore forms a chain-like structure that carries the first input forward by manipulating it through the components it passes.

How the data is transferred forward by the connections is up to the implementor: the data, in a form of *information packets*, can be processed instantly – as it intuitively would in a chain of sequentially operating processes – or it can be made reactive, if the data is to be waited. Information packets can move forward and be processed independently, creating a concurrent system with separate processes in progress.

If the order of the components is changed and the inputs and outputs are configured accordingly, we get a new structure that can produce a different result, but doesn't require changing the inner workings of the components or the logic by the input-output handling is based on. This approach therefore allows for a high-level approach that does not require altering the inner workings of components, allowing for a certain type of modularity[7] within the structure, thus respecting "separation of concerns" by separating the different components into distinct modules

[7]The author of the book calls this "Legoland programming"

(Laplante, 2007) as would be beneficial according to our requirements. The actual implementation is written by the implementor of the components: the structure is decoupled from the functionality of the components.

The re-usability of the components can further be improved by having the components support optional parametrization. The parameters are supplied to the component by a dedicated input port as "initial information packets" that contain the pre-determined parameters. Another way of supplying the parameters is to create a component that generates them and outputs it like it was an information packet. As the parameters, or their generators, are chosen by the designer of the flow-based structure, they can further enhance the customizability to pre-written components waiving the need of writing new ones for slightly different functionality.

FBP supports a graphical visualization of its structure, as its components form the mentioned connections between each other in a way we can depict as a graph-like representation by abstracting the structure to said component-connector layouts. This doesn't mean other paradigms couldn't be similarly depicted as well, but the nature of FBP being centered around the high-level flow itself, not the inner working of its processes, makes it straight-forward to draw a flow presentation of the logic as there is no need to further reduce the structure. As we now know the structure containing components and their interconnections as inputs and outputs, we can depict the flow, as seen in in a simplified structure of Figure 4.1.

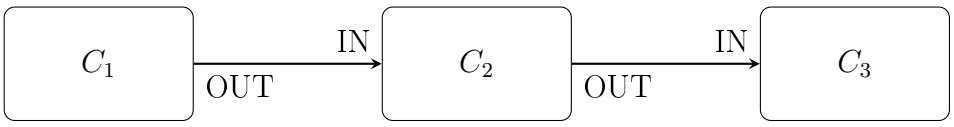

Figure 4.1: Components connnected by their input and output ports. The explicit direction of the flow can be seen.

If multiple outputs are defined by a component, branching can be utilized as each output can be lead to different component as their input. Each flow within a component is processed as a separate process, resulting in parallelism within the flow. This approach can satisfy our "observable parallelism" requirement (S_4) our artifact should meet. This structure is depicted in Figure 4.2.

Components can be placed within each other to form hierarchical structures. This "composite component" allows for using pre-determined template-like structures that contain lower-level components as their building blocks and the lowest-level components are provided a concrete implementation, as seen in Figure 4.3. Composite

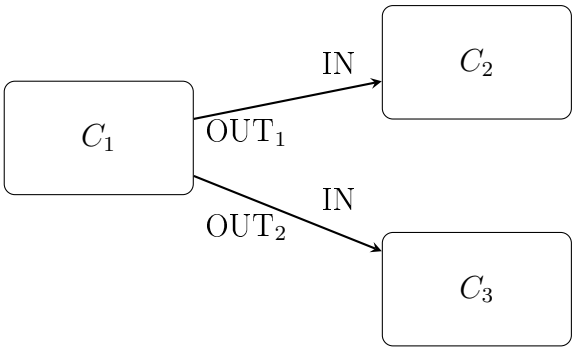

Figure 4.2: A branching occuring by having separate outputs connecting to different components by their inputs.

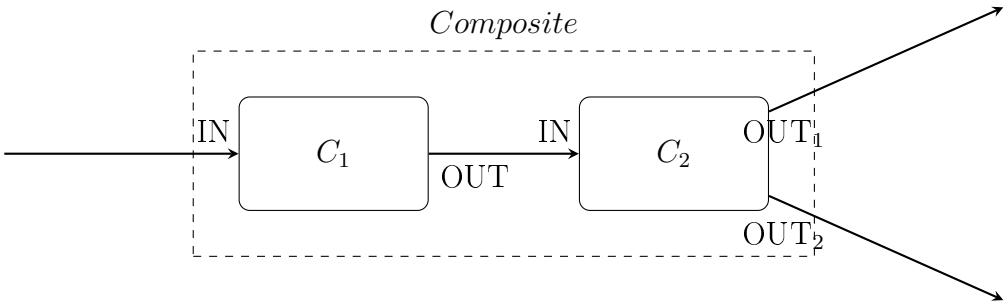

Figure 4.3: A composite component with the connection leading inside the composite component and the outputs reaching outside.

components are handled as other components, and therefore they require an input and a number of outputs: At a higher perspective, this makes a composite components look and function as if they were "normal" components, seen in Figure 4.4.

We are interested in finding an efficient solution to approach the growing demand for our content design needs, and arguments for saving development resources using FBP are in favor for its scalability: A growing size of an application is argued to result in more resources being spent on developing when using the "conventional" method of development compared to FBP *after* a certain point of application size is met. The initial investment in implementing and learning such a system would be a trade-off, but considering the future content creation and maintenance, this development investment can be argued to pay itself back in the long run. As we aim to provide an ever-increasing library of virtual reality training content and our application size is therefore plausible to be seen as ever-growing, the point would

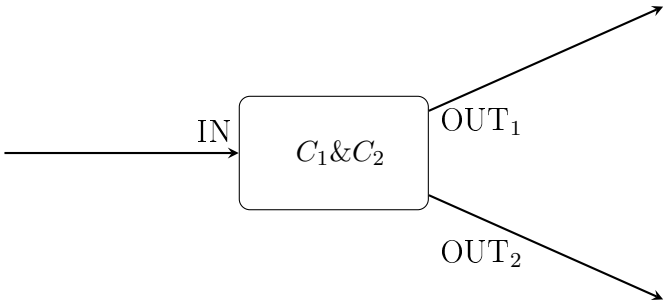

Figure 4.4: A composite component as seen from "outside": The inner workings are hidden and only the outermost input and two outputs are being exposed.

actualize through our development: Keeping the develop time manageable in the near and distant future is required, as required by our scalability requirement (A_7).

4.2 Statecharts

As our goal is also to keep track of the explicit state of a flow within a given script in addition to structuring otherwise a "free-form" flow, we can utilize a paradigm that determines a tangible state to implement such tracking. We are also interested in incorporating a sense of parallelism within our approach, so there has to be way of tracking multiple points at a time.

Automata theory present a way of depicting flows of various programs as state machine like structures with a set number of possible states to be in as "deterministic finite automata" (Hopcroft, 2008).

Statecharts introduce a way of hierarchically depicting an automata (Harel, 1987). State machines of Statecharts contain levels of abstraction depth. A structure that can be *zoomed* in or out in a sense of encapsulation allowing for inspecting the structure at higher or lower level without taking the implementation of each state into account: Due to the "state explosion" of all the required states where multiplying the existing states for creating new capabilities of depicting more complex logic would be needed, a "complex system" cannot be effectively presented in a simplified format as a "naive" single-level state diagram. A simple state diagram with two connections forming a "loop" can be seen in Figure 4.5.

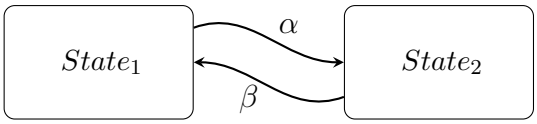

Figure 4.5: A simple, loop-forming Statechart graph. Both states have their own transition pointing to each other.

It is noteworthy – considering our set visualizability requirement (C_2) – how Statecharts offer a "visual formalism" by their nature, making the representation of complex state systems more convenient by attemping to evade the state explosion by allowing a multi-level state structure, and providing a self-guiding structure as a form of its zooming feature, allowing to inspect the chart from a higher or a lower viewpoint by encapsulating states within each other to form meaningful groups of "super-states". This approach is similar to the composite component method of FBP, as both of them allow building pre-determined building blocks from lower level blocks to be used as parts of their whole structure, as seen in Figure 4.6.

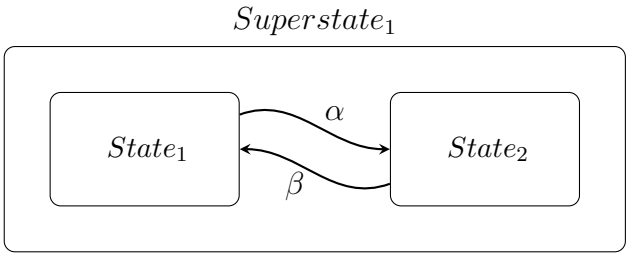

Figure 4.6: Substates within a superstate. When being in either of the substates, the superstate is also considered as "being in".

Somewhat similarly, as well, to the FBP, Statecharts' *states* (instead of components) are interconnected with *arrows* (instead of connections) guiding the "flow" within the chart structure. The arrows lead to between the state and incorporate *events* that lead the state the transition to another. The events can be accompanied by *conditions* that guard that the transition happens only if an additional requirement holds.

By transitioning between the state, the transition can trigger *actions* which are "one-shot" operations that affect the external state of the system, such as playing a sound or displaying an image. These actions allow the structure to not only reactively control the flow, but to also invoke events by the structural flow. In addition to actions, *activities* can also be used by defining functionality that is started when entering a state and terminated once leaving it, allowing for durational effects to

be carried out on demand, thus enabling continuous work to be carried out while staying inside a state.

As the super-states allow for the flow to stay within multiple states as once – in the superstate and one or multiple substates – while each of them potentially having an activity being in progress allows for functionality being run concurrently. These allow the structures to take advantage of simultaneous different effects: Meaningful combinations of a super-state and substates allow for building more complex and expressive structures. This particular idea enables concurrency at the hierarchical level, but limits it to nested structures.

Moving between nested groups of states allow for gradually building more and more complex structures for depicting clear separate "sections" of functinality. Combining various different depth state hierarchies into a single structure can result in a "jagged" structure like seen if Figure 4.7.

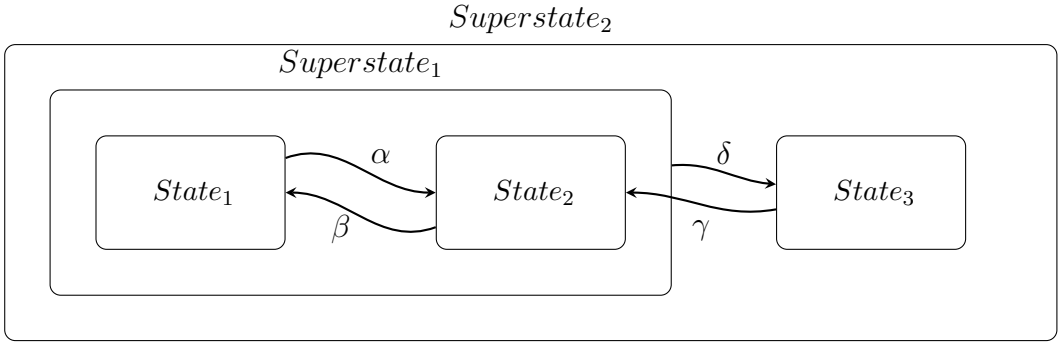

Figure 4.7: Adding more nested states and cross-referencing them allows for creating more hierarchical state clusters and and make different states share same common superstates.

Statecharts also support *orthogonality* that allows for creating combined states with multistate behavior within a single-level state. Entering a "box" like this makes the flow enter its designated starting states simultaneously and progress similarly simultaneously forward by conditions. The separate sides are divided by a dashed line. This can be seen in Figure 4.8.

We have seen how both FBP and Statecharts provide a visual presentation, each regarding their own use structure. We would benefit from finding a more general comparison point for the visual aspects of constructing flow-like structures in general.

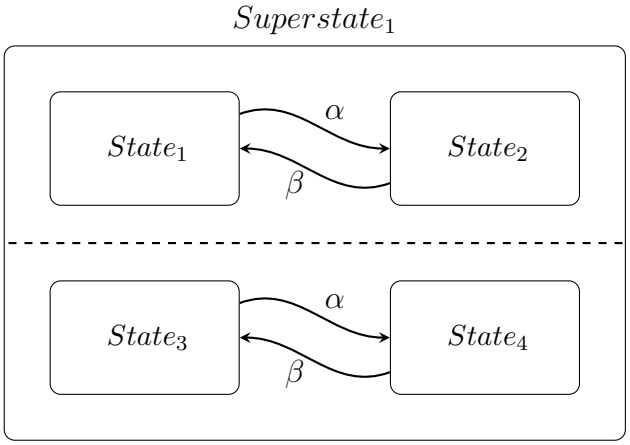

Figure 4.8: Orthogonality of Statecharts: The progression is simultaneously in two separate "sibling" states, progressing indepently of each other.

4.3 Secondary notation in depicting charts

As the artifact should (in its representational format) be highly visual in its nature, our goal is not only to make the the artifact technically sound but also ease the workload of designers and implementors when working with such an approach. Our one desired trait for the artifact is to support cognitive ergonomics: It should minimize the mental burden when using the artifact for content design, making reading it accurate and fast.

Considering that the artifact will be used by both technical and non-technical people, our goal is to minimize the constrains it might pose on the human readability optimistically disregarding readers' backgrounds. People interpreting graphs may have varying prior experience of having read or worked with different process models, and this can affects how well and correctly given graphs are interpreted (Mendling et al., 2007). One fundamental requirement of the artifact is to cater to different stakeholders for them to be able to communicate utilizing similar depictions of the charts: There is a need for a systematic, standardized way of depicting the resulting structure. In addition to this we should arguably try to stick with a single style of notation, or one should at least be ultimately adapted, to aid the understandability by minimizing the need of learning new notational cues (Becker et al., 2000).

We analyze this problem from the viewpoint of secondary notation that covers colors, shapes, spacing and other visual cues that affect the readability of different process models (Green and Petre, 1996; Schrepfer et al., 2009). Green and Petre (1996) define multiple dimensions a visual programming language can use to evaluate the

performance of it for visual design, one of them being secondary notation, which "conveys meaning to human reader". We can later use this criterion to evaluate our approach for its suitability for the artifact.

We focus on two visual factors that we hypothesize bearing importance based on the observations made during the content creation stages of MoSaC project and the artifact requirements. As mentioned in the Chapter 2, the "boxes and arrows" approach of a "mindmap" presented an ever-growing number of connections that, when drawn, would eventually clutter the visualization. It is not only intuitively burdening, but could also hurt the readability, affecting the time of interpreting the diagram and potentially making mistakes while doing so. We are therefore interested in optimizing the transition visualization of "boxes and arrows" to make the resulting flow structures fast to read, yet minimize the interpretation errors. We are also interested in utilizing colors, and by ensuring it provides benefits to similar factors mentioned, incorporating syntax highlighting within the resulting approach becomes justifiable.

Incorporating different shapes for depicting various required elements of the approach might not be beneficial for our applications: More complex shapes than rectangles and circles can introduce needless difficulty for novice modelers to draw the required structures by hand, obscuring the visual representation. Similarly, interpreting these free-hand drawn charts involves a risk of the interpreter misunderstanding the minute details in various shapes if complex or closely similar shapes are used[8]. As we aim to achieve flipchart drawability and readability, we have to consider the possible illegibility of free-hand drawn structured and therefore keep the complexity of the shapes to a minimum.

4.3.1 Depicting transitions

One goal for the artifact is to simplify the structure. One mean for achieving this is to maximize the readability of transition notation and minimize the amount of transition arrows. Purchase (1997) tested the readability of graphs and evaluated the reaction time and error making by considering five different factors in forming those graphs: "bends", "crosses", "angles", "orthogonality", and "symmetry". Any of these factors can affect a graph simultaneously or be present without the others having their own effect on the readability. Rearranging the graph or manipulating the

[8]Even drawings shapes as simple as circles vary by individuals and cultures (Ha and Sonnad, 2017)

transition notation, and thus minimizing the effect of these factors, can significantly improve the readability of the structure.

The line crossing of transition notation counting for the lines that overlap each other can be considered crucial for making graphs easier to understand (Purchase, 1997). An example of line crossing can be depicted by overlapping transitions between two pairs of components, as visualized in Figure 4.9. By rearranging the components of our example, we can remove the line crossings, as seen in Figure 4.10. A lesser effect is achieved with "minimizing the number of bends" and "maximising symmetry", visualized in Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14. No apparent effect regarding the readability was found within the other factors.

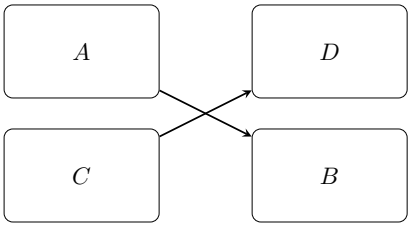

Figure 4.9: The reference arrows cross each other and form a cross-section.

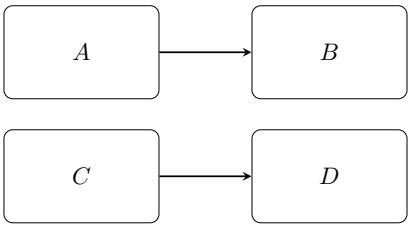

Figure 4.10: The line crossing resolved by rearranging the structure.

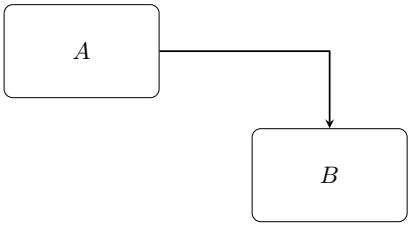

Figure 4.11: A line is bent to reach from A to B.

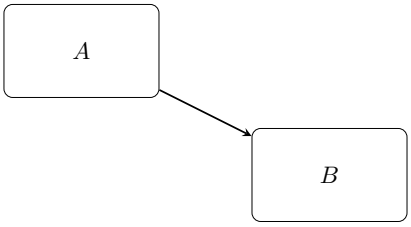

Figure 4.12: The bend is removed by making the line straight.

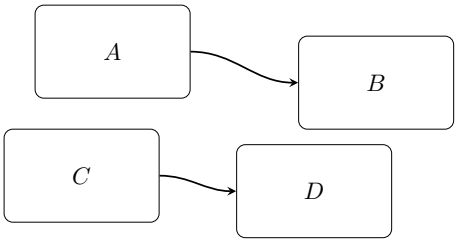

Figure 4.13: The symmetry is broken, as the components aren't horizontally or vertically aligned.

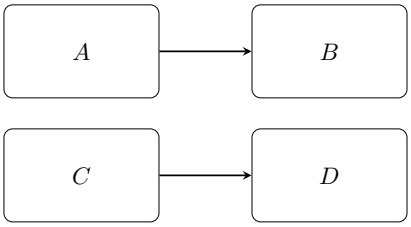

Figure 4.14: The symmetry is improved by aligning the components with each other.

All of these techniques can be utilized to make visual structure more readable, but Purchase (1997) also argues that minimizing linecrosses can come with the cost of less symmetry. We might therefore benefit if preserving symmetry wouldn't require forming line crossing at all: Removing the needless arrows that interconnect adjacent items can in our case diminish the need for transitional arrows to a minimum while benefiting from the points reviewed considering bends and symmetry where explicit transitions are justifiable in non-neighboring elements.

4.3.2 Syntax highligting in graphs

As we are interested in using the artifact as a whiteboard-drawable and digitally modelable graph, the use of colors is possible – as markers and digital color – to further increase the reading accuracy and speed. Depicting different parts of the artifact can potentially work in favor for improving the overall readability.

Reijers et al. (2011) studied how the use of color highlighting would affect the reading accuracy and speed in process models. Their hypotheses were that using colors to highlight matching operators (nodes in their models) would have a positive impact on both reading speed and accuracy. They further hypothesized there being a difference between novices and experts in favor of novices compared to experts with presumably better pattern-matching abilities.

The study utilized Petri Nets as its process model (graph) having transition arrows and nodes forming the logic. With 62 experienced modelers and 41 students (novices in modeling) were selected and their impressions over given graphs were tested against the hypotheses. The evaluation was performed based on a questionaire concerning the logic of a graph, either with or without the highlighting, depending on the test subject. Highlighting was used to distinguish different nodes such as seen in Figure 4.15 without the coloring and in Figure 4.16. The results suggested there being a statistically significant increase in accuracy for the novice modelers. The research concluded that using syntax highlighting this way (in process models) offers a "significant aid for novices to read the models" given the non-highlighted and highlighted graphs used in their study.

This finding poses an interesting point as, by our requirements, we want to cater to different stakeholders with the visual format of the artifact. Therefore, we would benefit from syntax highlighting to improve readability and increase accuracy – and minimize the errors in interpretation – to provide different stakeholders with visual

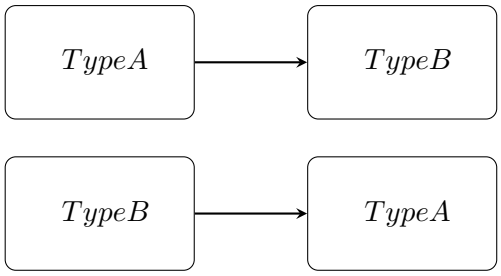

Figure 4.15: If the node-specific secondary notation is otherwise unified, using textual labels can be used to distinguish components of different types.

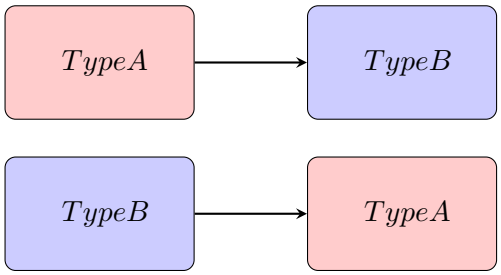

Figure 4.16: If we add an additional notation as highlight colors, we could improve the reading accuracy of different kinds of charts resembling process models.

representations to accurately interpret even when they are not usually working with such graphs or models.

Considering these points, we are presented with at least two clear ways in which the readability of a visual graph-like approach can be increased. Firstly, the way transitions are depicted and the structure is aligned is one tool for us to utilize in increasing accuracy. Secondly, utilizing syntax highlighting for coloring similar nodes can provide novice modelers with higher interpretation accuracy. We can use this knowledge for evaluating our approach by analysing the way our approach visualizes the transitional arrows or the lack of them.

4.4 Learning from these techniques

We are presented with a clear comparison point by the theoretical base provided by both of the different paradigms: flow-based programming gives us a reactive approach against which we can compare our resulting flow with the similar "forward-going nature" of our intended scripts. We greatly benefit from the mentioned parametrization of the predetermined components: Displaying a textual info, giving the player an object to interact with, or playing an audio clip can all be considered separate "components" that need their according parameters to work in synergy with the virtual environment. The VUTS method also utilizes abstraction level varying representations and a composite components like approach that better suits our specific needs. Lastly, as mentioned, the nature of the project puts enormous importance on saving development time, and the modular approach also used in FBP

offers a way to lighten the development load when large applications are in question.

As we also want to track the state of the player within that flow: Statecharts with their hierarchical "zoomable" approach – somewhat similarly to the FBPs composite components – can be used as a point of comparison for the state-depending parts of our artifact that we use to model the parts requiring a higher abstraction level. The activity and action like functionality of Statecharts is possible in our approach where durational functionality is present as well. By utilizing information packets of FBP or orthogonality of Statecharts, we can achieve concurrency to allow the flow to offer multiple simultaneous subflows at once.

Finally, as we want a clear visual representation of the flow, the knowledge on secondary notation can be used to validate our approach for readability. The transition depiction plays a significant role, and by analysing the notation used for depicting transitions we can evaluate how we might have tackled the issues possibly presented by transitional visualization. Utilizing color can aid the accuracy of novice modelers, helping various stakeholders to more capably read and interpret the produced graphs.

Arguably, we could utilize FBP or Statecharts as-is, but doing so would incorporate non-utilizable or detrimental functionality (from the simplicity and content point-of-views), such as unlimited hierarchical depth (that clashes with our design requirement of guidelining the use against creating too complex structures [C_1]) and information packets that transfer computational data around in a reactive manner – being manifestly ill-suited for our player-driven approach (S_2) where the player should primarily move the flow forward. As even a single mismatch in requirements observable by a brief analysis renders the proposed approach sub-optimal to reach our goals, we would require analysing each preexisting solution one-by-one while comparing it to our requirements to presumably face similar faults in their scope.

We can conclude that our specific substance-related, architectural, and content design goals can benefit from a *domain-specific approach* that pinpoints the required functionality and limits the scope to something narrower, while simultaneously satisfying all our defined requirements. It is in our greatest interest not to render the system needlessly complex (by our simplicity requirement) to produce wanted results lead by our quite particular and pedantic expressive needs. Ultimately, we want to keep the usable approach simple, concise and effective for the needs of Virtuario™ content creation, but to have enough flexibility from the system to cater our potentially growing needs in the near and distant future.

Even though none of these approaches precisely satisfy our set requirements by themselves, they provide highly valuable techniques relevant to our problem domain. We can benefit from comparing our approach to the satisfactory parts and show how using similar techniques has lead us constructing the artifact the way we did.

Based on these observations, we can evaluate the VUTS method to conclude whether or not it can be considered as a solution to our problem set defined by our requirements. We have to first review it before we can move to demonstrate and evaluate its suitability.

5 The VUTS method

Our approach incorporates modeling techniques similar to ones present in FBP and Statecharts, and visualization techniques of transition-related secondary notation knowledge in general. We believe this provides the best balance between expressibility and simplicity to allow us to effortlessly model our intended training logic as a structure of lower and higher level flow to satisfy all the given criteria without adding needless complexity by introducing abstractions we don't require in our content creation needs.

In the first Sub-chapter 5.1, we introduce the approach in general by its functional nature. In the second Sub-chapter 5.2, we will present its notation in more detail.

The emphasized definitions in this chapter are original and related to the proposed solution unless otherwise mentioned.

5.1 View-Unit-Task-Step Unified Training Structure

The *View-Unit-Task-Step Unified Training Structure* – or VUTS in short – is the approach and thus our constructed artifact that serves as a flow control framework that Virtuario™ uses in all of its programs, including the tutorial, menu screen, training content and the guidance phase between the menu and the training. VUTS contains the program-specific functionality in its lowest level components that are freely extendable by implementing its abstract methods that are run when the flow progresses onward. On the higher level VUTS defines the control structures creating the non-linear flow. The VUTS flow is run through a specified manager that is initialized with a sequence of complete structures as input, and signals the external system once the highest level terminal structure of each View is completed.

VUTS itself doesn't define any content-specific behavior, as it is merely a framework of a flow similar to FBP or Statecharts. It is up to an implementor to decide how the flow is converted into wanted actions such as player interaction or other game events. Later in this thesis, we present concrete implementations for the lowest level component to further display the capabilities of the framework for providing solutions for real-life scenarios.

At its core, VUTS is a visual modeling structure and the content depicted by it will have a flowchart-like representation. A low-level, detailed example can be seen in Figure 5.1 with a simple training scenario with some textual information preserved.

Another way of presenting the same structure with even less textual information is to use icons instead of text labels. An extreme visual version, with completely removed textuality (but having the action names left present as reminders), can be seen in Figure 5.2.

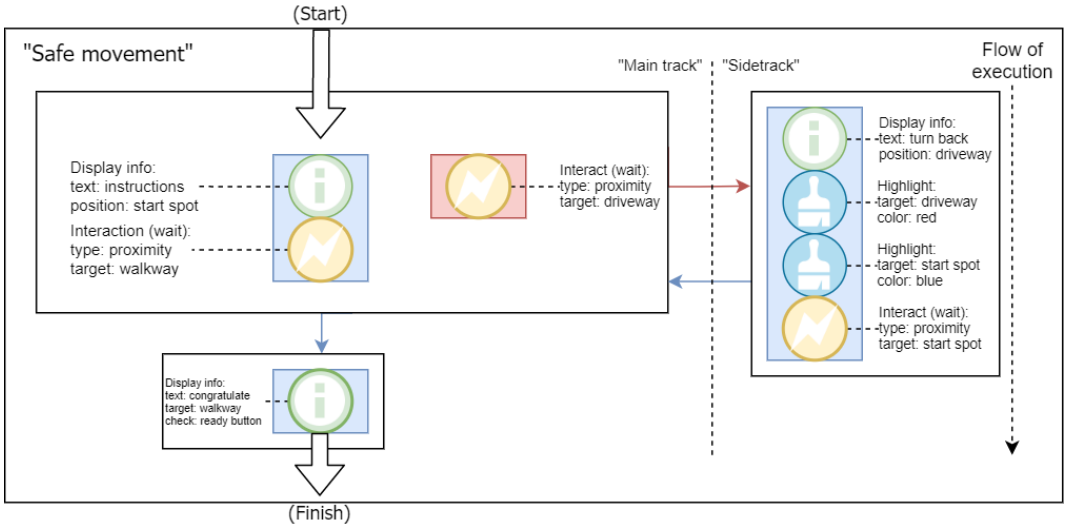

Figure 5.1: Moving safely as a pedestrian: One way to model and represent a simple training about safe driveway movement as a VUTS flow. The control flow branches by how the player moves in VR leading to VUTS activating different parts of the flow structure. The box on the right activates if the player moves hazardously from the starting area, and the bottom one if the player chooses a correct route. Each box has its own sub-instructions that determine the conditions for the branching to occur. Completing a set of instructions of a certain color code leads to branching depicted by an arrow of the same color: “correct” or “incorrect” routes.

5.1.1 Hierarchy

The *VUTS hierarchy* is a four-level structure that forms a directed graph that is used to define a single “scene” at a time. The concept of a “scene” in our use denotes a space in which the game flow happens with meaningful (visible) objects, a starting point, a finishing point, and in-between goals for the player to complete. The levels in the VUTS hierarchy form nested structures and the order of the hierarchy is respected at all times: Each letter in the word VUTS stands for a single level of hierarchy, forming a four-level hierachy that at its lowest level bears resemblance to FBP, and at its highest to Statecharts.

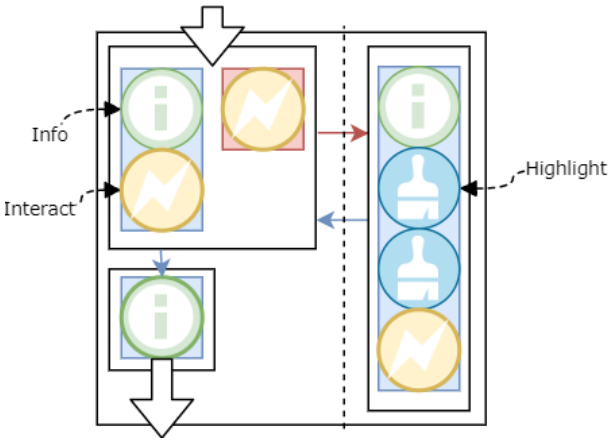

Figure 5.2: The same training content as in Figure 5.1, but depicted in a compact, purely visual, non-textual format. The flow logic of the content itself is visible even if the details are faded in this extreme visualization. These icons and similar implementation-level color codes are used by the Virtuario™ team in Unity to quickly distinguish various parts within VUTS flow structures. The dashed line is left here to similarly emphasize the correct route. The reminder text labels could be removed, as their icons are unambiguously recognizable.

A valid VUTS hierarchy must contain at least one of the lower levels within each other and a middle tier level cannot exist without a corresponding parent or a child. This provides zoom-levels similar to Statecharts, but in our approach, the number of hierarchies is locked down for a straightforward presentation and usage. Each level of the hierarchy defines its own sub-graph and intended rules for the flow to follow. Even if we aim for a general solution, we are not trying to provide an "all-purpose" solution like FBP or Statechart might.

Our approach knowingly "locks" many of the otherwise generalizable features down to concreteness in order to preserve clarity and guide us when designing content suitable for our narrow scope, defining a domain-specific language. By our simplicity (A_1) and guidelining (C_1) requirements, we seek to *simplify* and knowingly *restrict and guide* the complexity of the structure: We are not required or seeking to implement systems more complex than our content creation needs govern us to.

The hierarchy, as a whole, has an unambiguous starting point for the flow but can have multiple exit points: the flow is considered having started once the starting point is entered, and finished as one of the defined exit point has been reached.

As the flow itself is merely a structure that doesn't operate on its own, there has to be an outside system to manage the states and the flow within the structure. For this we use the *VUTS manager* (or just *manager* in short) – a flow controller – that is responsible for controlling and tracking the state and transitions within the structure, starting the flow from a correct point and finishing it accordingly[9]. The job of the manager is to keep track of the flow, as none of the VUTS component keep track of state or independently operate or progress in any way. The manager is in charge of the semantics of the flow structure by defining *how* the flow progresses and should thus be implemented to define and adhere the rules of the progression of the VUTS flow. Each VUTS component only defines its own behaviour to be executed *by the manager* once *called by the manager*. The VUTS components and the manager together form the *VUTS system*, a self-operating flow structure. To get meaningful actions, such as invoke VR functionality, we rely on an *external system* to enable this kind of behavior to be utilized through the VUTS system. This interplay is depicted in Figure 5.3.

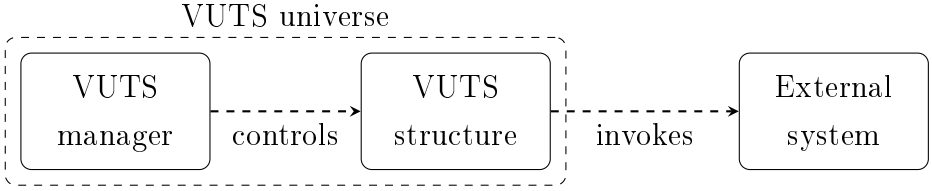

Figure 5.3: The interplay between the manager, the VUTS structure, and the external system. The manager has the control over the VUTS structure which specifies what external functionality should be called.

We are interested in moving within the depicted flow, but instead of being centered around an abstraction such as "information packets" our goal is to metaphorically "move the player" within the flow. This means that designing the functionality happening within the parts of the flow should (in our cases defined by the player-driven progression requirement $[S_2]$) directly be related to the player in some meaningful way, usually by reacting to player interactions, instead of calculating some black-box value to be passed on outside of the player's understanding. This is strongly related to our pedagogical substance requirement (S_2), underlining driving force of the player for the flow progression.

[9]The details about the inner workings of such manager is left outside the scope of this thesis. As the idea of the manager is to "run" the flow, its main functionality will be derivable from the abstract flow logic itself.

The hierarchy can also be thought to present four effective levels of abstraction that serve different stakeholders. Technical details are reduced per each level towards the top. When outlining new content, we can start from the top level and move downwards to define the details level by level.

Content (or View) level is the outermost wrapper of the content and can be used to briefly describe what, as a general idea or a package, is included in the training scenario as a whole. It defines a one logical scene within a sequence that defines the whole training program module. Its fundamental idea is to crystallize the whole flow within a theme or a description. For example, "safely moving in a manufacturing plant" as a description serves as this kind of "content level" goal.

Pedagogical (or Unit) level is about the learning goals of given flow in a format of high-level objectives and their order of execution. This level provides a birds-eye-view, state machine like representation of the modeled content and answers to the question of "when" the player is learning a topic relevant from the pedagogical point-of-view. For example, defining different learning goals of each separate phases within a training is a "pedagogical level" goal.

Design (or Task) level provides a general idea of "what" the player should and can do within the flow. It is more relevant to the non-technical writers and customers as a design aid and easier to understand as it hides much of the implementation level boilerplate. For example, defining all flow branching points and their high-level conditions by determining what separate actions player can do at any point within the training would be considered a "design level" goal.

Implementation (or Step) level is the lowest level, as it exposes the inner workings of the implemented parts of the flow and their parameters. This level is relevant to the technical side of the design team and answers the question of "how" the system guides the player and how the player manages to do given objectives within the environment. For example, displaying a floating info window once the player has interacted with an object can be considered such a "implementation level" goal.

The abstraction levels can together be presented as a design flow map. This kind of a representation can be used as a base for the design process and to display the final product to various stakeholders. An example of this is shown in Figure 5.4 where four levels of abstraction are depicted.

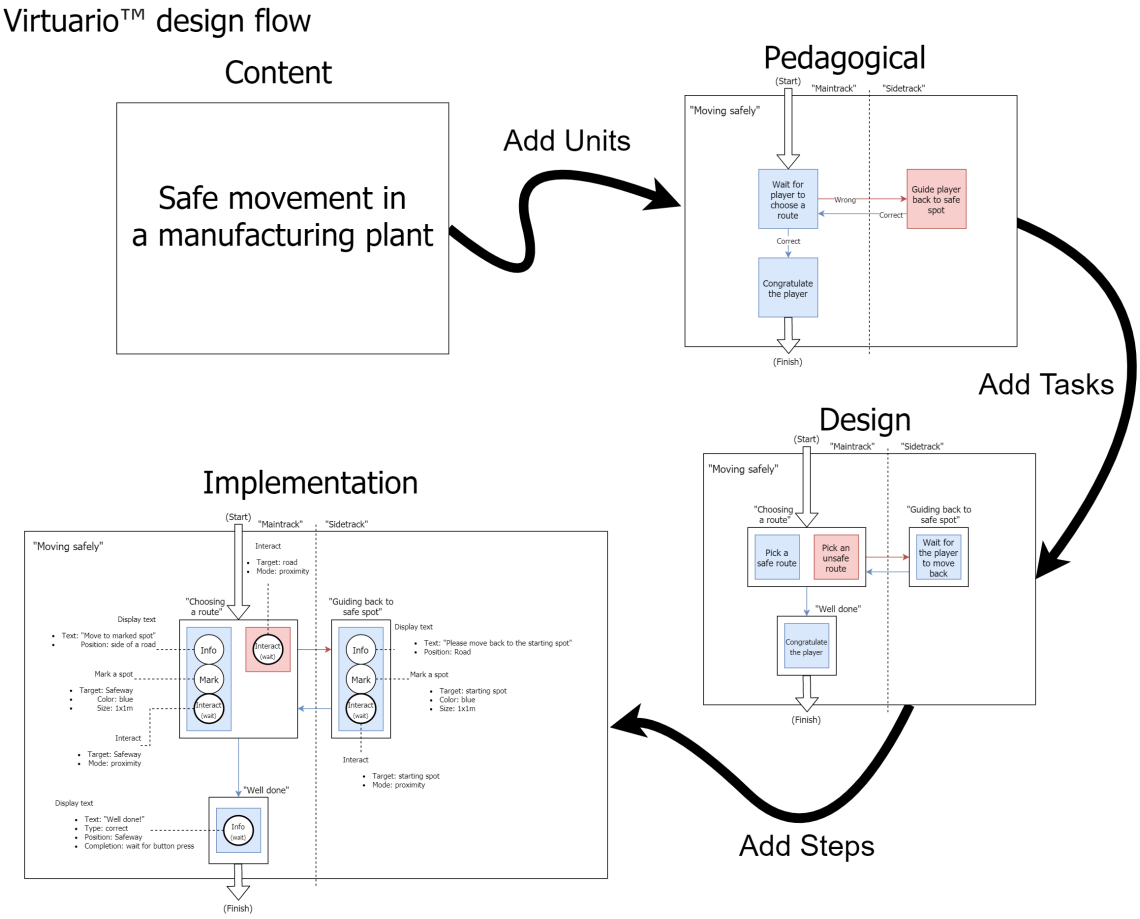

Figure 5.4: Different abstraction levels depicted. Each level can also be used a design phase for writing the script: Starting the content design from an idea leads to building the high-level flow, followed by determining the possible groups of actions and finally breaking them down to single instructions.

In the next sub-chapter we introduce these levels as their respective VUTS components. We introduce the levels from the lowest, implementation level first, progressing towards the highest, idea-driven content level.

5.1.2 Step

Steps are the lowest level components of VUTS, defining the implementation level. Each Step is a building block of an atomic operation to be implemented once and reused in different content scenarios. Steps represent parametrized functionality that can perform various time-based actions, defined by the concrete implementation provided by the implementor. It shares similarities to the lowest level component of FBP as its function is to execute implemented actions as a part of the flow structure when given a turn. Our general notation for a Step is a circle, drawn in Figure 5.5.

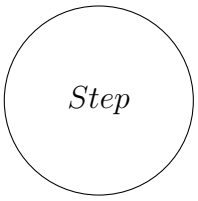

Figure 5.5: A single Step depicted by its circle notation.

Steps form sequences between each other and gets their executional turn once the previous Step is finished. This sequence forms the base for our lowest-level flow-like structure, a *Step chain*, seen in Figure 5.6.

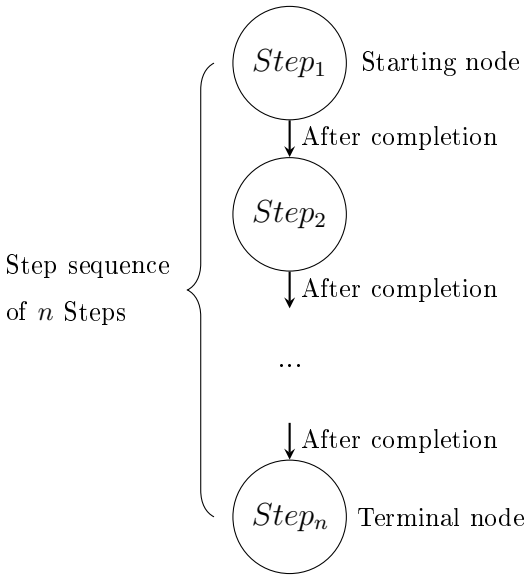

Figure 5.6: Steps forming a sequence.

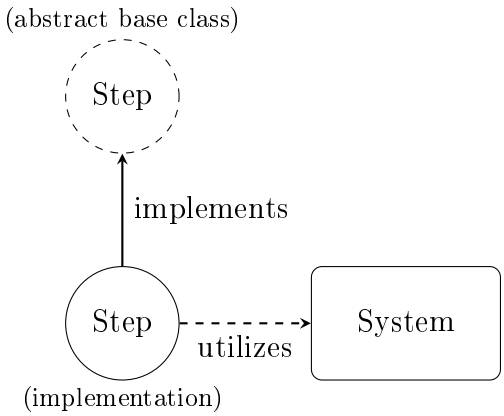

Figure 5.7: The relationship between a Step and the external system, such as a VR framework.

The Steps are freely programmed by the implementor, and can thus fully utilize the underlying system, be it game engine or anything else, to reach a wanted outcome.

The Steps are the only parts of VUTS that are intended to communicate directly with the external system, such as a VR framework, by executing functionality to alter the state of the system, visualized in Figure 5.7. Side-effects similar to that (affecting the general program state, not just its own if having any) are in fact desirable to manipulate some outside state, such as in our case the virtual environment.

Each Step has a *lifetime* – the timeframe during which the Step is active – which can be divided into three stages called *phases*: *entering*, *executing* and *exiting*.

Entering a Step means waiting a predetermined amount of time, a *delay*, that (if set) serves as a temporal buffer that delays the actual execution of the Step functionality.

Executing a Step is when the functionality defined by that Step is invoked. The inner workings of this phase are defined by the implementor, allowing them to choose how the execution lifetime of the Step is utilized to perform meaningful actions.

Exiting a Step finishes its functionality and means that this part of the flow can proceed forward. If some external functionality was started in the executing phase and wasn't specified by the Step to be stopped, it will stay running until explicitly reset externally.

To control the executing phase, the manager is required to iterate through each Step in progress. For this, the manager specifies *an update interval* that defines how frequently the manager polls each Step to invoke their executing phase defined functionality, and single cycle within it a *frame*. By having control over the update interval, the invocation of the functionality can be set to match the framerate or other update frequency of the external environment where VUTS in implemented.

The entering and exiting phases can be defined to contain functionality that trigger as each phase is reached. The delay value can be provided by the design as an additional parameter to allow using any implemented Step multiple times within a flow with different delays. Parameters in VUTS come from "outside" the framework. This means that there is no FBP like defined methodology for providing the parameters, and the implementator can choose which way the parameters are obtained.[10]

[10]If the Steps are defined, as the implementor wishes, within a development platform as separate programmable objects in "editor space", the parameters can be defined as textual values within the editor itself provided within the objects to be used once run.

The executing phase is where the actual functionality takes place. The execution can be *instant* and thus perform some action right after the entering phase is done (that is, right after the delay) and then move to the exit phase. This might, for example, mean playing an audio clip or displaying a text box – anything that can be considered to happen in an instant manner, similarly to the actions of Statecharts.

Another way is to define the execution phase of the Step as *durational*. Steps defined in this way execute over a duration of time, and in the case of Steps tailored for the use of Virtuario™, their durations are commonly defined in the magnitude of seconds. When a durational Step is executing, a *progression* value p is provided to the Step by the manager and can be used to affect the executional code of the Step. This value p is defined as

$$p = \frac{\text{time passed since starting the execution}}{\text{execution duration}}$$

The value of p is naturally clamped within zero and one. This therefore simply represents the "progression percent" where zero is the starting value that grows gradually to one over the timeframe of Steps duration. Using p for performing an interpolative action makes sure functionality executed over p does not depend on the duration of the Step, but rather on a relative value of its phase of execution. This design makes it more convenient to implement relative functionality that executes similarly (but in a slower or faster way) when the durational parameter is changed.

A concrete example of utilizing this parameter is implementing an interpolative animation Step that moves an object between two points over duration d using p to map the linearly interpolated position between these points. When p reaches one, the execution is considered done, and the execution phase is terminated[11]. If d is set to zero, the Step is then considered instant and in this case would merely set the position of an object to the end position. If d is set to something greater than zero the Step would then interpolate the position and apply it to the parameter-provided target object, as seen in Figure 5.8.

This approach is similar to the activity model introduced by Statecharts: we can start a functionality when the Step is entered and finish it once the exiting phase starts – or as the executing phase is over. For simplicity, there is no need for the implementor to track how much time the durational execution would take compared

[11]This way we can create interpolative animations for Virtuario™ that execute *strictly within* the Steps making sure they are finished once the next Step is started and that their completion isn't affected by changes on the duration parameter.

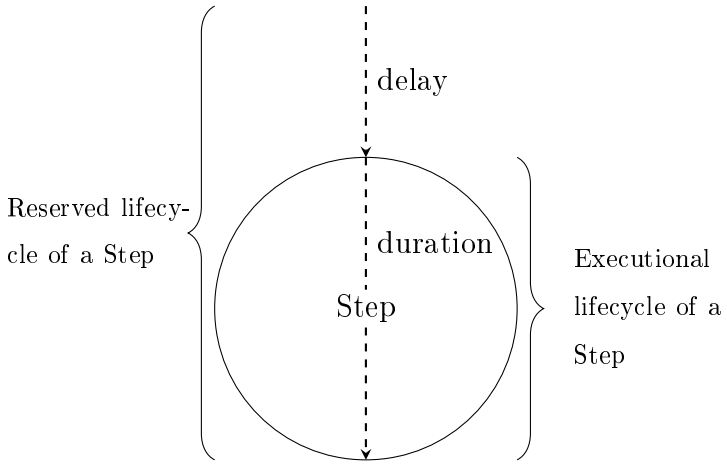

Figure 5.8: A temporal life of an executed Step.

to the lifetime of the Step, as it is always tied to the duration itself, being executed right after the (possibly delayed) entering phase. The total lifetime of a durational Step is therefore:

$$\text{Durational Step lifetime} = \text{delay} + \text{execution duration}$$

The third option of using a Step is making it *waiting* and disregard the duration altogether. A waiting Step stays within its execution phase until a condition is met. The Step specifies the waiting condition by listening to event of an external system (or another Step) to be invoked. The manager waits for a callback from the Step once the condition is met. Similarly to other Step functionality, how the waiting condition is defined depends on the Step implementation. The manager is only concerned with the callback to determine the Step is ready to be completed. An example of this would be defining an interactional Step that waits for the player to perform a certain interaction, such as pointing an object, before continuing. This type of Step usage enables the implementation level "player-drivenness" of the VUTS as the execution can be made to halt until the player has done something meaningful, making the player the controller of the flow. The total lifetime of this kind of Step depends on the enter phase delay and on the external factors of when the condition is met. We can define the lifetime intervals for this kind of Step as follows:

$$\text{Waiting Step lifetime} = [\text{delay}, \text{delay} + \text{event trigger time from execution start}]$$

As with all the VUTS components, Step lacks any side-effects inducing functionality itself, but differs from other VUTS components by providing an abstract base for

the best suited functionality to built around by defining the executional phase implementation. The functionality is invoked once each Step is granted its execution turn in the Step sequence by the external manager running the graph. The manager ultimately ensures each Step gets called for their entering, executing, and exiting methods appropriately.

When the Step is at the end of its lifetime, the terminating method is called by the manager, allowing the implementor to define whether or not to undo the actions executed at start and/or during the interpolation phase. Ignoring the terminating method allows for creating cumulative (semi-)persistent effects that stay triggered until later cleared. Combining an instant preparational step (eg. "Highlight the area"), waiting for desired condition to be met (eg. "Wait for the player to move inside the area"), and clearing the preparations (eg. "Remove the highlighting") is a base for the interactional synergy used in Virtuario^{TM}. Each Step determines its own undo method for controlling how the state should be reverted.

If the the end of the lifetime of a Step is never reached, for example making the waiting condition of a Step never satisfied, we deny the following Step their executional turns. This allows the completion of the Steps earlier in the chain to determine whether or not the later Steps in the chain get their executional turn.

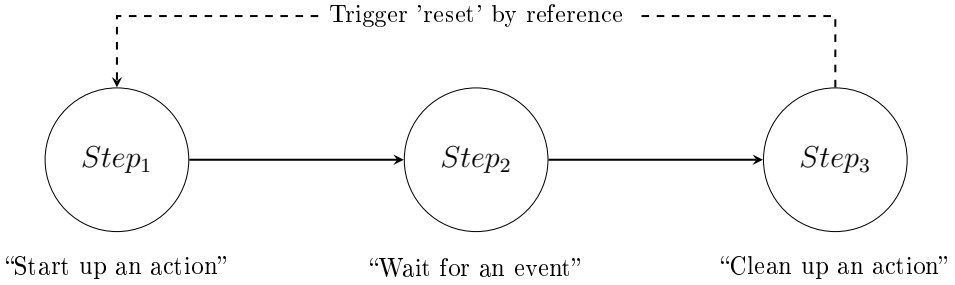

Figure 5.9: Initializing, waiting, and clearing behaviour by concrete Steps. An action is run in $Step_1$, followed by a $Step_2$ that waits for an external event to happen, and then $Step_3$ is run, which defines its functionality to be reseting $Step_1$ to undo the effect. The player can notice the effect of $Step_1$ when directly affecting or waiting for $Step_2$ to complete and seeing the effect disappear once $Step_3$ is executed. For demonstration purposes the Steps here are unconventionally depicted horizontally.

The inter-Step functionality can also be more direct. One core functionality of Steps is the ability to reference other Steps, mainly to deactivate or reset them "remotely".

Deactivating a Step means that the Step will be skipped by the manager if it is *further* in the chain, and the flow will be guided forward like it would after finishing a Step. If a deactivated Step was run *earlier* in the chain, the Step can be considered also to be reset. *Reseting* a Step means undoing its functionality like it was never run at all. By creating a "Reset Step" that can hold a reference to another Step, this references can be used to remotely undo the effect of a previously executed Step within the start behaviour of a later Step, visualized in Figure 5.9, without deactivating it – in case it will be visited *again*. This allows for a delayed "cleanup" operations to happen by other Steps, but also the reverse is possible: A Step can be proactively deactivated by an earlier Step, thus ignoring it as explained. This kind of functionality can seem hazardous, but as each type of Step needs to only be implemented once – encouraging for sustainable solutions – and this kind of Step usage can waive the need of constructing different kinds of Step sequences for a particular goal or coding mechanisms for whether a condition is met, this approach proves itself valuable. It is ultimately up to the implementor if such functionality is necessary, and such resetting and deactivating control structure mimicing Steps can be left unimplemented.

It is also noteworthy to emphasize that Steps don't themselves carry any concrete data around within the flow to pass forward. There is no "data flow" to pass to the following Steps. This is one of the main differences that differentiates VUTS from FBP. The VUTS manager is in charge of starting, executing and finishing each Step. If the Step is implemented as having inter-Step behavior to be executed during its execution phase, the manager is in charge of invoking this functionality as merely defined in a Step. Ultimately, it is up to the implementor to implement referencing within the Steps if there is a need for reading the state of other Steps.

5.1.3 Task

Steps, without any further structures, are always completed in a linear fashion: They do not branch or run in parallel. They form the sequental base for the flow to progress, but don't provide any structural way to depict choices, simultaneous flows, or loops[12].

[12]Technically, by manipulating the deactivating idea discussed in the Step sub-chapter, one could implement the different "branches" as sequental Steps that would selectively be deactivated to form seeming different paths. This method would understandably be hardly sustainable and difficult to follow.

Tasks are the second lowest level and define meaningful groups from the Step sequences by adding *concurrently running Steps*, *Step chain looping* and *exit results*. The Tasks are fundamentally parallel Step chains, each containing its own sequence of $[1, n]$ Steps, which are run from the first Step forward, and progressed according to the temporal and eventual criteria of each Step.

Concurrency in VUTS works at Task level: Multiple Tasks can be used to define simultaneously run Step chains, *sibling Tasks*. To concurrently run multiple Step chains, the manager keeps track of each separate Task and a pointer to the a Step within it currently in execution. As each Step chain can only have one Step in execution at a time, this means that each Task forms its own subflow and the number of Tasks equals to the maximum number of simultaneous subflows.

The running order of each Task is determined by its priority in relation to other Tasks. The Steps within Tasks are started and executed starting from the Task of the highest priority and progressed according its Step chain, moving to the next Task of higher priority when the Step chain is depleted or the chain progression halted. The manager iterates each Task through sequentially within a single iteration cycle (based on the update interval) and once the length of that cycle is depleted, the next Task is similarly started and iterated through. If a Step is defined as a single-shot functionality (without any time-based parameters or waiting) it is executed and the next Step within the Task is started within the same cycle, as no iteration time was consumed in the process. If the Step incorporates a delay that hasn't yet depleted, the manager continues the iteration from the Step of the next Task. If this kind of break is introduced, the manager adds this Step to the pool of running Steps and continues forward. The pooled Steps are kept in execution until completed, independent of other Steps in progress.

The manager therefore runs through all the available Steps within a Task depth first until a delay, durational execution or a waiting phase occurs. If there are no time-based breaks or waiting within the Task, its Step chain is completely run through until the execution moves to the next Task. Considering each Step to be started and executed gets their own turn on each frame, there is no resource starvation when run like this. The Task order is respected by the manager: The running order of each Step is based on the starting order of each Tasks. Each separate sibling Task can increase the Steps to be added to the pool by one. When the Step is completed, it is removed from the pool before the next delayed, durational or waiting Step is added in it. The Task concurrency is visualized in Figure 5.10.

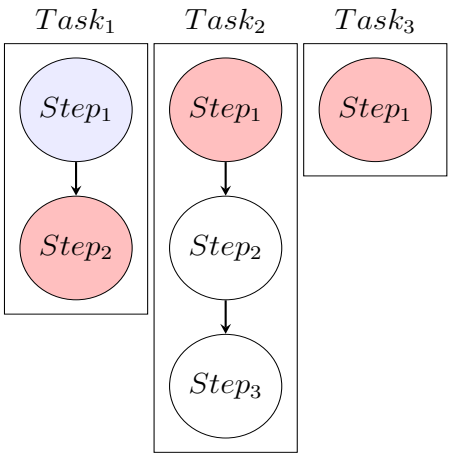

Figure 5.10: An example of Task being run through, ordered by priority. The Steps currently in execution, the last Step of $Task_1$ and the first Steps of $Task_2$ and $Task_3$, are marked in red. The already completed Step, the first Step of $Task_1$ is marked in pale blue. This can depict a situation where $Step_1$ of $Task_1$ has had its one-shot functionality completed and the Tasks current in execution are durational and therefore halting the execution further within the Task due to their execution not yet being finished. The red Steps define the pool of running Steps.

Without any delays, durational execution or waiting conditions happening between Steps there would be no sense of parallelity, making the Tasks just a sequentially run groups of Steps. The breaks make possible to define arbitrarily many parallel Step chains that can work independently and wait for the completion of different events or timers. This logic is somewhat similar to orthogonality of Statecharts where "sibling" states start and run simultaneously by given conditions. If the conditions are identical, the Step would likewise complete during the same frame, following the order determined by the Task priorities. By having different Steps with differing conditions, the Tasks yield more "interesting" chains of events, such as waiting for two separate interactions to happen.

If a Task reaches outside its terminal Step node, meaning that the Step chain has been depleted – there being no Step to continue to – the Task is considered as *completed*. What happens next is up to the set exit result parameter of the Task. If the Task is marked as *success*, this signals that the flow should be directed towards a next group of Tasks as chosen by the designer. If the Task is marked as *failure*, the flow is directed towards another group of Tasks, chosen by the designer in similar fashion. In both cases, this also means that all the sibling Tasks are instantaneously terminated before going forward and all the Steps inside reset to clear the possible

effects – unless triggering resetting is explicitly marked as "manual only" and no automatic "Step cleanup" is performed after transition.

If the Task is set to *neutral*, nothing happens other than leaving the Step chain, and all the other sibling Tasks continue running. A neutral Task can also be set *repeating or looping*, making it start again from the first step once the last one has been completed and doesn't perform a sibling Task reset when doing so. A repeating Task can only be neutral, as it won't direct the flow anywhere but only start again from the beginning of its own Step chain. Introducing this kind of behavior without any safe mechanisms would lead to starvation of other Tasks in cases where a repeating Task only contains one-shot Steps. If a Task like this was started from the beginning after its completion without corporating any kind of breaks, it would keep looping without any breaks in between. To prevent this, the manager waits a single frame after starting a repeating Task again *if* there was no frame progression during the execution of the Step chain within that Task. This is equivalent of giving the first Step of the Task a delay of a length of the duration of a single frame after each repeat cycle. As the number of Steps within a Task is finite, each Task is given a turn in all cases.

Combining the capabilities of waiting-based Steps and Tasks allows creating control structures the player can affect: When inserting a completion waiting Step inside a Task with an exit result E_1, another Task with a different exit result E_2 could be created on the side with a durational timer Step inside it. Now the branching occuring by E_1 would be dependent on the completion of the Step action whereas E_2 would be timed independent of the external factors. This could, for example, be used as a simple timer to branch the flow dependent on whether or not the player has completed a certain action within a time limit.

5.1.4 Unit

As mentioned, the exit result of the Task interrupts the flow and branches it accordingly. The *Unit* component is what encapsulates these sibling Tasks and point the success and failure branches where the flow should progress. All the sibling Tasks form a Unit that defines a boundary for simultaneous activity. Once a Unit is entered, all the Steps within its Tasks are sequentially started, and once the Unit is exited (as determined by the Tasks), all the Steps within its Tasks are terminated. The flow is then guided forward and the process is started again. This means that no more than one Unit can "run" at each time: Doing so allows for creating clear

functional states of possible actions and makes sure previous unwanted actions are cleared when the new ones start.

Each Unit contains at least one non-neutral Task leading to another Unit or terminating the flow if no leading Unit is set[13]. If the exit result of the Task is triggered, the Unit *targets* the flow to another *destination* Unit determined as one-to-one mapping of exit results to Units. This structure is similar to both the state-based structure of Statecharts. There is no restrictions to Units forming loops this way, so they can effectively form a deterministic state machine that always holds the progression within a single Unit. Likewise, there is no restriction which and how many Units can be pointed by other Units: The Units can diverge or converge freely – even point to itself – as long as "many-to-one" mapping by different exit results is respected so the leading Unit would be unambiguous, as seen in Figure 5.11.

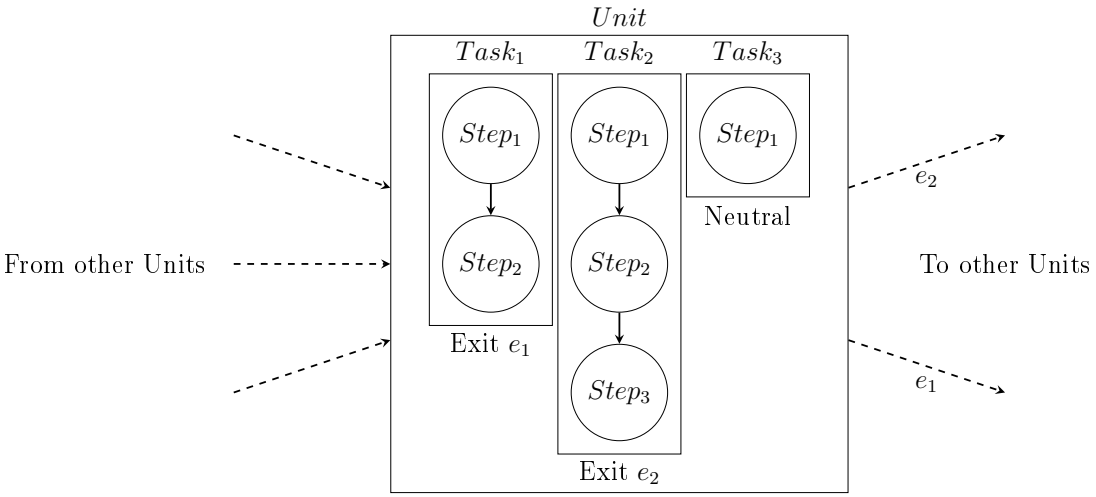

Figure 5.11: A targeted Unit containing Tasks of various lengths and destinations.

If the destination of a Unit is left empty, the Unit is considered *final* and the whole execution is then terminated and the whole flow is set to be completed. As there is no restriction which Unit the exit result pointer points, the Units can form loops with each other or themselves.

In Virtuario™, the most common branching by a Unit is to separate an "incorrect" and "correct" selection aftermath into separate Units and guide the "incorrect" Unit flow back to the starting point. This allows creating "testing loops" for a player

[13]In Virtuario™ training, we *want* an exit action to a training, as completing the training session is preferred. Having a sole neutral Task within a Unit by their definition would never lead out of the flow.

to make sure the player performs the correct actions before going forward in the training. If the Tasks are set to be waiting and defined as the player *can affect* which Task to complete, such as by providing different interactional targets for the player to interact within the Tasks, the choices made result in branching of the flow. The Unit-formed state machine like structure, depicted at a high level, can been seen in Figure 5.12.

As a simplification, Virtuario™ restricts the number of outputs to two, which are named as "fail" and "pass" derived from their most common usage pattern explained. This thesis follows this convention as it further increases the simplicity of the VUTS structures.[14]

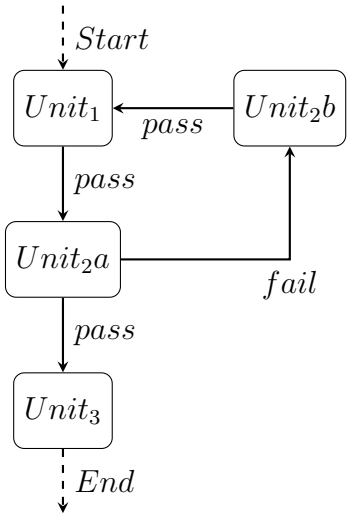

Figure 5.12: An example VUTS structure at Unit level using Virtuario™ pass and fail destinations (or "correct" and "incorrect").

5.1.5 View

View serves as the outermost container of the VUTS hierarchy. It defines one of its inner Units as the starting Unit and has terminal points defined by the container Units as explained in the Unit sub-chapter. Its functionality is to wrap the Unit graph to a single referenceable entity that can be used as a complete flow structure.

[14]There could be a non-restricted amount of exit results and Unit mappings for them, but as this Boolean approach fundamentally allows for infinite branching and all the examples and the own implementation of Virtuario™ follows the two-branch approach, we settle with the said pattern.

A new View can be loaded after completing the previous one to form the final training program: This is the way it is used in Virtuario™. A View also serves as the said highest abstraction level for depicting content as a general package – or sub-package in our case. The final top-level structure can be seen in Figure 5.13.

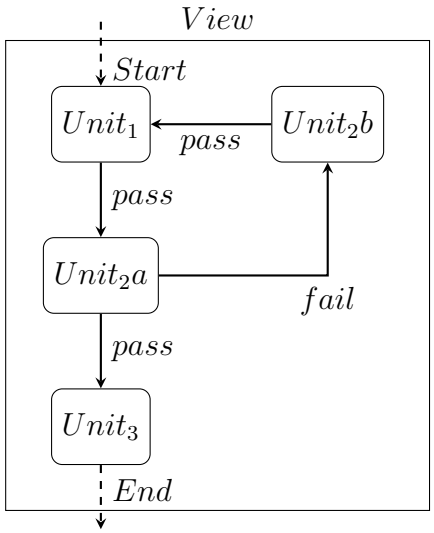

Figure 5.13: A View encapsulated VUTS structure.

5.2 Visualizability of the structure

One aim of the VUTS method is to provide a visual way of designing and compositing training logic. The structure should be light and easy to read. One core feature of the VUTS structure of its ability to be drawn by hand with minimal textuality required. Outlining a VUTS flow doesn't require using complicated shapes, precise figures or – most crucial to our liking – drawing complicated networks of arrows between each and every component.

Minimizing the edges between different "nodes" of VUTS (the VUTS components) was a primary motivation when designing the visual representation of it. As the layout of the VUTS components is primarily hierarchical the need for explicitly depicting intercomponent relations via arrows is minimized. So far, we have used arrows between Steps in a chain but as the execution direction is always from top to bottom, we waive the arrow notation for depicting this Step progression. As we still would require "arbitrary" branching, the cross-Unit referencing allows for guiding the flow between different components outside the otherwise nested structure that

Steps and Tasks form. The selected study (Purchase, 1997) within the Sub-chapter 4.3 about the use of arrows or edges is extremely relevant to us, as we aim to make the readability high and our way of minimizing the use of them can aid in tackling the issues.

As using colors can improve readability among novice modelers (Reijers et al., 2011), we encourage the use of color codes for different implemented Steps: Different Steps of the same implementation and Steps of intuitively "similar goals" (but different implementation, such as different Steps that provide "cosmetic functionality") share the same colors in our examples. The primary notation (the notation that fundamentally distinguishes an element from all the others) of each Step can be a Step icon or a textual label, depending on how the media on which the VUTS structure is drawn, such as paper by hand or in a digital form using a graph drawing software; the color-coded secondary notation adds a layer of extra readability.

For Tasks we use color as their primary notation to distinguish the two exit results and the neutral Tasks. In case there is no color available, we rely on writing S for "success" or F for "failure" on top of each Task to mark their exit results. Neutral Tasks are left "default-colored" and also no textual label is required. For looping Tasks, we use R for "repeat". As repeating Tasks can only be neutral, there won't be need for using multiple letters simultaneously.

The Units can utilize color as their secondary notation. We *recommend* limiting its use to depicting the pedagogical ("structure") level content as color-coded Tasks are abstracted away. Using colors in Units on this level can serve as making a distinction between the "main track" and "side-track" Units. Additionally, the transition arrows between Units can be color-coded following the convention of Task coloring: The color-code can be made to match the exit result. If there is no color available, "pass" and "fail" labels next to each transition are drawn.

Considering again the example VUTS structures presented in the Sub-chapter 5.1.1. We will call the one with textual information present "detailed" and the one in purely visual form "simplified". The simplified version is now presented without any text whatsoever, both depicted in Figure 5.14 with color-centric notational conventions visible.

Hierarchy-wise, four structural elements and one transitional elements can be explicitly notated:

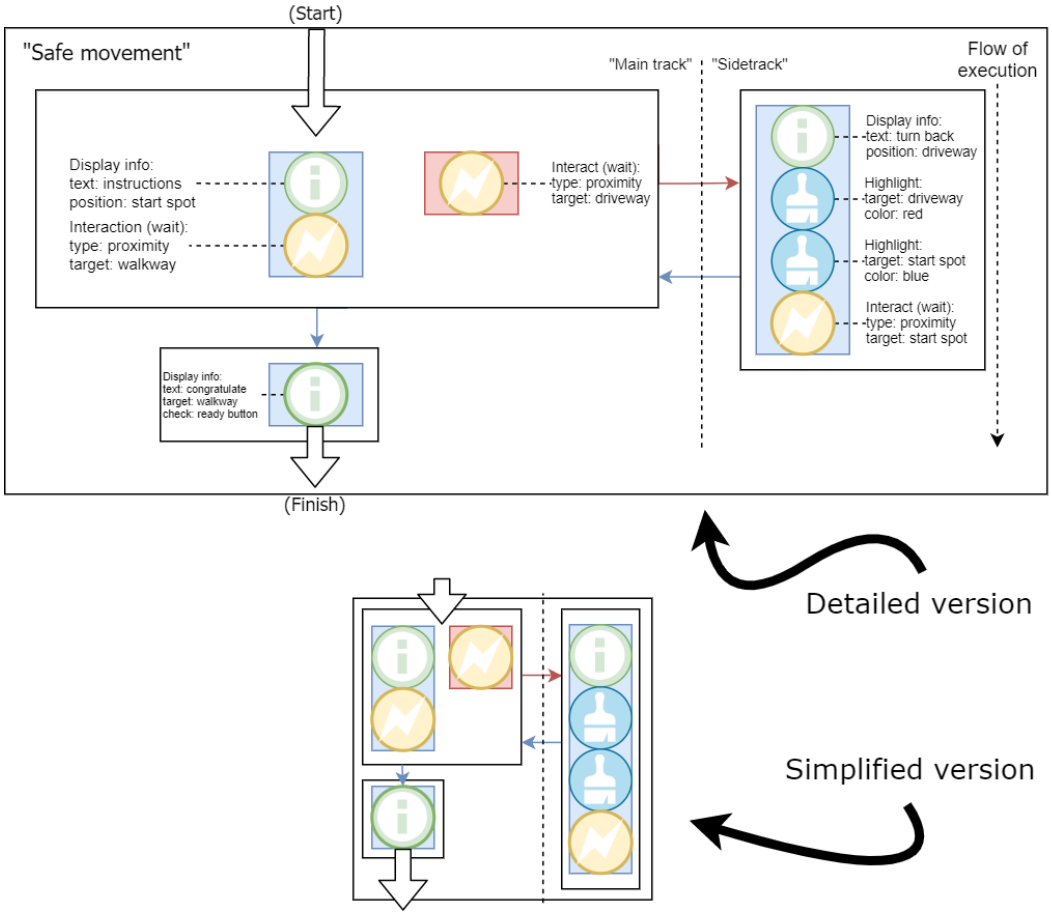

Figure 5.14: One VUTS structure depicted in two format previously shown in Sub-chapter 5.1.1. The Steps and Tasks here are color-coded as well as the Unit transition arrows.

Steps are presented as circles with an icon and a color code. When parameters are required to be shown, they can be written next to the Steps. Their running order is read from top to bottom. When drawn on a paper using limited colors available, short text labels can be used instead to provide similar Step-distinguishing notation. To depict waiting points, bolding is used. Another way we propose is to use underlining, as this can be easier to draw quickly. As noted in the Sub-chapter 4.3, sticking to one notation is preferable.

Tasks are presented as rectangular containers surrounding Steps. In our Boolean "correct" and "incorrect" branching format, we depict "correct" resulting Tasks with blue background and "incorrect" resulting Tasks with red background. The Tasks have a running priority from left to right.

Units are presented as rectangular containers surrounding Tasks. They are interconnected with *arrows* – pointing from a Unit to another Unit – that are color coded by the respective exit results of Tasks contained within the Unit. The arrows – and ultimately the flow within them – determine their running order but in our examples they follow similar top-to-botton ordering, making the starting Unit the one on the topleft. The "intended path" flows downwards and "sidetracks" are depicted on the side of the "main track Units"[15]. A "sidetrack" barrier can be drawn to distinguish the correct path, but doesn't bear any functionality.

View is presented as a rectangular container surrounding Units. The "idea" or the name of the content can be written on it to give a short textual explanation of the flow. The starting point of the View can be marked with a "starting arrow" and all the terminating Units can similarly be depicted with "finish arrows". The starting arrow has a purpose, as the starting point is not derivable from the VUTS structure itself, as any Unit can serve as a starting point. The "finish arrows" on the other hand are already implicitly included in the flow, as each Task with an exit result *not* redirected by the containing Unit is considered terminating – as the flow is redirected outside the Unit but the Unit isn't pointing anywhere. As this kind of overly subtle notation might be too discreet, we can explicitly mark the finishing Units with similar arrows. If only one View is depicted, the container can be waived.

The higher the abstraction level goes, the more textual descriptions can be used to describe functionality abstracted away. With these visualization rules applied, we can draw a simple, Step-level, VUTS structure on a flipchart, depicted in Figure 5.15, only using letters as textual labels for separating different Steps.

As shown by the example and the structure element description, the amount of transitional cues is limited as the transition inside Units are omitted. As the Steps within Tasks are always read from top to bottom and each Task within its container Unit is run in all cases, there is no need to explicitly mark these Step chain transitions or sibling Task groups branches with any pointing notation. Considering the points presented about the importance of the minimizing edge crossings, yet also preserving symmetry and eliminating bending (Purchase, 1997), using VUTS for

[15]This makes lenghty VUTS structures grow mainly downwards and caters the portrait oriented papers, such as flipcharts, better. Positioning the forwards going Units horizontally and the sidetrack downwards would alternatively make landscape drawing easier. In our simple example in Figure 5.14, due to having only two "main track Units", the "downward growing" isn't shown. Adding more subsequent Units would obviously change it.

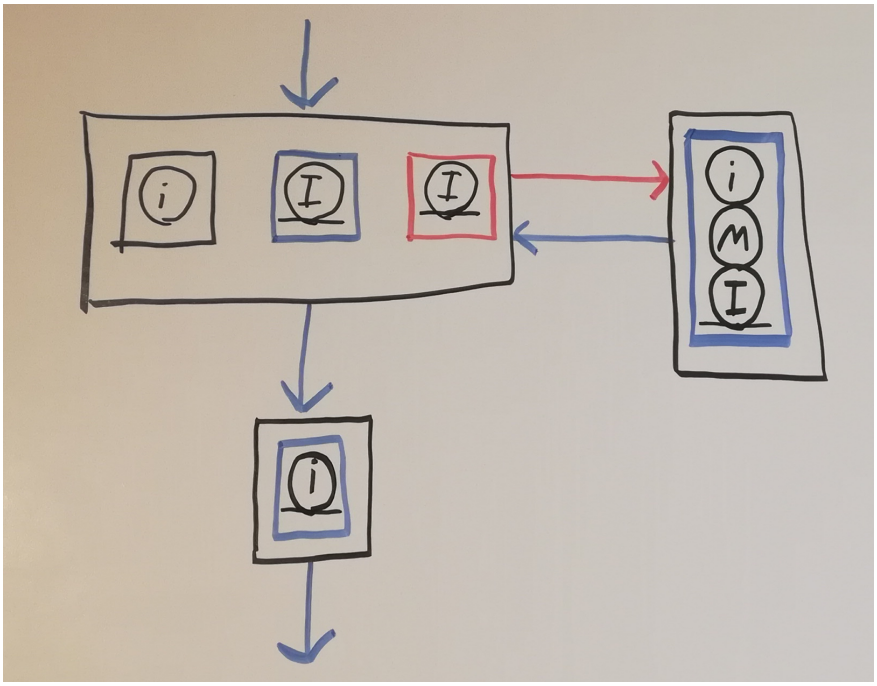

Figure 5.15: A simple VUTS structure drawn on a paper covering a View: "Display an info. If the player chooses the first route, go forward. If another route is selected, go to a Unit where an info is displayed, an area marked, and wait for the player to go back, and start from the beginning. When the forward route is selected, display an info and wait for the ready button to be clicked." This example also demonstrates the obscurities of free-hand drawing. For the sake of an example, we use underlining here as waiting point cues, compared to our other examples where bolding is used.

our content creation needs makes edge crossing needless while preserving symmetry and eliminating line bending.

As we omit all but the Unit transition arrows, we can depict the needed small-scale cross-Unit referencing as simple straight (non-bending) and non-edgecrossing lines that connect the "sidetrack" Units to the "main track" Units, visualized in Figure 5.16 and Figure 5.17[16]

These simple theoretical cases show how different Unit structures can form clear graphs – in the sense of the transition depiction related secondary notation – and these still contain complex hierarchical behavior inside them in form of sequential Steps and parallel Tasks.

[16]This is by no means a proof for freely modeling planar graphs with non-bending edges but merely an example of creating Unit hierarchies don't necessarily require "complex wiring" as most of their true functionality is hidden inside.

We have yet to show these kind of simple structures can be used in real-life content creation needs. In the next chapter, we will demonstrate how seemingly simple format like this can produce usable safety training logic by scripts.

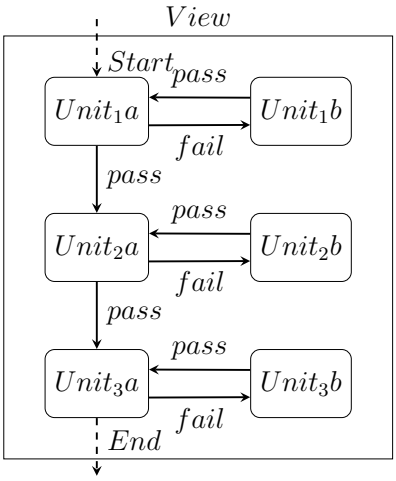

Figure 5.16: A "complete" sidetracking VUTS structure of three main Units and their branches. The need of edge crossing is minimized, as the high-level structure is kept simple.

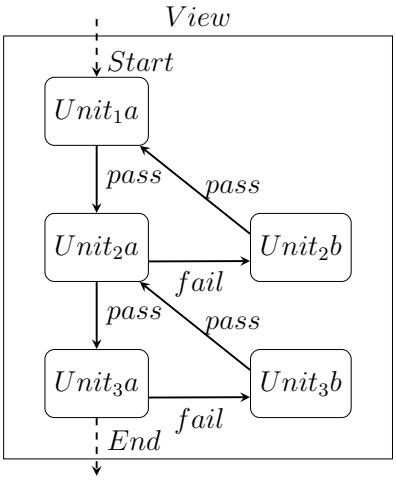

Figure 5.17: A similar sidetracking layout to Figure 5.16 but the sidetracks guide the flow further back. This kind of diagonal structure also allows for non-crossing edges.

6 Demonstrating VUTS with real-life scenarios

In this chapter we first show how VUTS can be integrated into the Unity3D environment to be utilized in content creation. In the second sub-chapter we utilize the Unity-based integration for implementing three different case examples from actual Virtuario™ content.

6.1 VUTS implementation through Unity, utilizing VR

This sub-chapter is intended to shed light on how the actual implementation could be done to utilize VUTS in a real-world application. As this thesis is more about the approach as an abstract model, this technical implementation-specific explanation is left brief and non-exhaustive and won't go into the code-specific details. We are nevertheless driven to know whether or not the VUTS method can be implemented in a game development environment to be used in actual content creation. The emphasized terms here refer to the Unity scripting API and Unity manual (Unity Technologies, 2019b,a).

In Virtuario™ VUTS is implemented through Unity3D, as it also serves as the general development platform of Virtuario™. As Unity3D provides a visual editor that can provide various visualization aids, we can use them to implement VUTS effectively. Emphasized terms in this sub-chapter are defined by Unity3D and are not specific to VUTS.

Unity3D allows for creating *scenes* that serve as wrappers for simultaneously existing objects with functionality and positional, rotational and scalar information, *GameObjects*. That spatial information incorporated within GameObjects as separate *Transform* component. Unity3D comes with various components that each have their own functionality, such as reacting to physics, playing audio or detecting collisions. The developer can implement their own components as *scripts* that can be similarly attached to GameObjects where convenient. The scripts can be made to expose their parameters as textual, visual or referenceable fields. These fields can be edited in the editor and the serialized changes will be used when the application is running.

In addition to editor view accessible numerical values, Unity3D allows for "drag-and-dropping" other objects as reference parameters in scripts and GameObjects can this way cross-reference each other without needing to spesicify compile-time

references as hard-coded values in scripts. This is a powerful tool that allows us to implement the Step-referencing of VUTS in a light way that doesn't require us writing any code for fully parametrizing our Steps.

The objects are being tracked by a hierarchy panel that displays each GameObject as separate entity including their potential hierarchy: Each GameObject can be *parented* to another GameObject to form *parent–child* hierarchies that make the parented child object move in relation to its parent object. The hierarchy can be seen in Figure 6.1.

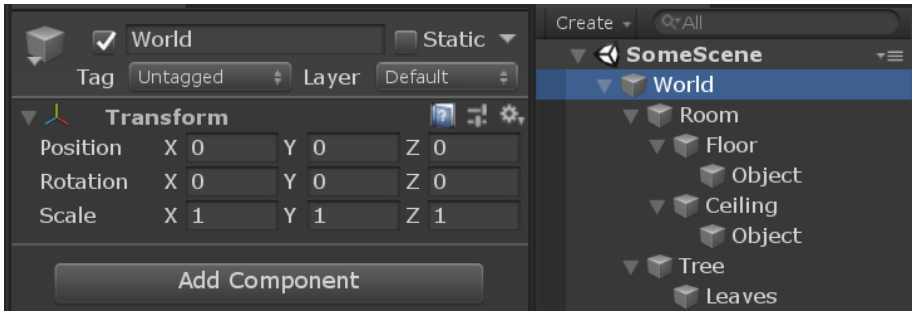

Figure 6.1: A screenshot of the hierarchy panel of Unity3D and a visible Transform component of selected object on the left. The child-parent hierarchies are visible by the intendation of each object name.

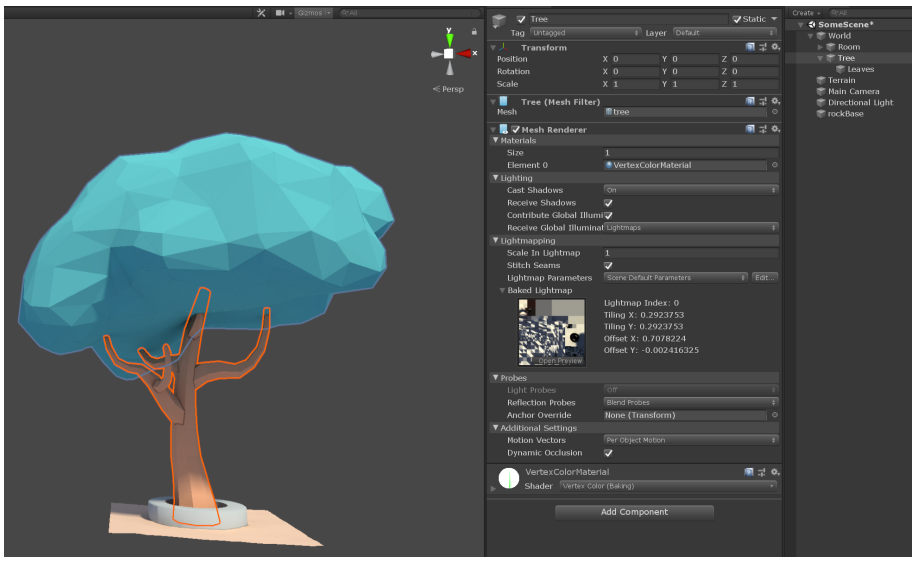

Figure 6.2: An object in the hierarchy is selected and highlighted within the scene.

There is clear interplay between the objects in a scene and its hierarchy: Selecting an object through a hierarchy panel highlights the object within the scene and vice

versa, as seen in Figure 6.2. This behaviour can be enhanced by implementing custom *gizmos* behaviour that allow further visualizing objects in scene when a GameObject within a hierarchy is selected. This is something we can utilize with our VUTS implementation to, for example, highlight the referenced objects supplied as parameters.

The VUTS implementation in Virtuario™ utilizes the hierarchy panel of Unity extensively: The whole VUTS structure is represented as multiple parent-child hierarchies by parenting lower-level components under the higher level ones. VUTS structure lives in the scene as GameObject with corresponding components added to them. As each GameObject also included the Transform component, we can utilize the spatial information as an extra set of parameters where applicable. A VUTS hierarchy with Steps having parameter generated names can be seen in Figure 6.3.

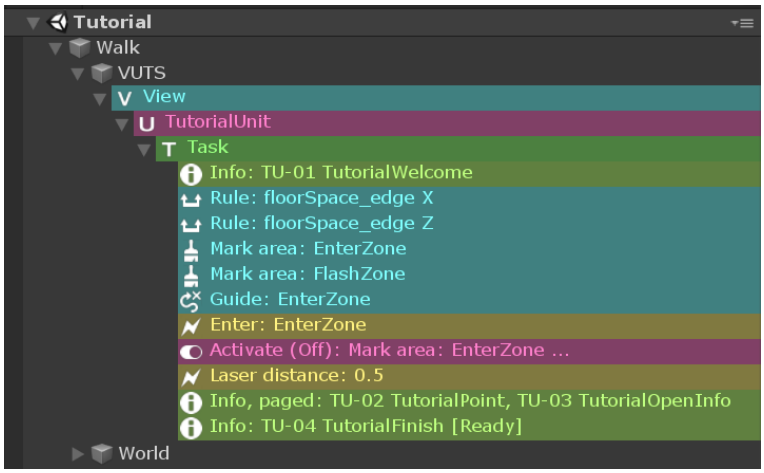

Figure 6.3: A screenshot of the hierarchy panel of Unity3D and a Virtuario™ tutorial modeled as a simple single-Unit and a single-Task VUTS hierarchy. Dedicated GameObjects are assigned for each component. The secondary notation is used in a form of icons, color coding and intendation.

To implement the Steps, we write the Step as an abstract base class for a script that is then overridden by our concrete implementation. The delay and duration are supplied as fields in a form of exposed parameters defined by the base class and being assigned values by the content designer utilizing the Steps implemented by the implementor. An editor editable Step component can be seen in Figure 6.4, selected within the hierarchy.

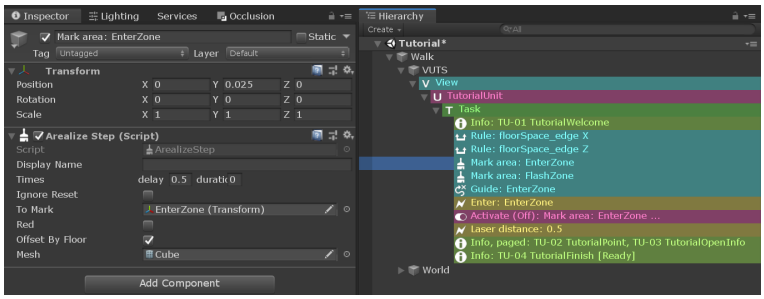

Figure 6.4: As a Step object is selected, its scripts are shown. In this case a Step of a type "Arealize Step" was selected and the editable script instance view is exposed for editing its fields.

In order for the VUTS system to "support" virtual reality, we need to implement functionality that can utilize capabilities offered by VR. In the next chapter, the actual use is demonstrated through real content for the training needs of Virtuario^{TM}. The examples will show how VR functionality is implemented within Steps making VUTS therefore a working platform for VR-supported training content.

6.2 Using VUTS in Virtuario^{TM} training content creation

Based on our substance requirements (S) we can distinguish three different flow patterns that play a center role in our content creation. Using these three models in tandem allows us creating the required content:

A linear flow (S_2: Player-driven flow progression) that doesn't offer the player any branching or parallelity, but ensures the player follows through a single line of instructions with maximal focus.

A loop-forming flow (S_3: Training through a distinct path) that can make the player experience moving laterally within the story rather than going forward through minor "sidetracks", but lead the player back on the "main" track.

A "subflow" structure (S_4: Observable parallellism) that can make the player focus on multiple tasks at once, but ensure simplicity within the structure so that the player can easily follow the progression.

Based on these three "progression styles" we present three different simple real-life

training scenarios that are modeled using VUTS. A script for each scenario is used for modeling the required structures including the behavior. The basic high-level layout for each of these scenarios can be depicted like in Figure 6.5.

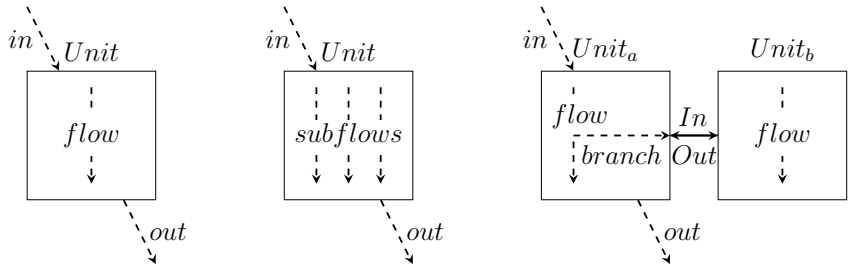

Figure 6.5: Three different layouts: Linear, subflow and looping progression that can be used together to form larger "super-flows" or used as-is.

In all of these three scenarios, the modeled environment is static: Nothing the player does alters the environment other than presenting additional visual cues *on top of the existing* environment. This does not have to be the case, as implementing Steps that manipulate the environment is naturally possible. But our examples happen to model scenarios that lack dynamic objects to be moved within the environment. These examples take a different approach where dynamic markers and instructions are drawn on top of the static world; a technique that is often utilized in Virtuario™.

6.2.1 Linear case example – Virtuario™ tutorial

The Virtuario™ tutorial is a short, approximately one minute long, linear introduction to virtual reality as used in Virtuario™. Before the actual tutorial begins, the player is guided to stand in an assigned spot and has two virtual laser pointing controllers in their hands. The script can be described as follows with the required funtionality underlined and waiting points **bolded**:

> "Once the tutorial starts a welcoming info window is displayed and the virtual play area is highlighted with its edges marked. A smaller area in front of the player is also highlighted and **the player is asked to step on it** to continue the tutorial and shown a guiding arrow pointing towards the area. Once the player has entered the highlighted area, the opened info window closes and a new window is opened telling the player how to use the controllers in their hands and the laser beam length of the controllers is set to 0.5 meters to introduce the idea of variable

> laser length. **The player is asked to press the "next button" on the info window to continue**. Once the player has clicked on the button and the next page is shown, another minimized icon is presented to the player's left. **The info window asks the player to click a floating icon**. Once the player has clicked on the icon, it expands into a new info window (introducing how a minimized info window works) that tells the player the actual training content can now be started **once the player clicks the "ready button" within that info window."**

We can notice that displaying an info window and waiting for the player to click on its buttons usually constitutes an operation that can be implemented "atomically" as a single Step that displays the info and halts its execution until a button is pressed – if required. All required atomically considered functionality can be defined by the underlined generalized actions:

- Displaying an info window (minimized or in full size, completing automatically, or waiting for the player to press a button)
- Highligting an area (any size)
- Visualizing object dimensions by their length (any target object)
- Showing a guidance window (pointing at any target object)
- Interacting with the environment (different interaction methods, any interaction target object)
- Deactivating an object (such as a window)
- Changing the laser beam length (to any distance)

By implementing the general actions as Steps and determining the flow structure based on the script, we can model the outlines of the tutorial in at least two ways, seen in Figure 6.6. This kind of completely linear flow can be presented in a single Unit and a single Task within it. The different "actions" presented by the script are divided into eleven Steps. In the Figure 6.6 the left flow can be read from starting from the top, first executing the functionality that displays an info window (implemented as an "Info Step"), followed by showing dimension markers ("Ruler Step"), highlighting the whole VR area ("Mark Step"), a smaller area for the player to walk into ("Mark Step") and finally drawing an arrow that guides the player to move

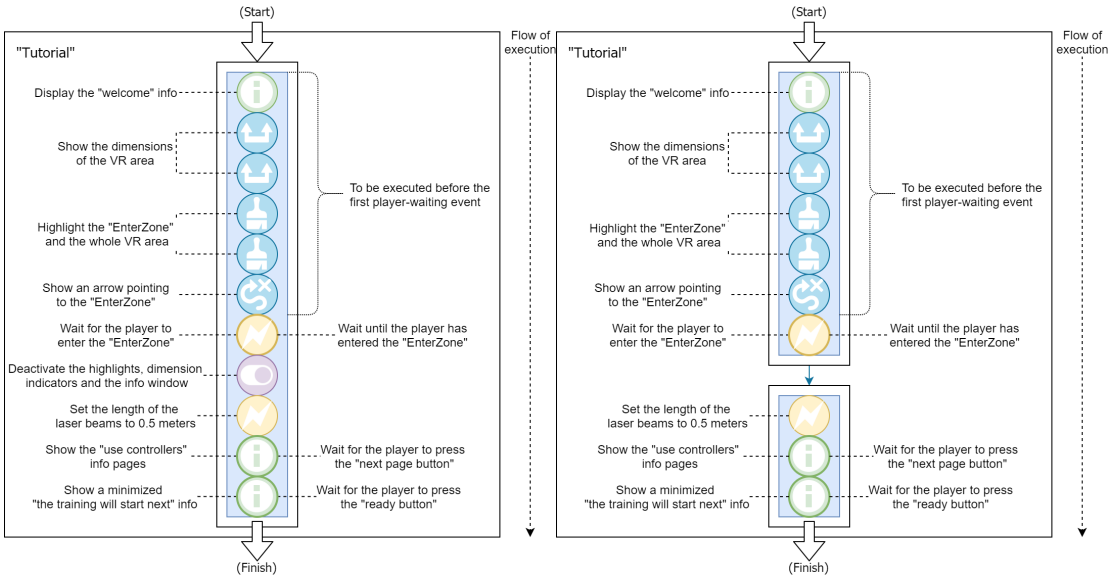

Figure 6.6: Two equivalent VUTS structures for this script: Instead of de-activating all the previous functionality, we can simply set the following functionality within its own Unit containing the last three Steps. The Unit transition triggers a reset in all the Steps in the previous Unit. This results in exactly the same behaviour in this case. The parameters are depicted on the left side of both flows.

within that area ("Guide Step"). These all Steps happen "within the same frame" in our implementation, meaning that their order doesn't matter here, unless they would cross-reference each other and therefore depend on their executed behavior. Reordering Steps allows creating different ordering configurations for different content designers to suit their intuition: There is no right or wrong way of ordering different Steps if they lack any durationality or interoperability. Similarly, ordering Tasks can serve various design styles of different content designers if the Steps within them can be started in any order, such as drawing (or otherwise modeling) "correct" Tasks on the left, and the "wrong" Tasks on the right.

After executing all the previous Steps, the flow halts and waits for the enter the highlighted area ("Interact Step") as it is set as completing once a wanted external action is done: In this case, the player walking inside the area. This part, from the player's point of view, can be seen in the screenshot in Figure 6.7.

There are multiple ways of depicting similar behavior by VUTS structures. In the previous Figure 6.6 the flow on the left manually deactivated all of the toggled functionality before displaying a new info window. An alternative to this is to separate this into two distinct Units. Instead of manually de-activating the previous function-

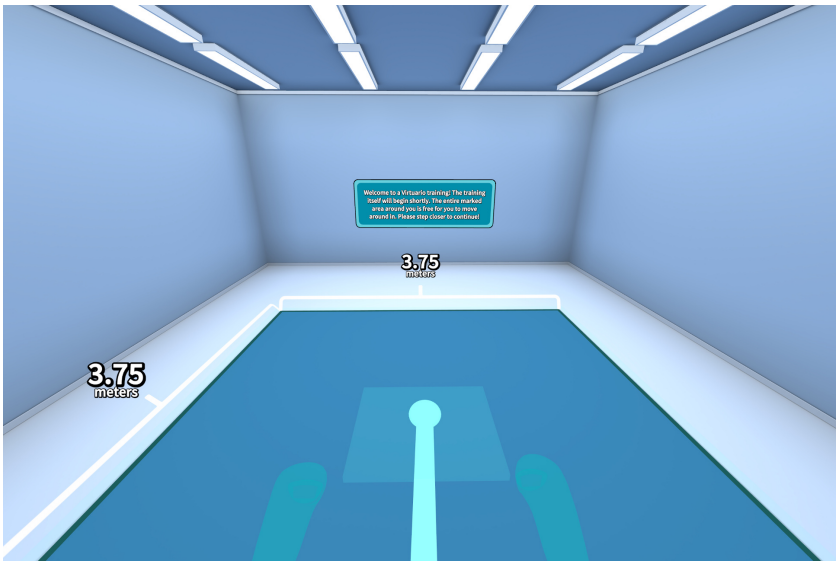

Figure 6.7: The moment after starting the tutorial from the player's point of view. The info window, the dimension markers, and the highlighted areas can be seen. The player is waited to step inside the marked area.

ality mentioned, we can let the Unit transition automatically take care of it. As each previously run Steps are "cleaned" in a transition, this ensures the de-activational functionality is applied without explicitly doing so. Similarly to the freedom of ordering Steps and Task, we can decide whether or not we want to place our functionality within multiple Units or use a single-Unit layout – as any configuration here is possible and different layouts can provide technically equivalent functionality. We can model the flow resulting flow as with two Units to separate different "phases" within the tutorial, or as a single-Unit variant and manually deactivate previously invoked functionality.

Both ways are equal in terms of the functionality observed by the player. The visual difference between a two-Unit layout and a single-Unit layout can be seen in the VUTS structure depicted on the right hand flow in Figure 6.6.

Once the player has completed the desired action of walking on the highlighted center area (completing the "Interact Step"), the markers can be removed and the info window closed ("Activate Step"[17]). The laser length of the player's controllers is set to 0.5 meters ("Interact Step", with its dedicated parameter controlling the

[17]A Step like that can be implemented to contain joint functionality of both "activating" and "deactivating" target objects. In this case we use the "Activate Step" with its mode parameter set to "turn off" and apply it to target Steps that implement their own deactivation as resetting their initialized functionality.

laser length[18]) a new info window presented ("Info Step"). The player is asked to click on the "next page button", seen in Figure 6.8. The execution is similarly halted (by the "Info Step") as in the previous case of walking in the area. The player can freely move around, but no new activities are introduced until the player proceeds by clicking on the button and thus executing the "Info Step".

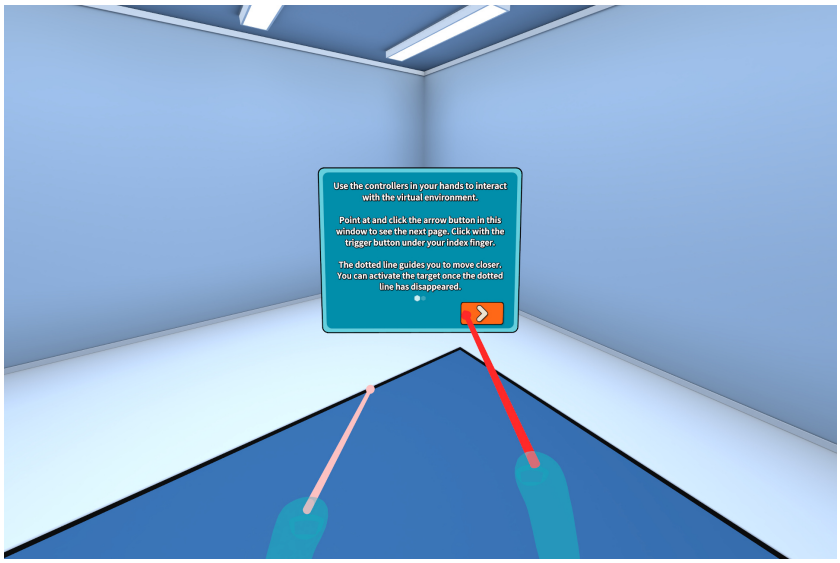

Figure 6.8: The player has entered the highlighted center area and is being presented a new info window. The execution is halted until the player clicks on the "next page button".

After the player has clicked on the button, the next page is displayed and the Step is marked as completed, and the flow can go forward. The Step displays a new page of info, not affecting the flow itself: The moment the player presses the button, the second page is displayed and the info is concluded as being fully read (as both pages have been displayed), so the player can go on. The next Step of the exetion displays a new minimized info window that the player is asked to open by clicking on it. The execution is halted until the player does so, as seen in Figure 6.9.

[18]Arguably, creating a dedicated Step just for controlling the laser length like so could be justified, but since usually – in our scenarios – laser length is changed right before the player is asked to point at world objects, both of these actions were incorporated within a single "Interact Step" that can have its interactional functionalities set off by the paramerers if there is only need to change the laser length. In this special case we want to shorten the laser distance for an info window usage and two kinds of interactionality are required – *if* implemented like this.

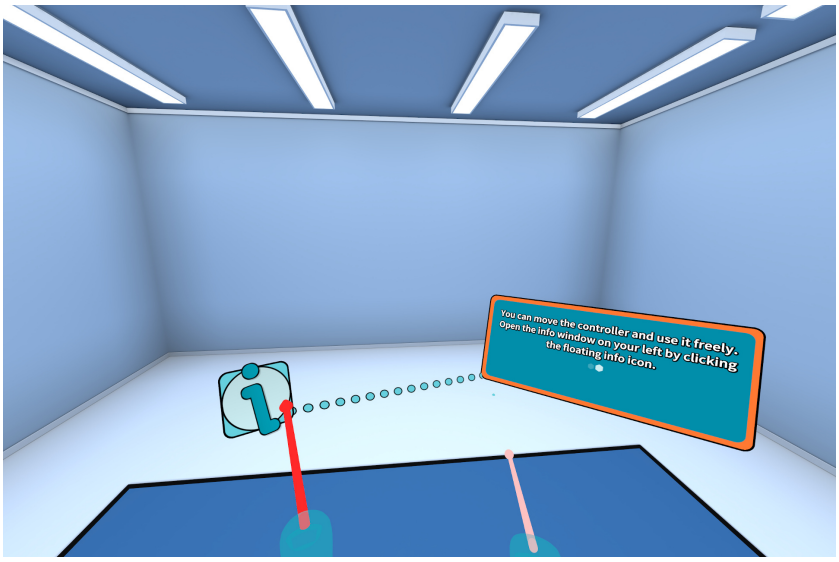

Figure 6.9: The player has clicked on the button and a new minimized window icon is presented. The player is asked to open the info, and is about to click on it.

Finally, the player is asked to click on the ready button on the last info window. Once the player has done so, the flow can be considered completed, as the final Step has been executed and the containing Task points the flow out of the Unit that has no leading Units. The moment before the player clicks on the button and when the flow is in its last Step is depicted in Figure 6.10.

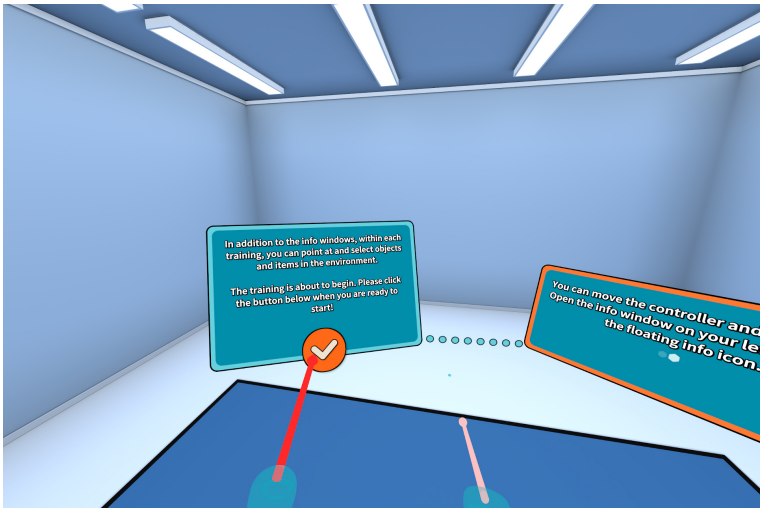

Figure 6.10: The player has opened the final info window and given the last brief instructions. The execution is halted for the last time before the flow is concluded to be finished, once the player has clicked on the ready button.

Figure 6.6 shows the "break points" of the flow with bolded outline: First one of them being a world interaction (walking on the highligted area) and the last two being the info windows that require player clicking on the buttons. This kind of notation can aid us in analysing the amount of information the player receives in different phases of the flow and help us find the possible "pedagogical bottlenecks" that can present too much stimuli to player at once. This matches to our guidelining requirement (C_1) stating the approach should help us easily evaluate how complex a content modeled with it approximately is.

This scenario introduced a simple, linear case, of VUTS usage. We can easily add more complex flow mechanics to the structure by utilizing non-linear flow.

6.2.2 Subflow case example – Observational training

A common usage pattern in Virtuario™ scripts is having multiple discoverable "target objects" – as safety observations – present at once (as seen in Figure 6.11) and allowing the player to find them all in any order they want. In some cases, the player is given direct information about the amount of the objects to be found such as in a form of having an info window stating the "observation counter" as a graphical progress bar. In other cases the player is not given any hints about the progression, but led forward once all the necessary observations have been made.

Figure 6.11: This environment offers a "free-roam" approach to safety learning, allowing the player to find and point the objects without any predetermined order.

In this case example, the player steps out of an elevator and is presented an office space containing twelve safety deficits. The environment is introduced to the player

before the actual "observing phase" and after acknowledging the instructions, the player can start walking around the virtual area in order to find all the objects.

A shortened script, with the exact observation-specific details omitted, starting from the point where the player starts making the observations could be described as following:

> "The player is simultaneously presented twelve observable objects around the environment. The player can walk around the circular area around the elevator[19] and **point and click any discovered targets**. Once the player points and clicks on one of the twelve target objects, an info window is presented near the object containing info about the observation. **The player continues making observations until all twelve observations have been made.**"

As implied in the script, this kind of training content requires support for simultaneous waiting events to allow the player to freely choose from multiple points of interactional targets. Each interaction poses its own "aftermath" actions, in this case opening corresponding info windows. As Task components allow for simultaneous Steps to be executed within a Unit, it is the structural part we can use for solving this case: Each observation ("Interact Step") and its corresponding info window ("Info Step") can be modeled as its own Task with these Steps inside. Completing one of these Steps should make the whole Unit "completed", as there are other observations to be made.

This interactional parallelity allows for the player to make any observation in any order. Each observation opens up its own info window and making new observations "minimizes" the last opened info window, leaving a floating checkmark icon visible within the environment. This can be seen in action in Figure 6.12. If this kind of behavior of "managing" other info elements is handled through its dedicated manager, there is no need to cross-reference each Info Step. This kind of behaviour can well be implemented outside the "scope" of VUTS – assuming this kind of behavior is wanted. Each "Info Step" is nevertheless executed and not reset, making the expected behavior as something that leaves functionality enabled until reset is invoked.

[19]The player can always walk freely, and it is why this is not counted as an "action" concerning the flow unless some kind of target destination is set, like in the tutorial example.

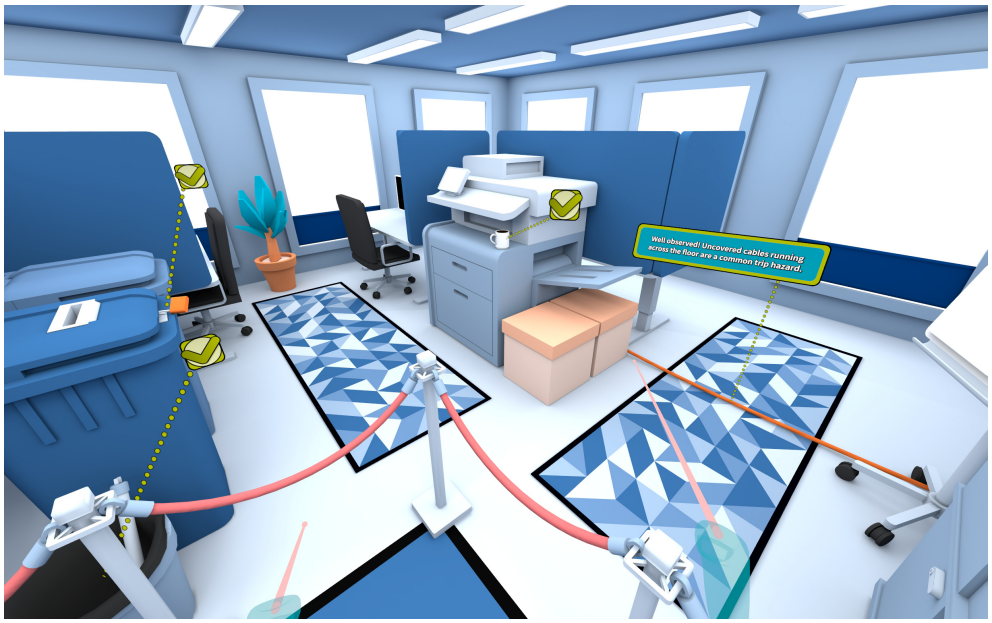

Figure 6.12: The player has made several observations by pointing and clicking at objects within the environment. Each observation is followed by an info window that later minimizes into a checkmark icon as new observations are made.

There has to be some kind of mechanic for tracking the observations made to know when the player is ready to proceed. By creating a Step that functions like a counter-based locking mechanism, we can use that Step to wait for all the observations having been made, manipulate the counter value after each observation and let the flow pass once a target value is met. The "Interact Step" of each observation ensures that the flow within a Task waits until the player has actually made an observation. After the observation has been made, and the "Interact Step" is done, we can manipulate the counter value by one, for example, and wait for the counter to reach twelve[20].

Once the player has found all the targets, a minimized info window icon can be shown and the flow can wait until the player has opened it and pressed its "ready button". Pressing on the button completed the Task and finishes the Unit and the flow. This is modeled in Figure 6.13.

[20]This is only one possible solution, and has a minor downside of having to manually set the counter to track a specific number of incidents. Another Step-based way would be to make the counter track all the "non-visited" Steps within each observation Task, by adding single "Tracker Task" after each opened "Info Step" and therefore not requiring manual counting by the content designer creating such flow.

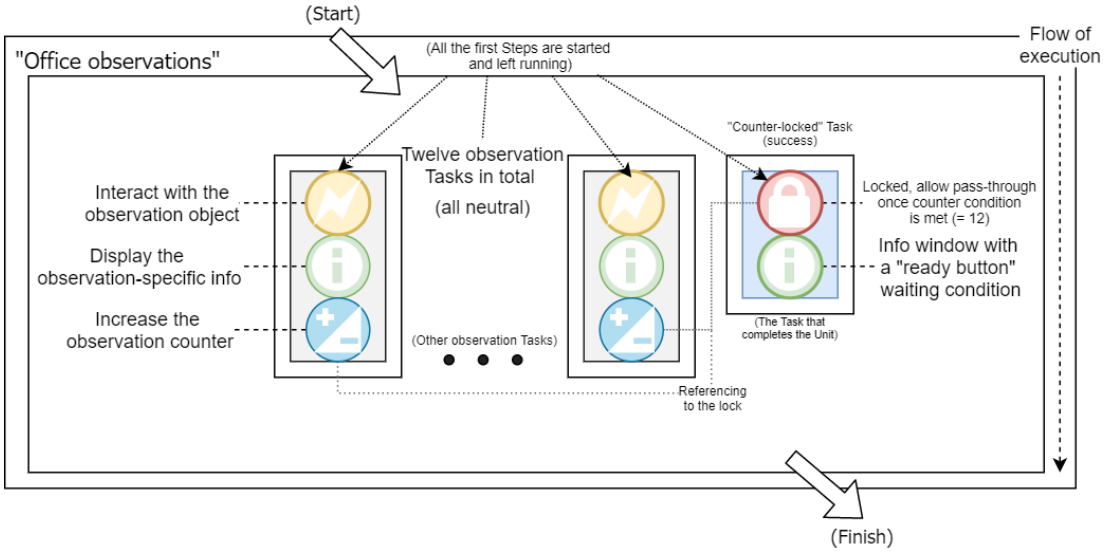

Figure 6.13: A "minimum viable flow" based on the script. When the flow starts the player is "in" thirteen Steps – twelve observation interaction Steps and one "Lock Step" halting the flow through the only "success" resulting Task. After completing all the neutral Tasks that manipulate the inner counter of the "Lock Step", the lock is released and the player can move forward.

In this example the observation Tasks were identical in their Step content, but there is no reason why it should always be that way: We could make different observations to yield different outcomes, such as showing animations of fixing the deficits. This example was still "linear" in a sense there would be no looping points (and the player only experiences the whole scenario once) or branching that would make player experience a different *set of outcomes* based on their actions. In the final example, we will demonstrate how to create flow structures that allow the flow to branch and loop.

6.2.3 Loop-forming case example – Safe movement

The previous cases were fairly linear in their nature. Considering there is also need for structures that support branching of the flow, we need to consider cases where the flow has multiple higher level states for the player to visit and mechanics for guiding the flow between these points. For demonstrating this, we depict a scenario where the player is taught about safe movement and the player has an option to choose their route not only in the sense of selecting the route, but also having control over the flow.

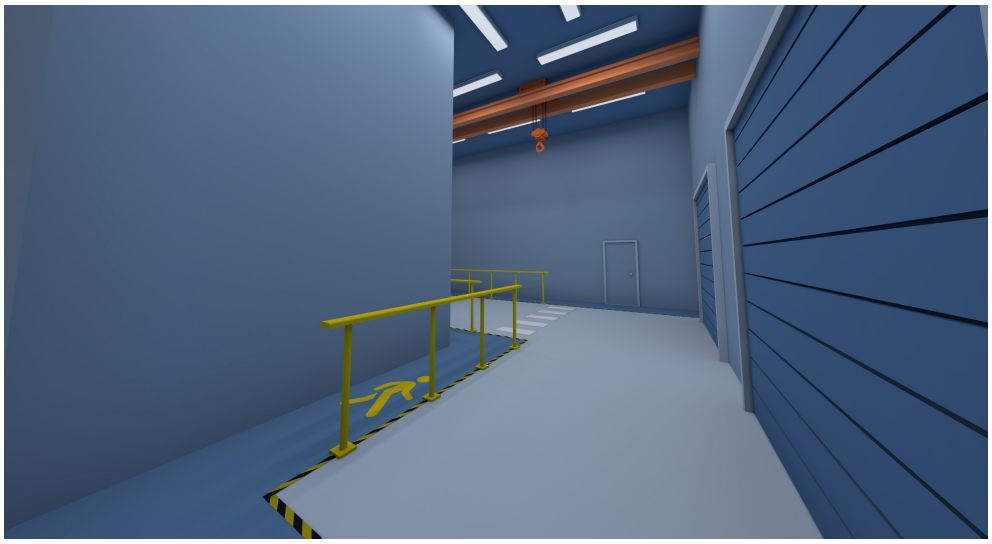

Figure 6.14: The player is presented a "two-way branch" that allows the actions of the player to determine which Unit is run next. The selection is made by tracking the player movement: Whether or not they walk on the walkway (on the left) or the driveway (the wide part on the right).

In this scenario, the player has an option to select between two possible routes: A safe walkway, designated for pedestrian movement, and a hazardous direct route that goes across a driveway with potential traffic. If the player chooses the wrong route, they are to be alerted and asked to return to the starting spot and try choosing again. This is repeated until the player chooses the correct route. The scenery visible to the player for this scenario can be seen in Figure 6.14.

The script could be described as following:

> "The player is **displayed an info window** giving the player a goal to move to the other side of the hallway space divided by a driveway. **The player can walk on the walkway section** and if done so, **displayed a congratulary info window**. The player also has an option to walk on the driveway but **an alerting info window is displayed** and **the area is highligted** if they do so, asking them to walk back to the starting spot that is also **highlighted** distinguishably **until they walk back on it**, **removing both highlights**. The virtual space is presented in a way that there is no room for walking further and a railing divides the walkway from the driveway making walking back to the starting spot the only option at that point. **Once the player reaches back to the**

starting point, **the first info window is presented again** with the same instructions. The process is repeated until the player finally select the safe route.

The script poses a clear branching and looping structure: The player can keep walking back and forth the starting and driveway areas without restriction but once they select the correct route, the flow is led forward. The Steps are the same as previously introduced in the first two examples for displaying info windows, waiting for the player to walk on an area, and highlighting areas. We could remove highlighting and close info windows using a dedicated Step but since we can model this functionality into different Units that make the manager reset all functionality within the previous Unit, there is no need to use a Step for this. Based on this structure, we can depict the flow like seen in Figure 6.15.

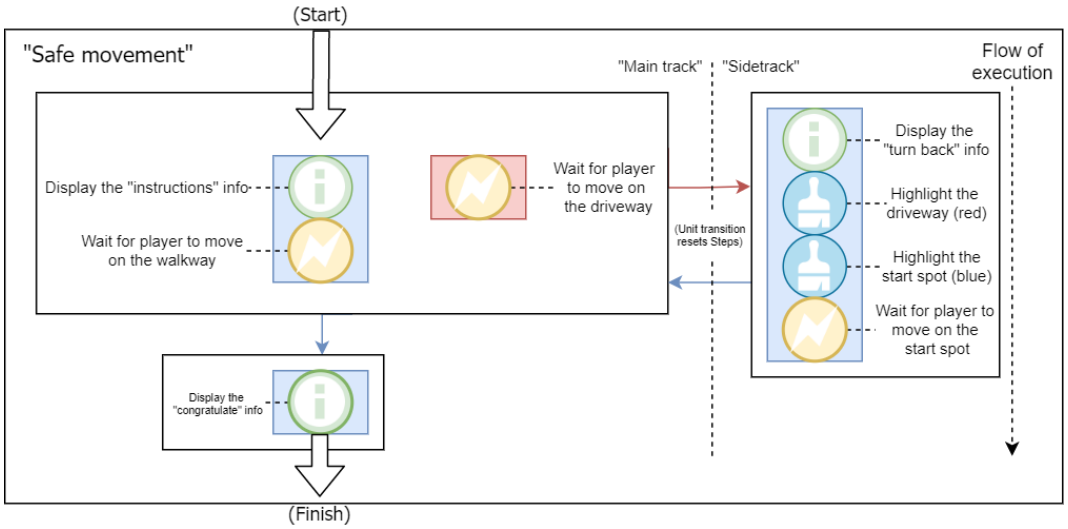

Figure 6.15: The VUTS structure based on the driveway safety script. The player can loop between the starting Unit (choosing the route) and the wrong Unit (stepping on the driveway).

The full structure here has three Units in total, one for depicting different "states" the player can be in within this scenario. The player starts in the Unit that presents the two possible branches in the form of Tasks. The first task contains one "Info Step" presenting the initial instructions and is followed by an "Interact Step" that waits for the player to Step on the walkway[21]. The second task contains only an

[21] As the Steps are to be run "at the same time", it doesn't matter where to put the "Info Step" as long as it is placed to be executed before the "Interact Steps" that halt the execution.

“Interact Step” necessary for tracking the player walking on the driveway. The first Task is marked by its exit result to be considered as “correct”, while the second Task is “incorrect”. By these distinctions, the starting Unit is set to point at the “success Unit” and “failure Unit” respectively. The “failure Unit” – the one containing the Steps that guide the player back – is set to point back to the starting Unit by its sole Task set to “correct”, marked as the “success Unit”. The “Interact Step” being the first Step within the Task halts the execution so that Task doesn’t complete “on its own” and allows for the player to guide the flow. Completing this Task will lead the player back to the starting Unit.

When the player chooses to move on the walkway the “correct” Task guides the flow to the “success Unit” with the purpose of showing the congratuling “Info Step” and – in this simplified example – wait for the player to press the “ready button” of the displayed window to completele the whole flow.

We have now constructed three different scenarios using VUTS structure and can carry out the evaluation based on its performance agaist the requirements.

7 Evaluation of the VUTS method

We have shown how the artifact can be used to model real-life training logic. Based on this observation we can evaluate the artifact to check its suitability against our defined criteria. In order to evaluate our approach, we go through each requirement and compare the seen capabilities to the requirement. Once the requirements are evaluated, we can consider answering the research questions.

7.1 Satisfying the requirements

We will evaluate each requirement by their order presented in the Chapter 3 based on the performance of the artifact to produce the content within the developing platform. We cover the three categories of requirements, considering our substance, set by the pedagogical needs of The Finnish Institute of Occupational Health, our architectural, defined by the development team, and the content design, guided by the use of the artifact to produce training content.

S_1**: Learning reproducibility** The flow in our examples can be built without any chance-depending elements. The path of the flow is either completely linear and can therefore be completed by following a clear path from the starting point to the finish by completing all the Steps in between. In other cases, where branching can occur, we can guide the player back to the point before the branching occurred as was seen in the case example 6.2.3 and make sure the player doesn't miss the "pedagogical path".

S_2**: Player-driven flow progression** The events occuring within the flow can be made depending on the player interacting with the environment and give player control of when to move forward within the flow. This is especially relevant on Unit-level, as player-controlled interaction event can be made the last Step of the Tasks leading the flow to other Units. This ensures the player can have, when wanted, control over when to move forward within the flow. Each Step can be implemented to have some kind of player-related waiting mechanism to make the flow *completely* player controlled if this kind of behaviour is required.

S_3: **Training through a distinct path** As we want the path to have a "red thread" that the player can easily follow, we can make the Unit structure simple by presenting "correct" and "incorrect" branching by Units, instead of more general branching format. By keeping the branching structure simple, we can make the player informed about them either going forward by doing things correctly, or explicitly telling them when their actions are not desirable and guide them back on track.

S_4: **Observable parallelism** As we learned from Tasks, and demonstrated by the case example in 6.2.2, simultaneous actions (in form of parallel Steps) offer ways to make multiple simultaneous actions and event-tracking possible. A Unit containing multiple Tasks can be made to contain many event-based Steps to allow player to choose between different interactions at the same time.

S_5: **Interactionality** The low-level implementation of different Steps allows for the implementing interactionality, such as the way we used in the form of an "Interact Step". This kind of Step can wait for the player to complete an interaction and pass the flow forward once the interaction has been completed. The implementor has the ultimate control over how such interactionality, if present, is implemented and used.

S_6: **Minimal learning curve** Implementing the Steps once and reusing their functionality minimizes the need for introducing different functionality to the player. Each re-used Step can be made to utilize the same exact functionality and make sure the player is not introduced new mechanics to learn unless wanted. Changing the Step-specific parameters can make the Steps function differently, but keeping the Step implementation light we can keep the need for learning the mechanics minimal. The simplified structure, as mentioned related to S_3, can further help the player to comprehend the flow and the events in it.

S_7: **Story-drivenness** We have seen how a script can be converted into a VUTS structure. Presented story-based events can be depicted as Steps at their lowest level or as Units when the high-level transitions are more meaningful. The structure allows for freely building an interoperable system of both low-level and high-level events. Each flow has a starting point and at least one ending point. In between actions can be depicted freely and make the player experience the "story" through these actions and points.

S_8: **Analytics** As each component of VUTS has a clear start and an ending point, we can safely conclude that meaningful analytics can be collected regarding their timestamps on when they are started (or "visited") and when they are completed. Each component can further be implemented to trigger events based on the actions they perform on a fine-grained level to get more transparency on their inner workings and get information as exact as required depending on the use case.

A_1: **Simplicity** By using a custom-made, domain-specific solution, with a constant depth, the architectural complexity of the artifact is set to a minimum that best serves our domain-specific needs. The component levels cater the substance needs as-is without introducing additional complexity needless for our usage. Implementing different Steps allows for creating different behaviour based on the content needs and based on our examples satisfies the three scenarios listed with varying content.

A_2: **Expressiveness** Our content examples depicting three different usage patterns were successfully modeled with the approach. The Step implementing pattern allows for further extending the system by introducing new functionality while keeping the structure depth constant. By combining the different flow patterns presented in the Sub-chapter 6.2, more advanced structures can be modeled.

A_3: **Modularity** As implemented Steps operate independently, we can create different scenarios by altering the structure (in a similar fashion as in FBP and Statecharts) without chaging the whole structure. Adding or removing Steps or otherwise changing the structure doesn't affect the other Steps unless they explicitly reference each other and is something that can be left unimplemented should more modularity be required. As each Unit is freely interconnectable, we can change the route of the flow just by changing the cross-Unit references.

A_4: **Reusability** The order and the type of the Steps, as well as the parameters of them, can freely be altered for catering different scenarios, as shown by the examples cases. Once implemented, a Step can be used in any content by having its optional values tweaked. As shown by the case examples, Steps created for one scenario may cater the needs of other scenarios as well.

A_5: **Customizability** As mentioned, through parametrization of Step components we can change the functionality of any actions within the flow. If the changes require higher level re-ordering of the scripted events, we can re-order the Unit structure accordingly.

A_6: **Integration** We have shown that integrating the solution to Unity3D is possible and our case examples were constructed utilizing it as the development platform. As VUTS is an abstraction without language-specific or environment-specific features, it can be implemented in any similar environment that supports depicting such flows in some visual sense, such as in a graphical development suite with scripting possibilities.

A_7: **Scalability** As we can reuse, customize and develop the content modularly, after developing the base implementation for VUTS, we can save time per each new training scenario created, as each new piece of content can utilize pre-existing tools and implementations readily available. As the system sets a limited scope for its functionality, creating new content and possibly implementing new functionality for it require less testing, as they are directly based on this "sub-system" – the approach – already implemented on top of a more expressive language or a development suite.

C_1: **Guidelining** Related to scalability and simplicity, the approach defines a limited scope of functionality that can be reused to create new content. As we have the Step-based implementations readily available, there is a clear advantage using them instead of developing new functionality. Sharing this similar functionality allows for creating content functionally similar in its nature and thus make it easier for the player to learn and follow. Additionally, as we lock our structure down to four levels of depth, we ultimately prohibit the designer for going for more complex structures that would be harder for the player to follow as flow structures.

C_2: **Visualizability of the structure** As shown in Sub-chapters 5.1.1 and 5.2, the approach provides a clear four-level visualizable flow for stakeholders to inspect. The format also caters both non-technical and technical people, as the visualized structure can be implemented as-is, only requiring the possible paramers to be shown if the implementation level is used. By utilizing color coding and minimizing the edge usage, we can improve the readability. When depicting the content on higher

levels, the need of those parameters is waived, and the flow can be designed visually with only minor textual cues included. As VUTS provides a one-to-one mapping of the training logic and its visual representation, the script can also be converted back into its textual logic form.

C_3: **Cognitive ergonomics** Based on the previously met requirements and being shown the case examples, the approach offering a simple, multi-level structure for depicting content can ease the mental work required for designing content by our other requirements. As there is no needs to guess how a content should be modeled in terms of transitions, the approach eliminates the ambiguity and thus makes designing the content more straightforward, allowing more time to be spent on modeling the content.

C_4: **Bookkeeping** By starting the modeling from the top, we can iterate through the four levels of depth and mark clear points how much time each level and its sub-content has taken per content designed. The approach allows for tracking the time per transition, per action, per state or per content manner, making different measuring techniques possible.

7.2 Experiences from using VUTS

The Virtuario^{TM} team currently utilizes VUTS in all our virtual reality content design and implementation. By our own estimations, the development time saved by utilizing the approach has made our large-scale content creation possible and sustainable. Our yet small team has been able to design and implement over a dozen different pieces of training or promotional content between summer 2018 when VUTS was implemented and late 2019, at the time of writing this thesis. The more we have used and invested in building VUTS, the more versatile its use has become by having more Steps at hand and also by re-using the general structures.

Our clients have been utilizing the first versions of the training platform to train hundreds of their workers with training content tailored to them. By using our approach, we were able to implement these VR training scenarios of the client-driven content scripts without any need of simplifying or otherwise degrading the original expressive quality of the scripts, yet maintaining fast iteration cycles based on client feedback.

The use of VUTS method has therefore fully supported our design and implementation of all the required client projects and our experimental ideas we have tried along the way from promotional to purely entertaining content. We have yet to face any disadvantages of completely relying on using the approach – other than the initial resource investment for developing it. We have also seen a drastic drop in content logic related bugs after putting this approach to use; this can be due to sharing the same of content logic across all the different implemented training scenarios, but further research is required. The difference is nevertheless considerable when compared to our older methods of creating new – non-interoperable – functionality for each training content separately, adding (needless) overall architectural complexity by each new piece of content created.

Once a virtual scene without any funcionality is constructed, adding VUTS logic on top of it to building content that surpasses a minimum-viable-product is a matter of *hours to a workday*, down from *days to weeks*. The initial investment in building the approach is currently well paying off: New content can finally be created at a highly competitive rate.

7.3 Revisiting the research questions

We can conclude that all the requirements are being met by the approach – the VUTS method. We have defined the necessary requirements for the artifacts and have shown how they are satisfied by using the approch to model content relevant to our needs. We can therefore derive the answers to the research questions from the above points. We first consider the first research question

RQ1: What benefits can a domain-specific approach offer over using general flow and state-based approaches?

We have shown that the VUTS method encourages reusing once created components in different content creating scenarios – similarly to FBP and Statecharts. By narrowing the scope with our approach, we were able to create a solution that caters our own specific content creation style, yet include the presented three main patterns of the flow structure. As both the low-level implemented components and high-level transitional structure are made to match our content model needs, less time can be spent programming functionality and structures to support various content than if they were implemented independently or if an exceedingly general approach was

used that wouldn't support our simplicity (A_1) and guidelining (C_1) requirements. More general, non-domain-specific approaches have functionality that goes beyond our expressive scope for defining VR content logic concerning our flow modeling needs. Most crucially, features such as unlimited hierarchical depth and information packets would not support our substance or architectural needs by providing abstractions that clash with our substance requirements related to player-driven experience (S_2) and the mentioned simplicity (A_1). As the general approaches offer extensive expressive capabilities not designed for depicting scenarios exactly like ours, the built-in guidelining factor is absent. We can now consider our second research question

RQ2: What benefits can we achieve by depicting VR content logic as a hierarchical, human-readable, and visual flow?

Our approach poses a tangible form that allows depicting the script in a mainly non-textual format that simultaneously doubles as an implementable flow structure. Each flow can be presented in different ways and the abstraction levels offer different perspective to the same content in addition to catering different stakeholders. We can convert the textual training logic into a visual flow and retrieve its textual format back from the visual representation – according to the visualizability requirement (C_2). We have also seen how the secondary notation readability criteria apply to our approach. Our approach incorporates syntax-highligting as colored elements for better catering novice modelers in their interpretation accuracy and by minimizing the need of transitional arrows and simplifying the remaining arrows for providing more accurate interpretation in general.

Finally, we can consider our research objective of being able implement the approach to satisfy our needs for VR content creation. We have successfully built a corporate-grade, commercial implementation for being used in Virtuario™ that utilizes the VUTS method implemented in C#. Our implementation, described in Sub-chapter 6.1, and case examples show that integrating our approach to a game development platform – Unity3D in our case – is possible and yields usable results, as content creation is possible by our requirements.

We can conclude the VUTS method satisfies our research questions and its Unity3D-specific implementation reaches the research goal and therefore meets our needs.

8 Discussion

Our approach meets the needs set by our *own* requirements: It can be utilized by the development team, aids the content design, and meets the standards set by The Finnish Institute of Occupational Health to be used in Virtuario™. We can conclude that our approach offers value to us in our product development but required the initial investment of being implemented and learnt to be used effectively.

The potential for more general use is still a question mark. We cannot conclude whether or not the VUTS method offers suitable tools for *other* content creators to be used effectively by only inspecting the VUTS method through our own requirements. This thesis and the design for this approach was primarily led by the needs of The Finnish Institute of Occupational Health. The requirements we listed may apply to other content creators with similar interests but this is not something we can claim. In any case, we are more than happy to have made our approach available to help the content design and creation of other parties – possibly finding even more use in it than we do.

Our constructed approach was created with the specific needs and hierarchy depth in mind but could further be developed to answer to potentially changing requirements: The future challenges include how to move the player between different VUTS structures – and possibly include even a higher level component – by taking into account the choices made within a single structure, for example. The hierarchy, by its separate layers, offers potential for further extension to increase its flexibility and supported levels of expressiveness. The visualizability of the whole structure including the Steps and their parameters is still something that will see further development to improve the "flipchart-drawability" of VUTS concerning the obscurity of free-hand drawing. The secondary (and primary) notation is a field requiring more in-depth research, and our use of colors by their arbitrary shades, for example, in depicting VUTS has been liberal and not justified by any other than purely subjective factors. Our approach could also benefit from having clearer visual cues for branching, and studies like Figl et al. (2013) suggest that distinguishing branching points can provide additional visual clarity. One additional apparent culprit on depicting the graph so far has been the use of Step icons that introduce complexity to interpreting them and being arguably seemingly difficult for drawing by hand. Finding better substitutes for the Step icons, such as extending the color-coding approach, as well as studying or experimenting with different shapes for easy free-hand drawing poses potential for further enhancing both the drawability and readability.

Even finding ways of incorporating textual labels in an easy-to-read, non-cluttering way might offer a solution to this challenge.

There are similar graph-like, visual solutions available that focus on both state-like and flow-like structures – suitable for exploration and inspiration for further development. The possible comparison points include Coloured Petri Nets that allows visually depicting a vast array of different processes, such as validating and simulating systems, as complex structures that also support concurrency and branching (Jensen, 1997), and DRAKON language (and its DRAKON editor) (DRAKON, 2019) that aims to optimize the readability of its charts that serve as both modeling and programming languages. One implementation-specific way of creating content is Blueprints Visual Scripting (Unreal Engine, 2019) that enables visually programming content logic in high customizability within Unreal Engine game developing platform. Other domain-specific, VR training related approaches with different specifications and requirements, but promoting similar flow depiction also exist (Mollet and Arnaldi, 2006). The two reviewed approaches introduced as comparison points merely scratch the surface of similar techniques created for solving problems like ours but display the the differences in expressiveness catering to a wider or a different scope compared to a domain-specific approach that can pinpoint the scope around the requirements.

Extending our scope in future work can allow for tackling the mentioned issues by refining the approach and ultimately formalizing it in a more exact way than our limited pragmatical review of it in this thesis did. Defining the expressive capabilities and limitations of our approach in a formal way would be a topic suitable for a doctoral thesis: The approach will presumably still mature over the course of upcoming projects developed utilizing it to truly test its maturity.

As the requirements can be fluid, there will potentially be need for adding additional layers of abstraction to our solution. This approach has shown much potential in its seeming simplicity and because of it has gained its place as the basis for Virtuario™ content creation. By our estimates it will remain so for the lifetime of the project while surely seeing continuous improvements in its Virtuario™ related implementation, its hierarchical format, and its notation.

The interests of The Finnish Institute of Occupational Health have been shaping the approach and its requirements through its use in Virtuario™ and governing the content of this thesis. Taking a more scientific perspective over corporal interests is what we would gratefully do in our future research of VUTS and its potential.

Acknowledgments

I would like to thank the supervisor of this thesis, on behalf of The Finnish Institute of Occupational Health, a research engineer and Virtuario™ project manager Kristian Lukander for the mental sparring and challenging my ideas. Additional acknowledgements are due to the other members of Virtuario™ team: Jose Uusitalo for processing the VUTS analytics data, Elias Salonen for providing the Step icons, and Vilma Penkari for gathering and analyzing the usability data regarding the VUTS usage from the player's perspective.

References

Bandura, A. and Walters, R. H. (1977). *Social learning theory*, volume 1. Prentice-hall Englewood Cliffs, NJ.

Barfield, W. and Furness, T. A. (1995). *Virtual environments and advanced interface design*. Oxford University Press on Demand.

Barnes, S. (2016). Understanding virtual reality in marketing: Nature, implications and potential. *Implications and Potential (November 3, 2016)*.

Becker, J., Rosemann, M., and Von Uthmann, C. (2000). Guidelines of business process modeling. In *Business process management*, pages 30–49. Springer.

Bowman, D. A. and Hodges, L. F. (1997). An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. *SI3D*, 97:35–38.

Brown, A. and Green, T. (2016). Virtual reality: Low-cost tools and resources for the classroom. *TechTrends*, 60(5):517–519.

Cipresso, P., Chicchi Giglioli, I. A., Alcañiz Raya, M., and Riva, G. (2018). The past, present, and future of virtual and augmented reality research: a network and cluster analysis of the literature. *Frontiers in psychology*, 9:2086.

Cuervo, E., Chintalapudi, K., and Kotaru, M. (2018). Creating the perfect illusion: What will it take to create life-like virtual reality headsets? In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, pages 7–12. ACM.

Dockx, K., Bekkers, E. M., Van den Bergh, V., Ginis, P., Rochester, L., Hausdorff, J. M., Mirelman, A., and Nieuwboer, A. (2016). Virtual reality for rehabilitation in parkinson's disease. *Cochrane Database of Systematic Reviews*, (12).

DRAKON (2019). Drakon editor. `http://drakon-editor.sourceforge.net/`.

Figl, K., Recker, J., and Mendling, J. (2013). A study on the effects of routing symbol design on process model comprehension. *Decision Support Systems*, 54(2):1104–1118.

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174.

Ha, T.-H. and Sonnad, N. (2017). How do you draw a circle? we analyzed 100,000 drawings to show how culture shapes our instincts. `https://qz.com/994486/the-way-you-draw-circles-says-a-lot-about-you/`.

Hamari, J., Shernoff, D. J., Rowe, E., Coller, B., Asbell-Clarke, J., and Edwards, T. (2016). Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning. *Computers in human behavior*, 54:170–179.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274.

Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram, and Sudha (2004). Design science in information systems research. *Management Information Systems Quarterly*, 28:75–.

Hopcroft, J. E. (2008). *Introduction to automata theory, languages, and computation*. Pearson Education India.

HTC (2019). Vive pro - the professional-grade vr headset. `https://www.vive.com/us/product/vive-pro/`.

Huang, C.-C. and Kusiak, A. (1998). Modularity in design of products and systems. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 28(1):66–77.

Huotari, K. and Hamari, J. (2012). Defining gamification: a service marketing perspective. In *Proceeding of the 16th international academic MindTrek conference*, pages 17–22. ACM.

Jensen, K. (1997). A brief introduction to coloured petri nets. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 203–208. Springer.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1, Fundamental Algorithms, Third Edition*. Addison-Wesley.

Kwon, C. (2019). Verification of the possibility and effectiveness of experiential learning using hmd-based immersive vr technologies. *Virtual Reality*, 23(1):101–118.

Laplante, P. A. (2007). *What every engineer should know about software engineering*. CRC Press.

LaViola Jr, J. J. (2000). A discussion of cybersickness in virtual environments. *ACM Sigchi Bulletin*, 32(1):47–56.

Linehan, C., Bellord, G., Kirman, B., Morford, Z. H., and Roche, B. (2014). Learning curves: analysing pace and challenge in four successful puzzle games. In *Proceedings of the first ACM SIGCHI annual symposium on Computer-human interaction in play*, pages 181–190. ACM.

Mazuryk, T. and Gervautz, M. (1996). Virtual reality-history, applications, technology and future.

Mendling, J., Reijers, H. A., and Cardoso, J. (2007). What makes process models understandable? In *International Conference on Business Process Management*, pages 48–63. Springer.

Mestre, D., Fuchs, P., Berthoz, A., and Vercher, J. (2006). Immersion et présence. *Le traité de la réalité virtuelle. Paris: Ecole des Mines de Paris*, pages 309–38.

Mettler, T., Eurich, M., and Winter, R. (2014). On the use of experiments in design science research: A proposition of an evaluation framework. *CAIS*, 34:10.

Mili, H., Mili, F., and Mili, A. (1995). Reusing software: Issues and research directions. *IEEE transactions on Software Engineering*, 21(6):528–562.

Mollet, N. and Arnaldi, B. (2006). Storytelling in virtual reality for training. In *International Conference on Technologies for E-Learning and Digital Entertainment*, pages 334–347. Springer.

Morrison, J. P. (2010). *Flow-Based Programming: A new approach to application development - 2nd edition*. CreateSpace.

Nykänen, M., Puro, V., Tiikkaja, M., Kannisto, H., Lantto, E., Simpura, F., Uusitalo, J., Lukander, K., Räsänen, T., and Teperi, A. M. (2019). Modern safety learning for construction industry – mosac (2018-2020). `https://www.ttl.fi/en/research-and-development-projects/mosac/`.

Oculus (2019). Oculus for business. `https://www.oculusforbusiness.com/`.

OMER, M., HEWITT, S., MOSLEH, M. H., MARGETTS, L., and PARWAIZ, M. (2018). Performance evaluation of bridges using virtual reality. In *Proceedings of the 6th European Conference on Computational Mechanics (ECCM 6) and 7th European Conference on Computational Fluid Dynamics (ECFD 7) Glasgow, UK*.

Parsons, S. and Cobb, S. (2011). State-of-the-art of virtual reality technologies for children on the autism spectrum. *European Journal of Special Needs Education*, 26(3):355–366.

Purchase, H. (1997). Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer.

Reijers, H. A., Freytag, T., Mendling, J., and Eckleder, A. (2011). Syntax highlighting in business process models. *Decision Support Systems*, 51(3):339–349.

Sacks, R., Perlman, A., and Barak, R. (2013). Construction safety training using immersive virtual reality. *Construction Management and Economics*, 31(9):1005–1017.

Schrepfer, M., Wolf, J., Mendling, J., and Reijers, H. A. (2009). The impact of secondary notation on process model understanding. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 161–175. Springer.

Silva, R., Oliveira, J. C., and Giraldi, G. A. (2003). Introduction to augmented reality. *National laboratory for scientific computation*, 11.

Unity Technologies (2019a). Unity - manual: Unity user manual (2019.2). `https://docs.unity3d.com/Manual/`.

Unity Technologies (2019b). Unity - scripting api. `https://docs.unity3d.com/ScriptReference`.

Unreal Engine (2019). Blueprints visual scripting. `https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html`.

Vaishnavi, V. and Kuechler, B. (2004). Design science research in information systems.

Van Wyk, E. and De Villiers, R. (2009). Virtual reality training applications for the mining industry. In *Proceedings of the 6th international conference on computer graphics, virtual reality, visualisation and interaction in Africa*, pages 53–63. ACM.