

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Procedural content creation in VR

Bruno Carvalho



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Daniel Mendes

Second Supervisor: Prof. António Coelho

March 11, 2022





# **Procedural content creation in VR**

**Bruno Carvalho**

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Jorge Silva

External Examiner: Prof. Maria Santos

Supervisor: Prof. Daniel Mendes

March 11, 2022



# Abstract

3D content creation for virtual worlds is a difficult task, requiring specialized tools based on a WIMP interface for modelling, composition and animation. Natural interaction systems for modelling in augmented or virtual reality are currently being developed and studied, making use of pens, handheld controllers, voice commands, tracked hand gestures like pinching, tapping and dragging mid-air, etc.

We propose a content creation approach for virtual reality, placing a focus on making procedural content generation (PCG) intuitive and generalizable. Our approach is to start with a library of 3D assets, with which the user populates an initially empty world by placing and replicating objects individually. The user can then construct procedural rules to automate this process on the fly, creating abstract entities that behave like a block of objects while still being treated and manipulated like other singleton objects.

To this end, we design a rule system for procedural content generation adequate for virtual reality, including nested object replication, relative placement and spacing, and randomized selection. We then design and prototype a natural interaction model for virtual reality suited to this rule system and based on natural interaction techniques with input from two virtual reality controllers. A prototype of this interaction model is built, and finally, a formal user evaluation is conducted to assess its viability and identify avenues for improvement and future work.

**Keywords:** 3D modelling, procedural generation, virtual reality

**ACM Area:** CCS → Human-centered computing → Interaction Design → Interaction design process and methods → Interface design prototyping



# Resumo

Criação de conteúdo 3D para mundos virtuais é uma tarefa complexa envolvendo ferramentas de software especializadas baseadas numa interface WIMP para modelação, composição e animação. Sistemas de interação natural para modelação em realidade virtual ou aumentada estão neste momento a ser estudados e desenvolvidos, dando uso a canetas digitais, controladores de VR ou de jogos, comandos de voz e gestos de mão como apertar, tocar, segurar e arrastar, etc.

Nós propomos uma abordagem para criação de conteúdo em realidade virtual que coloca um foco em realizar geração procedimental de conteúdo de forma intuitiva e generalizável. Começando com uma biblioteca de objetos 3D que o utilizador emprega para popular um mundo virtual inicialmente vazio colocando os objetos um a um, um conjunto de regras procedimentais podem ser definidas na hora, capazes de criar entidades abstratas que se comportam como um conjunto de objetos e podem ser manipulados como objetos individuais.

Neste sentido desenvolvemos um sistema de regras para geração procedimental adequado a realidade virtual e incluindo replicação automática de vários objetos, colocação e espaçamento relativo de objetos, e seleção aleatória. Depois desenhamos um modelo de interação natural para este sistema de regras baseado em manipulação bimanual com controladores de realidade virtual. Um protótipo desta abordagem é construído e finalmente uma avaliação formal é realizada para avaliar a sua viabilidade e identificar possíveis problemas e melhorias.

**Palavras-chave:** modelação 3D, geração procedimental, realidade virtual



# Acknowledgements

My sincere thanks to Professor Daniel Mendes. This work would have been impossible without his guidance and expertise.

I would also like to thank Professor António Coelho and fellow researcher Guilherme Amaro for various advice and contribution from their own fields of study.

This project was financed by ERDF - European Regional Development Fund through Programa Operacional para a Competitividade e Internacionalização - COMPETE 2020 within the scope of Portugal 2020 Partnership Agreement, and by national funds through the portuguese financing agency FCT - Fundação para a Ciência e a Tecnologia within the PAINTER project with reference POCI-01-0145-FEDER-030740 - PTDC/CCI-COM/30740/2017.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context and Motivation . . . . .	1
1.2	Objectives . . . . .	3
1.3	Document Structure . . . . .	3
<b>2</b>	<b>Related Work</b>	<b>5</b>
2.1	Modelling in virtual or augmented environments . . . . .	5
2.1.1	3D Object selection and manipulation . . . . .	6
2.1.2	3D object searching . . . . .	7
2.1.3	Approaches to content creation and interaction . . . . .	7
2.2	Procedural content generation . . . . .	11
2.2.1	Procedural grammars for cityscapes . . . . .	11
2.3	Discussion . . . . .	12
<b>3</b>	<b>Procedural VR</b>	<b>17</b>
3.1	Vision and usability goals . . . . .	17
3.2	High-level example overview . . . . .	18
3.3	Objects and Operations . . . . .	20
3.3.1	Editing objects . . . . .	22
3.3.2	Operations . . . . .	24
3.3.3	Clone by reference . . . . .	25
<b>4</b>	<b>Prototype Development</b>	<b>29</b>
4.1	User Interface and interaction . . . . .	29
4.1.1	Basic interaction . . . . .	30
4.1.2	Scene composition . . . . .	32
4.1.3	Interaction modes . . . . .	33
4.2	Editing objects . . . . .	35
4.3	Undo system . . . . .	39
4.4	Feedback system . . . . .	40
4.5	Miscellaneous . . . . .	42
4.5.1	Grab constraints . . . . .	42
4.5.2	Edit augmentations . . . . .	44
4.5.3	Global scaling . . . . .	44
4.6	Implementation architecture . . . . .	44
4.7	Modeling patterns . . . . .	46
4.7.1	Sample workflows . . . . .	48

<b>5</b>	<b>User Evaluation</b>	<b>51</b>
5.1	Testing methodology . . . . .	51
5.1.1	Testing environment . . . . .	51
5.1.2	Tasks and scenarios . . . . .	52
5.1.3	Tutorial . . . . .	55
5.1.4	Questionnaire . . . . .	57
5.2	Results . . . . .	57
5.2.1	Participants . . . . .	57
5.2.2	Task performance . . . . .	58
5.2.3	Usability and Task Load . . . . .	58
5.2.4	Conclusions . . . . .	61
<b>6</b>	<b>Conclusions</b>	<b>63</b>
6.1	Future work and limitations . . . . .	63
	<b>References</b>	<b>65</b>

# List of Figures

2.1	Put-That-There and Mockup Builder . . . . .	6
2.2	CSG operations in Mendes et al. [12]. . . . .	8
2.3	Extruding operations in Teddy [10]. . . . .	9
2.4	Glove and ball modelling setup in Worlds with Strokes [3]. . . . .	9
2.5	Creating new models in DesignAR . . . . .	10
2.6	Creating a new building from multiple sketches [13]. . . . .	11
2.7	A basic L-system producing a self-similar polygonal fractal [4]. . . . .	12
2.8	Example split grammar rules for generating a building facade [13]. . . . .	12
2.9	Taxonomy of interaction models and content creation approaches. . . . .	13
3.1	Example object trees . . . . .	23
4.1	Grab interaction . . . . .	30
4.2	Floating menu . . . . .	31
4.3	Teleportation arc . . . . .	31
4.4	Controller button assignments . . . . .	32
4.5	Scene object tree . . . . .	33
4.6	Slave grab example . . . . .	36
4.7	Mover editing example . . . . .	37
4.8	Rotator editing example . . . . .	37
4.9	Tiling editing example . . . . .	38
4.10	Priority laser collision . . . . .	39
4.11	Feedback examples . . . . .	41
4.12	Live feed example . . . . .	41
4.13	Control flow overview . . . . .	46
4.14	Matrix tiling pattern . . . . .	49
4.15	Randomized chair pop sample workflow . . . . .	50
5.1	Restricted baseline menu. . . . .	52
5.2	Computer room test scenario . . . . .	53
5.3	Town center test scenario . . . . .	54
5.4	Tutorial scenario . . . . .	56
5.5	Demographics of volunteers. . . . .	57
5.6	Primary performance metrics . . . . .	59
5.7	SUS usability results . . . . .	60
5.8	TLX Task Load results . . . . .	60



# List of Tables

2.1	Classification of related modelling works. . . . .	15
3.1	Object vs operation table . . . . .	27
4.1	Interaction modes . . . . .	34
4.2	Edit operations through edit mode menu buttons . . . . .	35
4.3	Feedback system color codes . . . . .	42
4.4	User controllable grab constraints. . . . .	43



# Chapter 1

## Introduction

3D content creation is a complex and time-consuming task usually realized on a desktop environment, in editors like Maya or Blender with WIMP-style interfaces (windows, icons, menus, pointers). It is used to create digital content like virtual worlds for television and cinema, human and animal models for biology and medicine, machine models for engineering and the manufacturing industry, digital characters and scenarios for video games, etc. Each domain has a different purpose for 3D content creation and therefore requires a different set of tools or approaches.

### 1.1 Context and Motivation

Teams of artists require extensive knowledge in animation, modelling, composition, rendering and other areas to create the complex 3D models and scenes often seen in video games, advertisements, and film, and their software tools reflect this, being filled with functionality that is very valuable to multi-disciplinary teams but might be overwhelming for single artists and hobbyists.

The most commonly used content creation tools are traditional desktop applications that use WIMP interfaces and 2D displays (computer monitors). This approach has a few limitations. The first is that the input space is limited to the two-dimensional movement allowed by the mouse, making it difficult and awkward to realize certain kinds of 3D operations requiring 3 or 6 degrees of freedom, restricting the potential set of interaction techniques for the user. For example, this input mechanism does not allow the user to simultaneously and independently rotate and draw on a 3D object, an operation requiring an input tool with at least 6 degrees of freedom. The second limitation concerns the output space - the visualization of 3D scenes and models is made limited on a 2D display, and the user's perception of shape, depth and size would be improved if stereoscopic visualization was employed instead. These limitations can be addressed separately, e.g. it is possible to use a stereoscopic visualization tool like a head-mounted display (HMD) for visualization while maintaining traditional mouse and keyboard input.

When creating new content, artists often find that certain 3D models and scenes are composed of many repeated components with little to no variation. To decrease the burden of creating these kinds of models new approaches based on procedural content generation (PCG) were introduced. Instead of creating each such model directly, a system is designed that is capable of generating multiple entities automatically following a set of rules implemented by the artist. These rules can specify, for example, the scale and position of multiple generated objects, transformations applied to them, and the number of repetitions, often with low-level mesh randomization or transform randomization. However, the creation of these descriptive PCG rules is often more complex than the creation of each entity itself and require experience in visual programming, meaning that even experienced artists can have trouble using this approach at first.

Many users don't have experience with the desktop modelling tools previously mentioned nor programming skills for procedural content generation, yet they may wish to, say, create a simple scene with premade assets from third parties. For example, a hobbyist may be interested in assembling his home town using a city asset pack - with buildings, roads and vehicles - from the web, without requiring months of expertise training. Creating such a scene is a big endeavour, but conceptually it is simple enough to warrant research into more specialized and familiar tools. This is our primary motivation for this work, and we will propose an approach to content creation in virtual reality that can fill in this *niche* for novice users.

Creating such a modelling tool is made inherently difficult as we are in a virtual reality setting. The main advantage of doing so in a virtual environment is that we can simultaneously address both limitations we've spoken of earlier: it improves the perception of scale and shape through stereoscopic visualization and it allows for natural interaction models to arise, like those based on voice commands, manual gestures, controllers or other specialized tools with 3D tracking and an input space with 6 degrees of freedom.

Another use case that is orthogonal to this novice user *niche* is that of a **productivity tool for game development**. A particularly good example is the assembly of a *city or town center* from buildings, roads, cars and other related assets. Another good example is the assembly of a desk-centered building interior, like a *classroom* in a school or a general *computer room* in an office, which will be the overview sample in the next section.

Let us describe this second use case in detail. Imagine a game development studio creating a new game for desktop, console or virtual reality (or some other platform). It is an adventure and exploration game, set in a large world and with many important sites for the player to visit, and even more relatively unimportant and minor ones. These minor sites, assets, environments, or generally *models* could be anything like clearings in a forest, bridges over rivers, gardens and fountains and parks, office and residential buildings, various kinds building interiors (storage rooms, office rooms, hallways, bathrooms, bedrooms...), and much more. We consider also minor elements within a major site or event in the game, which will not get the player's attention often, but are needed to set the context and populate the environment, such as pedestrians and shops around roads in a car-driving game, trees and shrubs in a wild adventure game and so on.

Creating all these minor models by hand takes an amount of time from the creators that is



disproportionate to their importance. The creators would like to focus on more important aspects of their game that they and their players care more about. These minor models cannot simply be disregarded or discarded: the game would look unrealistic, rushed and unpolished without them.

## 1.2 Objectives

Our goal was thus to design a modelling system for virtual reality with a focus on procedural content generation capabilities and develop a prototype of this approach relying exclusively on natural interaction techniques. The system should still be approachable to users with little to no experience in modelling software, and as such, it is *not* a sole substitute for common feature-rich desktop applications. Overall, we aimed to:

- Design a set of simple procedural content generation rules and key voice commands for modelling in virtual reality.
- Design an interaction model, targeting a modelling tool in virtual reality with procedural content generation capabilities, based on input from voice commands and two-handed controller input.
- Prototype the designed approach and perform a formal evaluation with users.

Our final approach does not include voice commands. We envisioned these would be used for textual search of assets and objects, but this idea was discontinued during development in favour of other prototype features.

## 1.3 Document Structure

In chapter 2 we present related scientific work focused on the domains of 3D modelling in virtual environments, and cover also some procedural content generation techniques. In chapter 3 we present our proposed modelling approach and also a sample *workflow*. In chapter 4 we present the prototype developed that realizes this approach using only natural interaction techniques, including other various features and interesting modelling patterns. In chapter 5 we present the results of the user evaluation performed, and finally in 6 we conclude with a brief summary of the results, some limitations of our approach and potential future work.



## Chapter 2

# Related Work

In order to adequately contextualize our work we will first present a review of the literature surrounding the area of content creation and interaction in virtual or augmented environments, including an overview of the challenges we face. Afterwards we present and analyse some of the current VR modelling systems and research prototypes and compare them to our approach. Finally we will present a very brief overview of procedural content generation (PCG) methods and techniques, with a focus on those suitable for interactive use in a virtual setting.

### 2.1 Modelling in virtual or augmented environments

Systems for content creation in virtual environments have a different set of requirements than on desktop environments. Consider a scenario where a user wishes to model a building within a city scene. In a desktop editor like Maya, navigation around the scene is performed with one mouse button, and object selection with the other. Some CAD operations like translation, intersection and extrusion of models are achieved with guide arrows and overlays around the selected model, while more complex functions can be accessed through side or dropdown menus. These functions' parameters can also be set exactly, by filling in a value in a text box - for example, typing 1.4 in the scale field to *scale* an object by 40%. Furthermore, existing models can be added to the scene after being searched by traditional lookup with a file manager or text-based search engine.

In a virtual or augmented environment these tasks cannot be realized in the same way. In a virtual environment the input devices used usually include a handheld controller and a head-mounted display (HMD) with tracking capabilities, instead of a mouse, keyboard and screen. As such, modelling tools must use different techniques to achieve similar functionality. We now cover previous work on this area, and see how several authors have tried to tackle these challenges in an attempt to design more natural interaction systems for content creation in virtual environments. We will start with a few examples of general interaction approaches and then cover the challenges one by one.

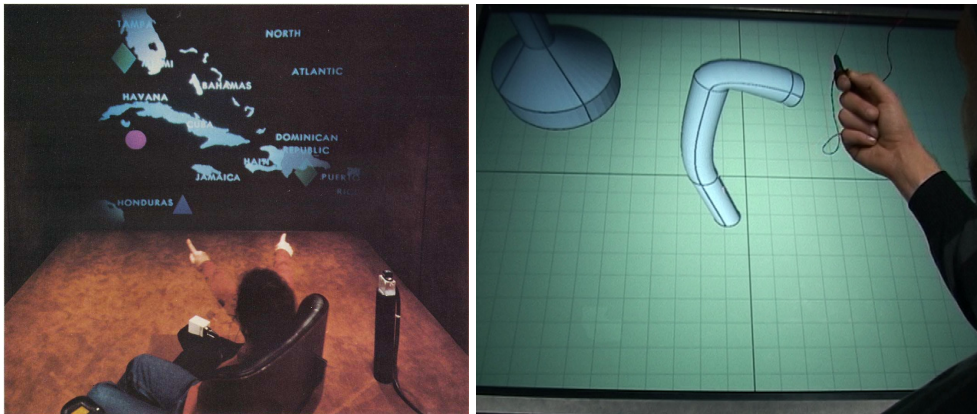


Figure 2.1: User selecting an object on the screen in *Put That There* [2] (left). User extruding an object in *Mockup Builder* [5].

One possible approach is to allow the user to express his intent using voice commands. In *Put-That-There* [2] a simple augmented environment with a room-sized screen allowed the creation of a 2D scene composed of simple shapes which could be moved, resized and colored using structured voice commands like "Create a blue square there", while the user pointed at a specific point on the screen. Object selection was performed by either naming the objects unambiguously or pointing at them (Fig. 2.1) with the index finger, and several CAD operations can be performed through specific voice commands.

Another possible approach is to track the user's hand gestures in three dimensions. In *Mockup Builder* [5] an augmented reality content creation setup is presented where the user can sketch planar figures freely on the surface of a stereoscopic screen using their dominant hand's fingers, then extrude them upwards with a pinch gesture to create 3-dimensional meshes (Fig. 2.1). Bimanual gestures are used to move, scale and rotate meshes, and a contextual menu can be opened on the screen to snap objects to the plane and perform other more complex operations.

### 2.1.1 3D Object selection and manipulation

The first milestone for modelling in VR is an interaction model for object selection and core manipulation operations - translation, rotation and scaling. Naturally this problem has been extensively studied and classification systems of interaction techniques have been developed by several authors. Weise et al. [19] propose one such system based on 13 categories. Underlying selection techniques is an interaction metaphor, usually one of grasping, pointing or surface selection.

For example, when grasping, the user places his hand - or selection device - on top of the object, closes his hand to grab it, and releases the object by reopening their hand. While the object is held, it can be directly translated by the grabbing hand, or with the help of the second hand also rotated and scaled.

On the other hand, when pointing, the user directs the index finger of their dominant hand - or selection device - towards the desired object like a cursor, and a ray is cast from the user's hand or

virtual viewpoint to intersect an object, highlighting it and prompting it for selection with similar grabbing techniques [1].

Other defining characteristics of different techniques include degrees of freedom, reference frames, input devices and disambiguation mechanisms [19] [1].

### 2.1.2 3D object searching

When creating a virtual scene an asset library is available, either locally or online, with 3D digital objects. The number of such assets has been increasing lately, particularly online, such that there is a need for adequate indexing and retrieval methods for 3D objects in large collections, as well as proper search engine interfaces and results visualization. [15]

Funkhouser et al. [8] introduced a search engine for object retrieval where 3D models are indexed based on their shape properties, queried interactively using text keywords, 2D and 3D sketching and iterative refinement, and matched based on shape and similarity.

Pascoal et al. [15] proposed an interactive search-and-filter interface for retrieval in virtual reality. Users of this spoken interface have a particular object in mind and begin by specifying some of its features, usually color and dimension; the interface then presents a list of candidate objects in a half-barrel layout, and users navigate through the list to find their object, or filter the results again with more features - gradually discerning the candidates from the intended object. It is also possible to search for similar objects. In this system the queries are spoken aloud from a prebuilt vocabulary but still have natural syntax.

### 2.1.3 Approaches to content creation and interaction

Content creation can exist in many forms: one approach, and perhaps the simplest one, is to start with a vast library of primitive shapes, like cubes and spheres, and premade assets - themselves made with other content creation tools - and *instantiate* them to populate a virtual world (like a city) or assemble a more complex model (like a car). This LEGO-like approach requires only object selection, retrieval, and basic manipulation operations, and constitutes the baseline of all professional 3D modelling software.

In a virtual reality setting the interaction model constitutes the various mechanisms and techniques available to the user to interact with the tools provided by the application to manipulate and create new content; it is dependent upon, at least, the input devices available, the desired degree of precision, and the expected learning curve for new users.

A possible extension of the *instantiation* approach includes deformative CAD operations such as boolean operations, and this is the approach taken by MakeVR [11] and Mendes et al. [12] in a full virtual reality setting.

In MakeVR [11] the only input device is a VR controller, one for each hand, so the interactions revolve around bimanual gestures for navigation and basic manipulation, plus the joysticks and buttons on the controllers for other operations like boolean deformations, application of colors and textures. There is also a floating menu in the virtual world with even more operations.

In Mendes et al. [12] objects are grabbed by closing a hand while on top of it, dragged while maintaining the hand closed, and released by opening the same hand later, an implementation of the *virtual-hand* metaphor [1]. The core constructive solid geometry (CSG) modeling workflow employs an assembly metaphor where users perform the basic CSG operations - union, intersection and difference - by grabbing and dragging two objects with both hands simultaneously, overlapping them in the air, and after a short pause - to lock the objects in place - then releasing or moving the hands away from the intersection (Fig. 2.2). Alternatively the user can release the dominant hand object only, and a WIMP-style menu appears with all four operations, from which one is selected by a grab gesture. Instantiation begins with a spread fingers gesture with the non-dominant hand facing up, evoking a palette metaphor, to spawn a WIMP-style floating menu from which objects may be selected.

Most modelling systems that we studied, and which target natural interaction in a virtual environment, approach content creation in a different manner [12], preferring a sketching-based approach where the primary tool is a real or virtual pen, or the user's fingers, and content is made with drawing gestures on a 2D surface or in mid-air.

Teddy [10] introduced a familiar sketching interface for freeform models on a traditional mouse or tablet setup, where 2D sketches are turned into 3D polygonal surfaces naturally. Modeling begins with a 3D primitive or closed silhouette being drawn on an empty canvas. This sketch is then *inflated* in such a way that wide areas expand more and narrow areas expand less, so that an ellipse becomes bread-shaped, a circle becomes a sphere, and a rectangle becomes cylindrical. The 3D model can be rotated and freely painted with open strokes, extrusions are initiated with a closed stroke inside the model (Fig. 2.3), and scribbling chaotically on a region of the model erases all painted strokes, and scribbling in extrusion mode smooths a region, eliminating bumps and cavities.

Closely related is the work of Cochard et al. [3] in which the interaction model centers around the use of a tracked glove on the dominant hand and a tracked rotating ball on the non-dominant hand (Fig. 2.4). 3D meshes are generated using one of two *shape builders* (extrusion or Teddy-like [10]) after the user draws a freeform shape using their dominant hand's fingers. The rotating ball is used for scaling objects but also coloring and texturing, by laying out the different menu options

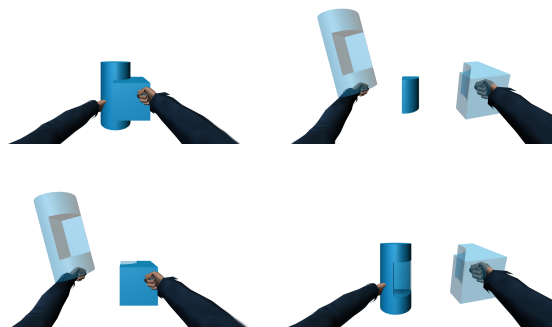


Figure 2.2: CSG operations in Mendes et al. [12].



Figure 2.3: Extruding operations in Teddy [10].

and colors around the ball's surface.

In DesignAR [16] the core modelling loop is very similar, but the setup is an augmented reality workstation with a stereoscopic screen so users can work in three dimensions. In this approach, after a closed silhouette is drawn on the screen (Fig. 2.5), it can be extruded "out of the screen" along the z-axis or rotated into a solid; then users can refine the 3D model by creating new vertices and edges, slicing the model in two, and extruding faces along their normal vectors. The interaction model is centered around the use of a design pen on the dominant hand to create content like splines, new vertices, edges and faces on the screen, and supportive actions are performed through the second hand. The main goal of this interaction model is to cement the intuition that the pen is the primary content creation tool, and to use touch interaction techniques for most modelling features due to their high fidelity. In this case, mid-air interaction is lacking, and mostly limited to selection tasks with the second hand.

Multiple sketches can also be combined to form a more elaborate model. Nishida et al. [13] proposed a modelling approach where urban buildings are automatically generated from a series of manual sketches that represent various aspects of the building, such as the roof, facade and windows (Fig. 2.6). A convolutional neural network is used to identify which of several standard *snippets* best fits the drawn sketch and with which parameters, and that snippet is then chosen to

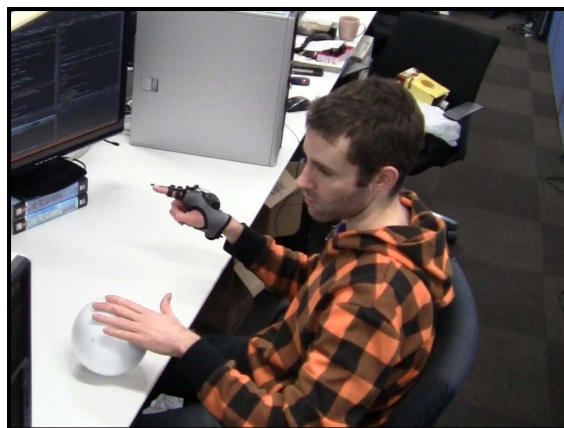


Figure 2.4: Glove and ball modelling setup in Worlds with Strokes [3].



form a part of the building's *grammar*. Various building *blocks* can be generated this way and assembled into a more complex model.

In a fully immersive virtual reality setting applications should make use of available 3D tracking tools and develop an interaction model with a 3D input space, either gesture-based (like Mid-air modelling [12]) or controller-based (like MakeVR [11]). It is also possible to combine this input model with speech commands, pen-based input, etc.

In ErgoDesk [7], the interaction model combines 2D pen-based input with 3D tracker input and simple spoken speech commands. An interactive tabletop is used for visualization and for 2D-input, and a sketching pen is the primary means of content creation. It has three buttons, each one used for a different task: drawing gestures, object manipulation and camera movement. Another tool is a 6DOF 3D tracker used to help rotate and scale the field of view to visualize objects from different angles; a second 3D tracker can be used to annotate the object at the same time. The visualization mode is stereoscopic while this prop is in use and monoscopic (2D) otherwise, while the pen is being used. Speech input is limited and is used to issue simple commands like copying, instantiating, and changing the color of objects.

In DIY World Builder [18] we see a mix of several interaction and modelling paradigms. Users build virtual worlds using both a virtual wand and a real tablet attached to their non-dominant wrist. Object selection and basic manipulation is performed using the wand; various tools are available and can be selected on the tablet, including a brush to fill terrain and a height brush to relevel the terrain surface. To create objects or apply textures and other properties to existing objects, the user selects from a palette in the tablet and places the object with the wand. Navigation is accomplished with real walking and augmented with wand-pointing for longer-distance movement.

In Low-cost 3DUI [9] a simple modelling workflow in virtual reality is presented where a pair of Wii controllers are used to instantiate primitive objects and manipulate them with all basic operations. Object selection is performed with a *raycasting* metaphor with the ray emanating from the user's shoulders through the user's dominant hand; object scale is adjusted according to the distance between the user's hands; finally, objects are rotated by moving both hands while they simultaneously *hold* an object mid-air.

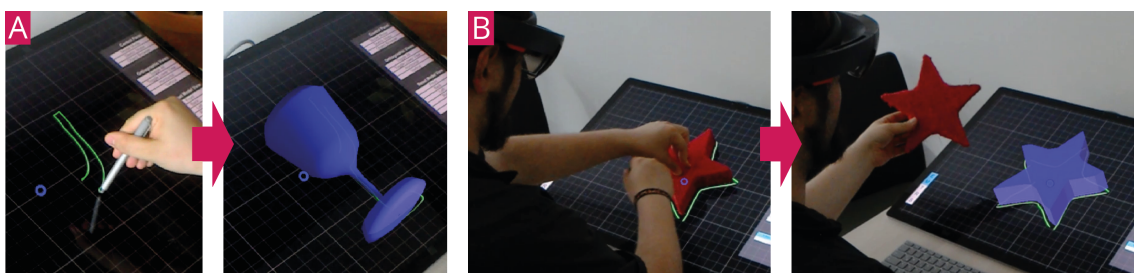


Figure 2.5: Creating new models in DesignAR [16]: Creating a contour and then generating a rotational solid (left), and drawing the contour of a real world object (right).



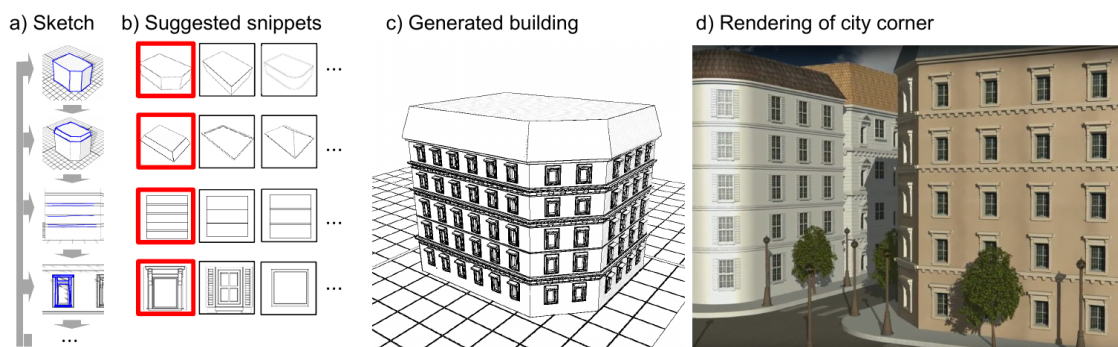


Figure 2.6: Creating a new building from multiple sketches [13].

## 2.2 Procedural content generation

In simple terms, procedural generation refers to a methodology of creating new content or data through the application of a *function* instead of by hand. The generated content can be adapted from other content, for example, by applying random variations through computer-generated noise, solving constraint problems algorithmically so the new content satisfies specific properties, or just exercising a set of transformation rules defined by the creator in an appropriate procedural generation language. This practice can be applied to create various different types of data, from textures and random meshes to terrain, trees, road networks, cities and buildings [17]. We narrow down our overview of existing methodologies to those for generating buildings (interiors and exteriors), cities and city layouts.

### 2.2.1 Procedural grammars for cityscapes

Coelho et al. [4] present a survey of techniques for procedural modelling of terrain, cityscapes and building facades based primarily on *L-systems* and split grammars. A basic *L-system* is composed of a formal grammar and a collection of *production rules*; starting with an *initiator* string, each *atom* in the string may be expanded into a string of symbols according to a *generator* rule, recursively. They are usually accompanied with a mechanism for generating a figure or geometric object (Fig. 2.7).

Procedural generation of cities began with the work of Parish and Muller [14]. Their system, called *CityEngine*, takes as input various geographical and sociodemographic maps with information such as terrain elevation, water bodies and population density, and extended L-systems are used to derive the road network of the city (highways and roads) and the distributions of buildings; a simple L-system is used to generate the buildings themselves after being extruded from the shape of each allotment.

Split grammars are a type of formal grammar that incorporate the notion of geometrical shape into the production rules, making them a better fit for automatic modelling of architecture [4]. In this grammar, the alphabet's symbols are (usually 2D) shapes that can be *split* according to the production rules and annotated with additional information like size. They were introduced

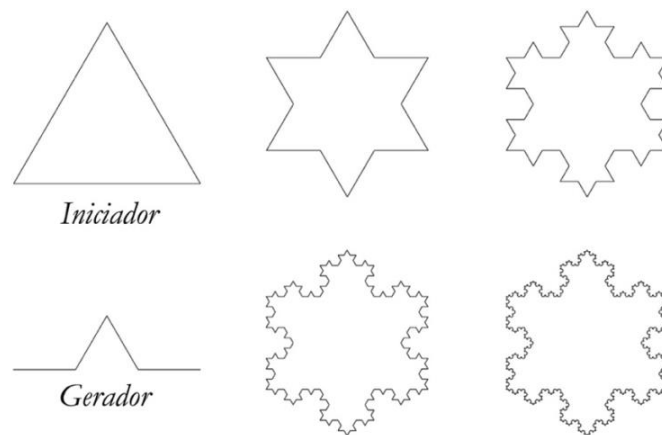


Figure 2.7: A basic L-system producing a self-similar polygonal fractal [4].

by Wonka et al. [20]. In this work, the symbols of the grammar are augmented with parametric attributes representing additional information like physical dimensions of shapes, depth and texture data. A control grammar is further used to help refine the attributes of the shapes created by the split grammar. The resulting buildings exhibit non-planar doors, windows, balconies, etc. constrained by the designed rules.

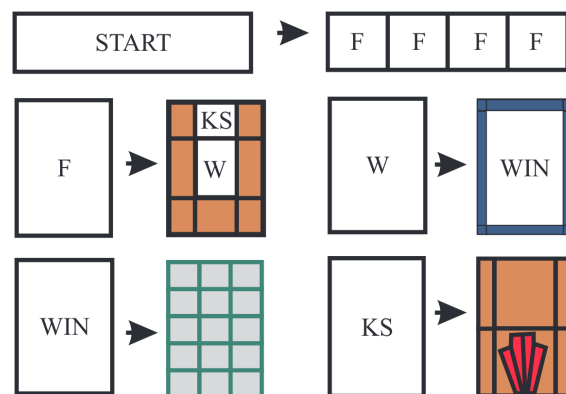


Figure 2.8: Example split grammar rules for generating a building facade [13].

## 2.3 Discussion

In the previous section we have covered several different works in the literature and analyzed their approaches to content creation, their interaction models, input and output devices. All works are substantially diverse, but several common patterns arise amongst these categories.

When it comes to input and output devices, a primary distinction can be made between those works utilizing 2D and 3D input and output spaces. A modelling solution with a 2D input space usually has either a WIMP interface on a computer with mouse and keyboard input, or touch input on a tabletop or tablet. A 3D input space requires tracking of the user's hands with armbands,

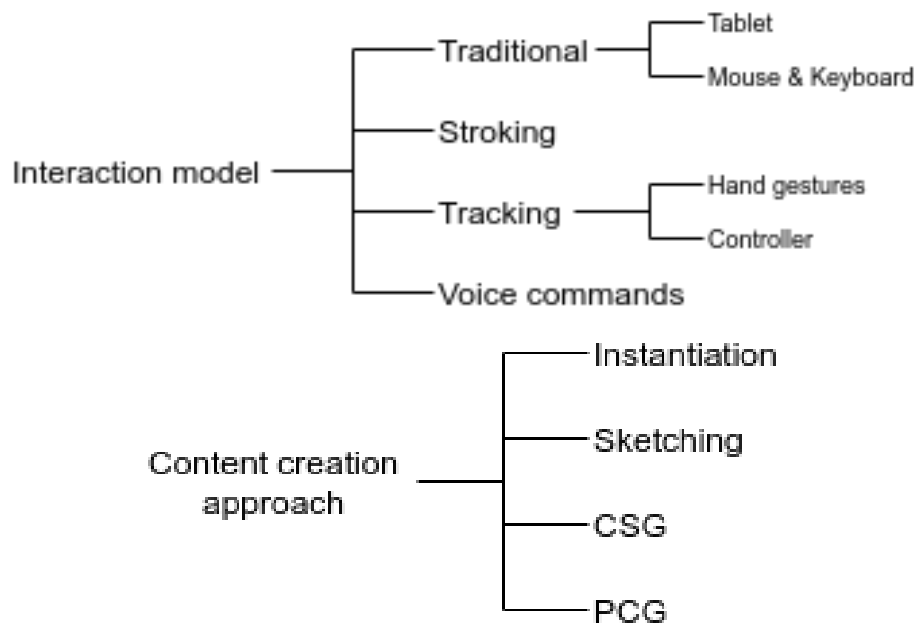


Figure 2.9: Taxonomy of interaction models and content creation approaches.

gloves, motion sensors or cameras. Most drawing and sketch based approaches require a considerable degree of accuracy for their input, and so prefer 2D input spaces given the low fidelity and tracking precision of current 3D tracking devices. When it comes to user’s perceived output space, a 2D output space includes tablets, non-stereoscopic tabletops, and computer screens, while a 3D perceived space provides stereoscopic visualization of the 3D world, with example devices including stereoscopic tabletops/screens and head-mounted displays (HMDs).

Interaction models are the ways in which the user interacts with the modelling system and is closely coupled with the input/tracking devices used (Fig. 2.9). Some of the approaches revolve around using hand gestures or 3D-tracked controllers either in a fully immersed virtual reality environment ([11], [18], [12], [3]) or in an augmented environment ([16], [5], [7], [2], [9]). Others rely on stroking and drawing in either two or three dimensions with a pen or just the user’s fingers ([16], [13], [7], [5], [10], [3]). Voice commands can be a primary or secondary input mechanism depending on the modelling approach, with uses ranging from tool selection to object manipulation and instantiation ([7], [2]). Finally some works studied rely on more traditional WIMP-based input from mouse and keyboard ([13]) or tablet ([18]), particularly those works of procedural modelling of urban buildings which do not put of focus on the interaction model at all ([20], [14]).

Approaches to content creation we can group in the four categories of *sketching*, *instantiation*, *CSG* and *PCG* (Fig. 2.9). Most works are sketch and drawing based, where the user must draw the contours of objects they wish to create, or draw on already existing ones to cut or extrude them ([11], [16], [16], [7], [5], [10], [3], [9]). The instantiation approach is the simplest one – the user selects an asset from a collection and places it in the scene, performs basic manipulation operations – translations, rotations and scaling - and adjusts other properties such as textures and lighting ([11], [18], [7], [12], [2]). In the *CSG* approach the user creates new objects using constructive

solid geometry operations like intersection and union of two or more objects ([16], [5], [12]). Finally with the *PCG* approach the user does not directly create every object instance but instead specifies rules, patterns and properties that are used to generate the object(s) ([13], [20], [14]).

Many of the works we studied combine 3D input and perceived spaces ([11], [16], [18], [5], [12]) while combining two different content creation approaches: sketching ([11], [16], [5]), instantiation ([11], [18], [5], [12]) and CSG ([16], [5], [12]). However, we have not found a modelling approach that combines 3D input and perceived space in a fully-immersive virtual environment with procedural generation techniques and a natural interaction model, and the goal of this dissertation is to fill in this niche approach to content creation.

Work	Input Space		Perceived Space		Interaction model				Content creation			
	2D	3D	2D	3D	Tracking	Stroking	Voice	Traditional	Sketching	Instantiation	CSG	PCG
MakeVR [11]		•		•	•				•	•		
DesignAR [16]	•	•		•	•	•			•		•	
Urban models [13]	•		•			•		•	•			•
DIY World Builder [18]	•	•		•	•			•		•		
ErgoDesk [7]	•	•	•		•	•	•		•	•		
Mockup Builder [5]	•	•		•	•	•			•		•	
Mid-air modelling [12]		•		•	•					•	•	
Teddy [10]	•		•		•	•			•			
<i>Put That There</i> [2]		•	•		•		•			•		
<i>Worlds with Strokes</i> [3]		•	•		•	•			•			
Low-cost 3DUI [9]		•		•	•				•			
Instant Architecture [20]	•		•					•				•
Procedural Cities [14]	•		•					•				•

Table 2.1: Classification of related modelling works.



## Chapter 3

# Procedural VR

The system we propose allows a user to assemble one or more *models* in a *modeling sandbox* and is intended for a virtual reality environment. A model in this system is just a layout of various static assets, with some extra hierarchical relationships between these *objects* – such as groups – and *procedural rules* to change their visibility status, placement and orientation *randomly* in a controlled fashion. The system allows the user to build this model by providing them core modeling operations, such as cloning and moving objects, *essential entities* such as groups and uniformly spaced tilings, and an assortment of **randomization operations and objects**. We shall describe all of these in detail in this chapter.

We start with a global vision of the system and a brief summary of the difficulties our modelling approach was designed to overcome. Then we go through a short, high-level overview of how a user would use the system to create a simple computer room model, introducing some of the concepts and entities within the context they are meant to be used. Then we describe in more detail the various modeling operations supported, the concepts and objects in the system, and the relationships between these, primarily from a user’s high-level point of view.

This chapter is best understood as an abstract introduction to a modeling approach, namely a set of operations and objects. We do not discuss here the *realization* of this approach, particularly the interaction protocols used to perform any of the modeling operations in the sandbox – these concern the prototype we developed, and we explore those aspects in the next chapter.

### 3.1 Vision and usability goals

Recall the game development example from chapter 1. The following is a summary characterization of the problem:

- It is **repetitive**: if performed naively, the amount of the work required to polish and create all these minor models of the game scales proportionally to the amount and size of such models, even though many such sites look very similar to each other.

- These models are **static** within the game - or at least not *interactive* - in a significant number of cases. Static here means the objects do not move in the world, at least not as a response to what the player is doing or to what is happening around them.
- Imperfections and minor issues are **inconsequential**: it is acceptable for some models to look misplaced, like a box overlapping another on top of a table, as long as it is uncommon and not *gross* enough to be distracting.
- Collectively the previous three issues make it a burden to work on these models.

We believe the approach we will describe and have developed addresses these issues especially for situations and models similar to the examples we gave at the beginning of this section.

- It addresses *repetitiveness* by supporting the creating of **randomized models**. Whenever such a model is **instantiated** some of its elements may appear, disappear or change **layout**. The randomization is in a certain sense **predictable** and completely controlled by the designer. This allows a model to be designed so as to be used for many contexts.
- The scenarios/models created are completely **static**, though they can obviously still be integrated with an engine that supports live collisions, gravity, etc.
- Even in the presence of large amounts of layout randomization, our approach offers many ways for users to prevent or reduce the likelihood an instantiated model is *inadequate* because it has issues such as:
  - contains overlapping objects;
  - has too much self-similarity or symmetry;
  - contains elements that are disproportionate (in size or shape);
  - contains elements that do not go well with each other.
- The set of procedural rules available is quite small, and it allows many different models to be created from the same set of assets. The learning curve can range between being steep for users with no prior modeling experience, to being very flat to users with both modeling and programming experience.

## 3.2 High-level example overview

In this section we will place ourselves in the shoes of a game designer. Suppose we wish to assemble a computer room for a game. Not a room the player will interact with meaningfully, just a room the player may pass through on his journey. Assets in this room are really just "lying around" in the game.

We may have several such rooms in our game, so we will develop a **randomized model** of the room, one we can essentially copy-paste from place to place – manually in an adventure game, or programmatically in a procedurally generated game.



We will also assume that we have the assets for the room already. Preferably we have several *variants* of each *canonical kind of asset*: tables, monitors, books, laptops, chairs, keyboards, trash bins, carpets...

The computer room could of course be designed in many different ways. We will follow these design guidelines, that any instantiated version of the randomized model must follow:

- The model of the room will not include the doors, the walls or anything on them, only tables with computers and other office props.
- The room has a fixed rectangular size.
- The "basic unit" in the room is a table of modest size with exactly one computer on it and one chair next to it, plus some more items like a book, a notepad and a photo.
- The tables are laid out in an open office style, with no barriers between them. Non-uniform layouts – tables not lined up like in many school classrooms – are allowed.
- The tables themselves must *never* overlap each other, and it must be very unlikely or impossible for elements on top of the tables to overlap each other.
- The position of tables on the room and of items on the tables should vary per room instance and per table instance, respectively.
- The tables themselves, the computers, chairs, and other items on top of the tables should be a different *variant* per table instance, and different for each instance of the entire room model.

The variants mentioned above are simply **different assets** - different colors, shapes and styles of chairs, for example. Our proposed approach does not take into consideration any sort of low-level mesh editing (like boolean operations or deformations), texturing, and in fact does not even support object scaling.

The modeling process to create this room model can be roughly broken down into the steps listed below. We will clarify some of the entities and operations in the next section.

- Prepare a scene (out of the application) with all of the relevant assets, laid out in some reasonably organized way. One instance of each asset will suffice. Then start the application and enter the *modeling sandbox*.
- Create a Random object that aggregates each canonical kind of asset. We'll have one such entity for tables, one for chairs, one for monitors, one for keyboards, one for books, and so on. Each time one of these Random objects is *cloned*, one of the aggregated assets is chosen and displayed, uniformly at random.
- Assemble the main elements of one table together: the tabletop, the chair and the computer objects. For example, position the computer on the center of the table facing the longer side, and the chair right in front of it.

- Add more objects around the computer such as books, notepads, coffee cups, mousepads, photos, lamps, etc. Here by *objects* we mean the ones created in the second step.
- Add a slight amount of positional and rotational randomization to the objects on top of the table with the Mover and Rotator modifiers, respectively. This way the objects' position and orientation relative to the table varies in each instance. This serves to generate tables with visually different layouts.
- Create a Group object containing all the objects on top of the table. Objects which are part of this group have a fixed layout within the group. If having varied layouts for the objects on top of the table is desirable, then more groups can be created with similar compositions but with different layouts of the computer and table assets, and then aggregated with one Random.
- Create another Group object containing the tabletop, other elements beside of it, like the chair and trash bin, and the group object from the previous step. This is the **table model**, our basic unit.
- Clone this group several times, creating a new instance of the table model each time. Drag each instance to a different place within the area delimited for the room in such a way that the tables don't overlap. Lay them out in some organized fashion: rows by rows like a matrix for example, or all next to each other and facing outwards.
- Group all the tables together. This is the **room model**, our final design. This model can now be exported and embedded in some part of the game.

The result is a room model which, whenever *cloned*, produces a different room whose tables have different compositions and whose items have different arrangements on top of the tables. If this model is now used in a game each instance of it will be distinct.

### 3.3 Objects and Operations

We have informally introduced some object types and operations. Now we explain these in more detail.

When a user enters the modeling sandbox they have access to the **primitive assets** loaded into the *active scene*. Our approach does not specify a mechanism to spawn these primitive assets, neither prior to entering the sandbox nor on-demand during the modeling process. Our prototype sets up the scenes at compilation time. We proceed by assuming these assets are already available in the sandbox.

We call these initial assets **Primitive** objects because they can only be moved around and copied, they cannot be edited or partitioned into smaller blocks, unlike the *procedural* and *composite* objects we shall describe next.

The first composite object is the **Group**, which is the union of one or more objects with a fixed relative layout. A Group is created by switching to the *Create Group* tool, selecting one or more objects in the sandbox, and then joining the selection into one aggregate object. The incorporated objects are called the *elements* of the group. Once formed the Group can be moved and rotated as a single object and the elements keep their relative distances and orientations.

The first procedural (also called *randomized*) object is the **Random**, which can be constructed with the *Create Random* tool in exactly the same way as a Group, but instead of showing the union of all incorporated objects with a fixed layout, it shows exactly one of these objects, chosen uniformly at random and independently of all other procedural objects. The incorporated objects are called the *variants* of the Random, and exactly one of these variants is *visible* or *active* at any moment. Upon creation of the Random, all variants snap to the position of the first one selected at creation time, but retain their world-space orientation. Intuitively all the variants appear, by default, in the same place, though this can be adjusted later.

Another composite object is the **Tiling**. A Tiling wraps a single object, the *tile* or *child*, and lays out a fixed number of clones of this tile along a straight line with uniform spacing. The number of tile clones, the spacing between tiles and the orientation of the tiles can be adjusted, and the tiling direction can be changed by rotating the Tiling object itself.

The last two procedural objects in the system are the **Mover** and **Rotator** objects. Similar to a Tiling, these wrap a single other object – the *child*. The Mover adjusts the child’s location by applying a *random offset* chosen uniformly at random from within a box-shaped *offset range*, and the Rotator adjusts the child’s orientation instead, around a *primary axis*, by a *random angle* also chosen uniformly at random from within a permissible *angle range*. Both these offset and angle ranges are specified when the object is created. Given that these two objects simply modify an aspect of their child element we also call them *modifiers*.

Finally there is the **Empty** object, which behaves identically to a Primitive object but is actually *invisible*, and there is mechanism within the sandbox to toggle their visibility to the designer. This object can be used to represent *nothing* among the variants of a Random, and if the model exported from the application is imported to some other application it is intended that they be deleted or ignored along with hidden Random variants.

All objects can be naturally composed: the elements of a Group can be other composite or procedural objects, and so can the variants of a Random and the child element in a Tiling, Mover or Rotator. Let us see a few examples:

**Computer Prop** Suppose we have three primitive assets for keyboards and four primitive assets for desktop monitors. To create a randomized desktop computer prop from these primitives, create a Random *A* from the three keyboard primitives, another Random *B* from the four monitor primitives, position *A* just in front of *B*, and then create a Group *G* with *A* and *B* as its elements. This creates a model *G* that shows exactly one monitor and exactly one keyboard, which ones though are chosen randomly.

**Chair Prop** Suppose we want to place a rolling chair in front of a table and want to have it look "natural" by not having it centered nor perfectly aligned with the table. If we have multiple rolling chair models we can start by aggregating them in one Random like in the previous example. Now place the chair exactly in the perfectly centered location in front of the table, wrap it with a Rotator  $R$  with an angle range of  $\pm 30^\circ$  around the vertical axis, then wrap it with a Mover  $M$  with a box offset that's close to a straight line parallel to the main axis of the table. The modifiers should be applied in this order. This yields a model  $M$  in which the chair is always at a fixed distance from the table, but can appear positioned further left or further right of the center and be facing slightly leftwards or rightwards.

**Chessboard Prop** Suppose we have primitives for chess pieces and a chessboard, and we want to create a randomized chessboard prop with pieces on it (in a random chess position, not necessarily legal). Create a Random  $C$  holding all the chess pieces. To add some imperfection to the pieces' placements, wrap  $C$  in a Rotator with an angle range around the vertical axis, then wrap it in a Mover with a very small offset range, say 5% of the width of a board square. Wrap this in a Tiling  $T_1$  with 8 tiles, spaced out like the chessboard's squares, and align it vertically along a file. Wrap  $T_1$  in another Tiling  $T_2$ , again with 8 tiles properly spaced out, and align it horizontally along a rank of the chessboard. This creates a *matrix tiling* with 64 unit tiles in total. If in the Random  $C$  we include one or more Empty objects as variants then some of these tiles will be invisible, i.e. empty squares. This schema does not guarantee the board is in a legal position, nor that the count of pieces is legal.

The composition of objects essentially forms **object trees**. The children of a group node are the Group's elements, the children of a random node are the Random's variants (including those not shown), and the other three object types have only one child. The leaves of an object tree are all Primitive or Empty entities.

The roots of the object trees in the examples are  $G$ ,  $M$  and  $T_2$ , respectively. A visualization of these trees is shown in fig. 3.1. Once the models are built, these three objects are the only objects that the designer can interact with in *normal mode*. But now suppose we have decided to change some object that is not at the root of an object tree, for example we may want to add a new chess piece to the Random  $C$ . We can do this by **editing objects**.

### 3.3.1 Editing objects

All composite/procedural objects may be **edited after creation** – to adjust children positions and orientations, add and delete elements or variants, or adjust parameters like spacing. When editing an object  $X$  this object is called the **edit subject** and only the children of  $X$  can be modified or moved. The operations available while editing are only those that pertain to adjusting the subject.

When editing a Group the designer can modify the relative positions of the elements, delete elements, and add new elements, either by cloning elements within the group or from outside into the group. Similarly, when editing a Random the designer can change the relative positions of the

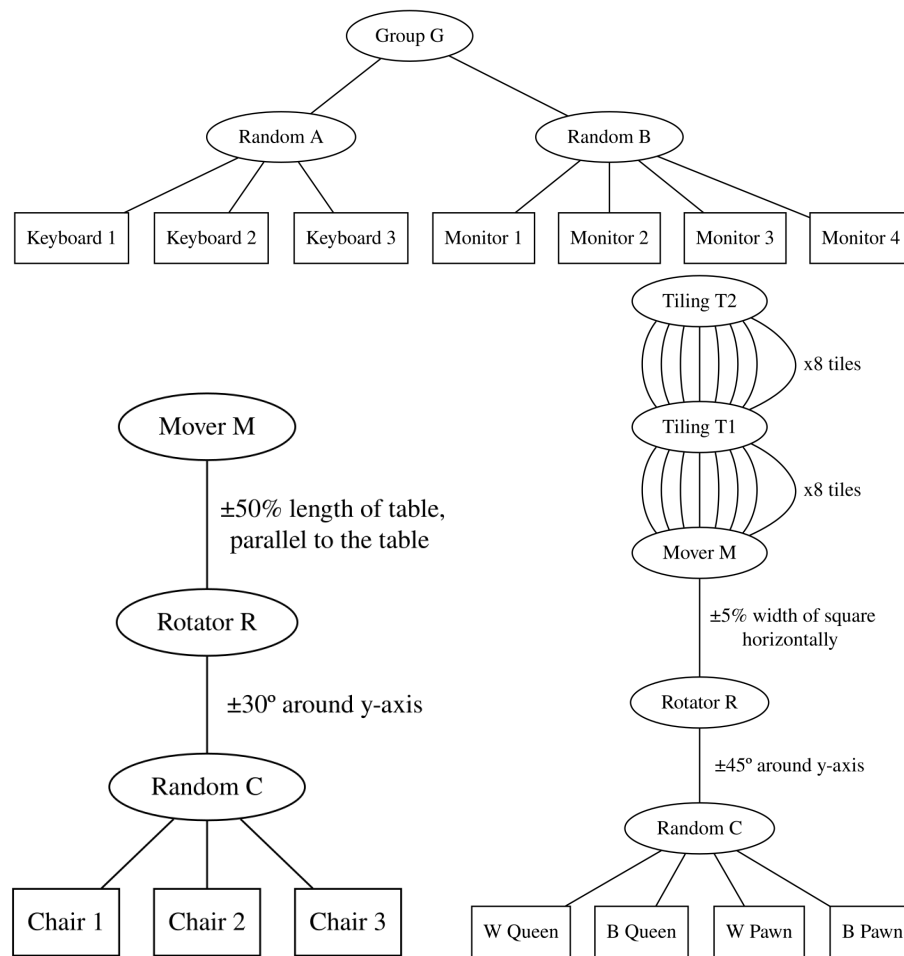


Figure 3.1: Object trees for the three examples. The chessboard tree is shown in compressed form with four piece primitives. Square nodes represent primitive objects.

variants and also add and remove variants. In either case, objects outside the Group/Random being edited cannot be moved around, deleted or modified in another way. In the case of the Random, this introduces the ability to have the variants appear with different relative positions.

Tilings, Movers and Rotators actually *require* being edited as the designer should set their parameters upon creation, so the system should immediately switch to *edit mode* when these objects are created. For the Tiling, the number of tiles and the spacing between the tiles can be adjusted. For the Mover, the permissible offset box appears while editing and it can be resized. For the Rotator the permissible angle range can be adjusted around one of the three rotation axes.

Editing may be performed **recursively** in the object tree: while editing a Group the editing process can move to one of its children; while editing a Random the active variant may be edited instead; and while editing a Tiling, Mover or Rotator the child entity can be edited. Let us see a few editing examples:

- **Computer prop.** Suppose we just created Group *G*, but we are not satisfied with the relative

positions of  $A$  (the keyboard) and  $B$  (the monitor) and we would also like to add a mousepad to the model. If there are multiple mousepad primitives we can create a Random  $P$  to switch between them. Now enter edit mode of  $G$ ; the elements  $A$  and  $B$  can be moved independently of each other, and so we can reposition them. The Random  $P$ , which is not an element of  $G$ , can be *cloned* into the group as the third element and placed beside the keyboard.

- **Chair prop.** Suppose we want the chair to rotate freely not just  $30^\circ$  but  $180^\circ$  all around. We can enter edit mode on the Mover  $M$ , and then recursively edit the Rotator  $R$  (child of  $M$ ), and there modify the angle range to our liking. When done we *return from editing* twice, first to conclude editing  $R$  and then  $M$ .
- **Chessboard prop.** Suppose we did not include any Empty object in the deeply nested Random  $C$ , then the chessboard shows 64 chess pieces which is clearly excessive. We can recursively enter edit mode of  $T_2$ ,  $T_1$ , the Mover, the Rotator and then  $C$ . Here we can clone an Empty object into the Random as a variant, perhaps several times. When done we must conclude editing 5 times to return to normal mode.

In the first and third examples we mentioned *cloning*, which is one of several operations available in normal mode and some edit modes, but there are several others that are important to develop models.

### 3.3.2 Operations

There are a few more operations available beyond moving around entities, creating and editing them. The following operations are available in normal mode, when not editing entities.

A **Clone** operation can be applied to any object, and it creates another instance of the same object, with its children cloned recursively. The object tree of the clone is identical to the original. Cloning a Primitive simply creates another Primitive with the same asset. This is the normal mode operation; when editing Groups and Randoms a similar operation is available that introduces new elements and variants into the Group and Random respectively, as discussed previously.

A **Reroll** operation can be applied to any object, and it requests the object to make a different set of random choices, recursively down the object tree. When applied to a Primitive object the operation does nothing. When applied to a Random object it requests that another variant be shown, and that the shown variant itself be rerolled. When applied to a Group the elements are themselves rerolled, and similarly for a Tiling. When applied to a Mover the random offset is rerolled, and when applied to a Rotator the random angle is rerolled, and then the child object itself is rerolled as well.

A **Disband** operation can be applied to any object and it serves to "break apart" a composite object into its constituents, by deleting the root node of the object tree but not its children. When applied to a Primitive object this operation does nothing. When applied to a Group or Tiling with  $n$  elements or tiles, the Group/Tiling node disappears while the  $n$  entities remain as object roots. When applied to a Random a similar thing happens, but only one root remains, the visible variant,

with the hidden variants being deleted. When applied to a Mover or Rotator the position/orientation offset are bound to the child object, as if the designer had put the offset by hand, and then the modifier node is removed, leaving the child element as a new object root. Intuitively, this operation does not perform any visual modification to the model, it only modifies the object tree structure.

Finally the **Delete** operation simply deletes the whole object tree rooted at the object it is applied to. This is also available while editing Groups and Random to remove elements and variants, respectively.

### 3.3.3 Clone by reference

In the previous sections we overlooked an important aspect of the Clone operation, perhaps the most important aspect of our approach: composite and procedural objects are **cloned by reference**.

When a Group  $G_1$  is cloned in normal mode into another group  $G_2$ , the two groups are **linked**. If another clone  $G_3$  is made of either  $G_1$  or  $G_2$ , then the three groups will be linked to each other. The same link-on-clone logic applies to the other composite and procedural objects, and to clone operations performed while editing. We call two objects that are linked to each other **siblings**.

Linking is **recursive**: if  $G_1$  has another linked object  $A_1$  as its element, then the corresponding clones  $A_2$  and  $A_3$ , in  $G_2$  and  $G_3$  respectively, are also linked to each other and to  $A_1$ . We say that  $A_2$  and  $A_3$  are **cousins** of  $A_1$ .

Linking **propagates edit operations**. When a linked object is modified in edit mode, the edit operation automatically propagates to all of its siblings. For example, if the designer edits group  $G_1$  by cloning an object  $X$  and adding it as a new element  $X_1$ , then two clones  $X_2$  and  $X_3$  of  $X$  – the cousins – are automatically created and added to  $G_2$  and  $G_3$ , respectively, in the same position and orientation. If the designer then moves  $X_1$  inside  $G_1$ , the same movement is applied to  $X_2$  and  $X_3$  in their own local space, and so on for the remaining operations available to edit  $G_1$ . The designer has **live feedback** of these mirror operations on the sibling objects, so he can visualize the effect of his changes simultaneously on all siblings that fit in his field of view.

Linking is **reciprocal**. Edits to  $G_2$  also propagate to  $G_1$  and  $G_3$ ; there is no "master" object.

Linked objects are **not necessarily identical**. If two Randoms  $R_1$  and  $R_2$  are siblings they have the same object tree structure and also the same set of variants, but they do not necessarily choose to show the same variant. Similarly, two Mover objects  $M_1$  and  $M_2$  have identically sized offset ranges, but their random offsets are different. Two siblings still make different and independent random choices.

Using the **Unlink** operation on a linked object, the designer can break the link that object has to all of its siblings. In our example, applying the Unlink operation to  $G_3$  allows edits to  $G_3$  to not propagate to  $G_1$  or  $G_2$  and vice-versa. Edits to  $G_1$  will still propagate to  $G_2$  and edits to  $G_2$  will still propagate to  $G_1$ . If  $G_3$  is then cloned to  $G_4$  then  $G_3$  and  $G_4$  will be siblings, but still separate from  $G_1$  and  $G_2$ .

Because of the way the group  $G_1$  was built, any linked entity  $X$  that is an element of  $G_1$  is linked to its cousins in groups  $G_2$  and  $G_3$ . There is no mechanism in the system to unlink  $X$  in

these entities. This is what ensures that  $G_1$ ,  $G_2$  and  $G_3$  retain identical object trees in the face of recursive edits to their children.

The ability to add elements and variants to a Group and Random, respectively, introduces the possibility of creating *inclusion cycles* in the object tree. The simplest example would be trying to add  $G_2$  as an element of  $G_1$ . Clearly these cycles are not well-defined, so an attempt to create one is detected and the requested operation rejected. How this is done is explained in the next chapter.

We conclude this chapter with a summary of the mechanics and functionality of each object in table 3.1.



	Group	Random	Tiling	Mover	Rotator
Composition	Holds $n$ ordered elements, each with a fixed relative position.	Holds $n$ ordered variants, each with a fixed relative position, exactly one is visible.	Holds $n$ clones of a template tile, uniformly spaced along a line.	Applies a random positional offset to an object, chosen uniformly from an axis-aligned box.	Applies a random rotational offset around an axis to one object, chosen uniformly from an angle range.
Parameters	When created the center of mass is set to the first element's center of mass. All remaining elements retain their position and orientation.	When created the center of mass is set to the first variant's center of mass. All remaining variants retain their orientation and all positions snap to the center of mass.	When created the tiling has exactly one tile and zero spacing, and the system immediately enters edit mode.	When created the mover has a "point box" that does not allow any random offset, and the system immediately enters edit mode.	When created the rotator has an empty permissible orientation range, and the system immediately enters edit mode.
Clone	The elements are cloned, linked and rerolled recursively.	The variants are cloned, linked and rerolled recursively. The shown variant is repicked.	The tiles are cloned, linked and rerolled recursively. The tile count and spacing are not changed.	The child object is cloned, linked and rerolled recursively. The clone's positional offset is randomized.	The child object is cloned, linked and rerolled recursively. The clone's rotational offset is randomized.
Reroll	The elements are rerolled. The relative positions do not change.	The shown variant is repicked and that variant is itself rerolled.	Each tile is rerolled. The spacing and count do not change.	The positional offset is repicked and the child itself rerolled.	The rotational offset is repicked and the child itself rerolled.
Disband	All $n$ elements become top-level objects.	The shown variant becomes a top-level object, the other hidden variants are deleted.	Each tile becomes a top-level object, just like a group.	The child becomes a top-level object in its current position.	The child becomes a top-level object in its current orientation.
Editing	Modify relative positions and orientations, delete elements, clone new elements.	Modify relative positions and orientations of each variant, delete variants, clone new variants, cycle through the variants in order.	Modify the tile count, adjust spacing by dragging a utility handle at an end of the tiling range, adjust orientation of all tiles by dragging a utility capsule-shaped bell.	Modify the positional offset box by dragging a utility handle at one of the axis-aligned box's corners. The handle snaps near the axis to allow 1d and 2d offsets.	Modify the rotational offset range by dragging one of 3 utility handles that move along three tori surrounding each of the three axis. The handle snaps at multiples of $90^\circ$ .

Table 3.1: Functionalities of each type of object.



## Chapter 4

# Prototype Development

In this chapter we describe the architectural aspects of our system and the interaction protocols designed and implemented in the prototype to realize the modeling operations explained in the previous chapter. Our goal was a realization of our modeling approach that employs only *natural interaction* techniques in the virtual reality environment, in particular no numerical inputs to specify parameters such as random offsets.

Beyond the natural interaction protocols designed to move, create, edit and operate on objects, we also explain the feedback system that helps the user keep track of the current system state and the undo system that records and can revert any operation.

We also elaborate on implementation details of the prototype, including the representation of the object tree model and hierarchy, the command-based undo system, the reactive feedback systems, the dynamic floating menus and interaction modes, controller input mechanics, validation of operations, the challenges associated with editing objects recursively, and other minor features such as movement constraints, edit augmentations and global scaling. An overview of the implementation's internal architecture is found at the end of the chapter.

### 4.1 User Interface and interaction

The prototype was developed in Unity as a virtual reality game application with the SteamVR tool. All *scenes* and primitive assets available were packaged at compile time – we did not include a mechanism to create new scenes on-the-fly in this prototype. On start the application immediately recognizes the virtual reality headset and controller devices and loads into the modeling sandbox the first scene. The modeling process can begin immediately in this scene. The ability to load a new scene at any moment is available through a keyboard command.

For development we used an Oculus Rift virtual reality headset, but the application should support other SteamVR-compatible headsets, with at least two buttons on each of two controllers,

with only modifications to the input bindings. The virtual reality controllers are the only form of modeling input available, there are no voice commands or gestural input.

#### 4.1.1 Basic interaction

The user holds two controllers, one on each hand – one of the hands is taken to be the *dominant hand*, usually the hand the user writes with on paper. The other hand is the non-dominant hand. The dominant hand’s controller shoots a **laser** outwards that collides with the first object or interface entity in its path (fig. 4.1). The object or entity the laser collides with is highlighted with a *yellow outline*. The primary way to interact with objects in the sandbox is to point the laser at them, see the yellow outline, and then press the trigger button to perform the desired operation on the outlined object. The operation performed is dictated by the current **interaction mode**. The default interaction mode is *grab mode*, in which holding the trigger attaches the object to the laser and moves it around. The object can be moved while the trigger is held, as it will remain fixed at its current position and orientation relative to the laser (with some exceptions). When the trigger is released it is unattached from the laser, remaining in its current position. We call this a *laser grab*.



Figure 4.1: Hovering over a tree object. In the middle panel the user is in grab mode and moves the tree. In the right panel the user is in clone mode instead, and the cloned tree is immediately grabbed.

The laser is also used to interact with an interface menu floating over the user’s non-dominant hand (fig. 4.2). This menu is hidden unless the user’s palm is facing upwards (like holding a color palette). Unlike the objects in the scene, these buttons are simply *clicked* instead of being grabbed, by pointing the laser at them and pressing the trigger. The function of most of these buttons is to switch between different interaction modes, and each interaction mode changes the operation that is performed when the user presses the trigger over an object. Interaction modes are listed and discussed in section 4.1.3.

The *interaction loop* essentially boils down to: switch interaction mode in the floating menu, perform desired operations on objects using the laser, and repeat. There is, in fact, only one multi-step interaction protocol, the one used to create Group and Random objects.

The user can also *grab* objects directly by moving the controller over them and pressing the hand grab button in the back of the controller. This can appear more natural at first, but it turned out to be much more cumbersome than just using the laser grab, so it should be avoided and will not be discussed further.

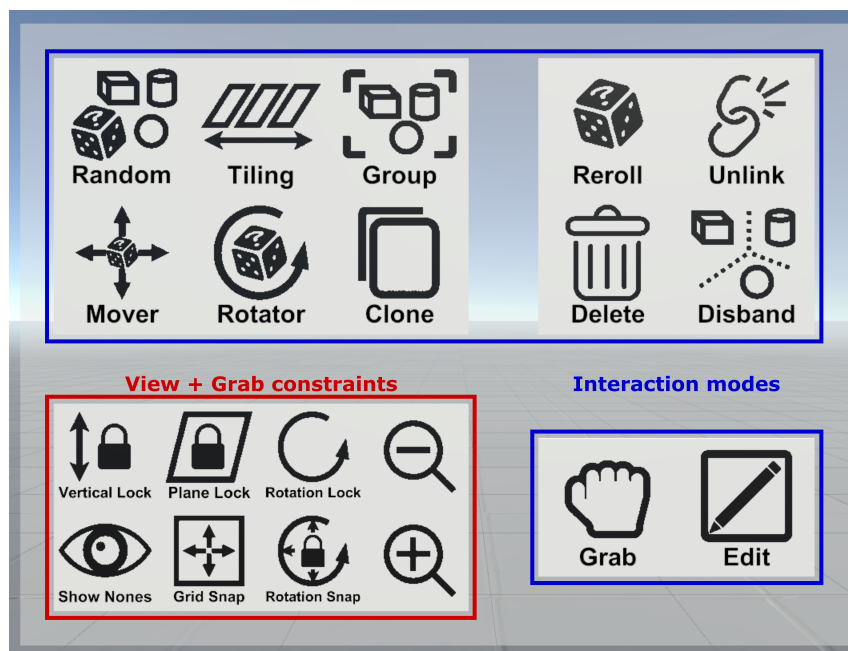


Figure 4.2: Top-level floating menu over the user's non-dominant hand. Interaction mode buttons are shown in blue and the remainder in red.

Locomotion in the sandbox is of course possible by simply moving around in the real-world playing area, and it can be aided by standard **arc teleportation** (fig. 4.3). The arc does not collide with objects in the scene and there is no mechanism to "fly around", so the user is always *grounded*, making the modeling sandbox predominantly two-dimensional. This can be either a useful or missing feature, depending on what the user is trying to model. To teleport to another point  $X$  in the sandbox's plane, the user points the controller joystick forward in the direction of  $X$ , adjusts the teleportation distance by rotating the controller up and down to change the slope of the arc, and then releases the joystick. Moving the joystick left and right allows him to rotate his field of view in-place. Both of these mechanics are built into the SteamVR library.

The prototype requires two buttons in each of the two handheld controllers. The dominant hand's controller has two buttons **Accept** and **Cancel** whose functionality depends on the context

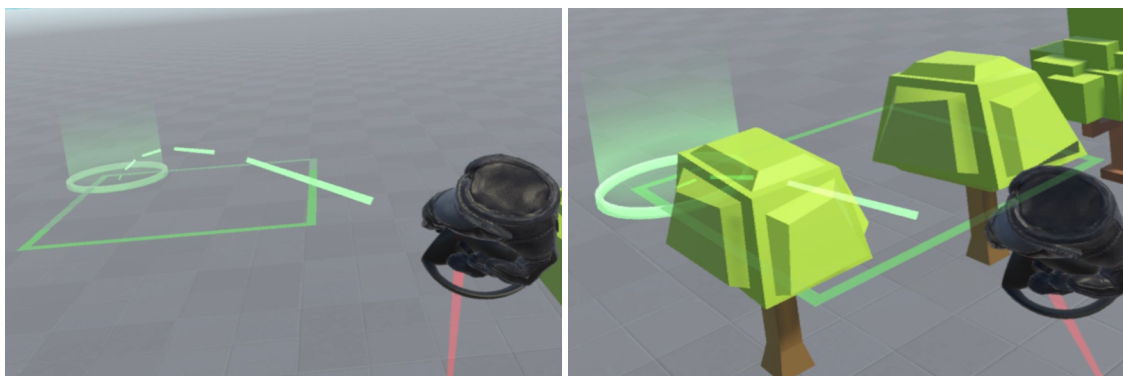


Figure 4.3: Teleportation arc, which does not collide with scene objects.

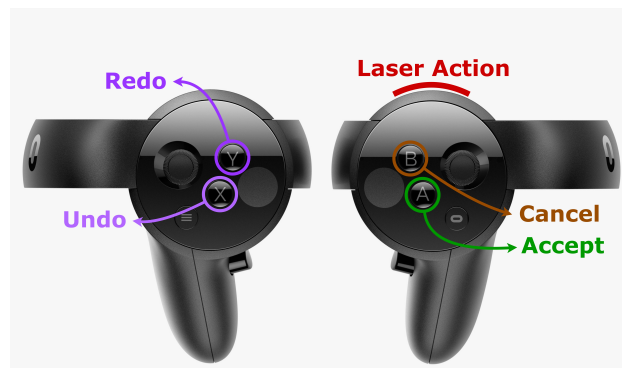


Figure 4.4: Controller button assignments for right-handed users.

(fig. 4.4). While grabbing an object the Cancel button aborts the grab and returns the object to its initial position, and in the other contexts it serves to switch interaction mode. The Accept button is used to create Groups and Randoms and conclude grab operations gracefully. The other hand's controller has buttons for the **Undo** and **Redo** operations which feed into the undo system. These are explained in section 4.3.

Finally there is a whole color-coded **feedback system** to help with keeping track of the application's state, current interaction mode, edit depth, recent operations and the composition of objects in the scene. This is explained in detail in section 4.4.

### 4.1.2 Scene composition

A **scene** initially contains only Primitive objects. In the modeling sandbox a user operates and composes these objects with procedural and composite entities to create their model. All objects in the scene are static – they do not respond to gravity and do not collide with each other or with the user. The lack of gravity and collisions makes the sandbox more akin to a desktop modeling application and less natural at first sight, as it requires the users to handle floor alignment and collisions themselves. This is necessary however: the undo system does not cope well with gravity messing around with object transforms, and complex procedural and composite objects such as large Groups whose elements have random positional and rotational offsets do not interact well with neither gravity nor collisions.

The scene model consists of the **object tree** and the **undo history**. The former is just the set of all objects currently in the scene. This can be visualized as a tree where at the root we have a symbolic scene root node, internal nodes are procedural and composite objects, and the leaves are primitive objects (fig. 4.5). The nodes in this tree that are children of the symbolic root node are the top-level objects that the user can interact with in normal mode, and the remainder are embedded as children of other objects. All the operations a user performs in the sandbox that change the object tree model are recorded in the undo history, even basic operations such as object movement and rerolls.

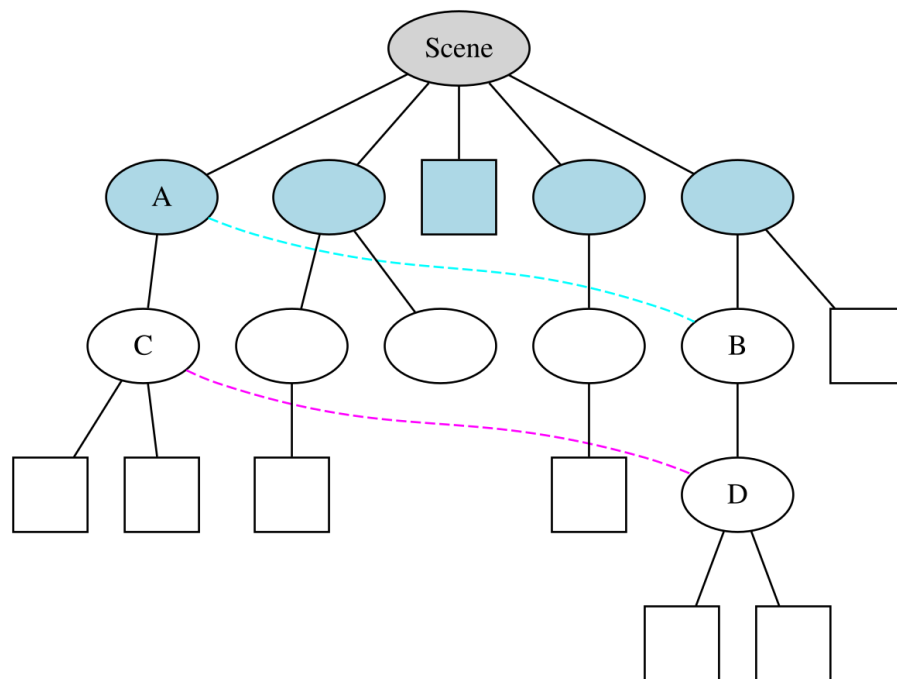


Figure 4.5: Example object tree of a whole scene. The grey node is the symbolic root node, blue nodes are top-level objects, circle nodes are composite or procedural objects, and square nodes are primitive objects. The dashed cyan line shows a *sibling link*, and the dashed magenta line shows an example of an implicit *cousin link* between *C* and *D*, corresponding children of siblings *A* and *B*.

### 4.1.3 Interaction modes

The current interaction mode (or state) dictates primarily what happens when the trigger is pressed while the laser hovers over an object (fig. 4.2). These correspond roughly to some of the modeling operations available and are all listed in table 4.1. The remaining buttons in the menu serve to modify *grab constraints* (section 4.5), scene scale (section 4.4) or in edit mode to apply an operation directly to the current edit subject (table 4.2).

To switch interaction mode the user simply clicks on the appropriate button in the menu. The grab mode is the primary interaction mode used to move objects around. Pressing the Cancel button on the dominant hand controller switches to this mode automatically, and while in grab mode editing an object, the Cancel button serves to conclude editing that object. A hint appears above the controller button in this case. This aims to speed up the workflow since most mode transitions are just returning to grab mode to reposition objects in the scene.

Most of the operations available were already introduced and properly discussed in chapter 3. Groups and Randoms must be created with several children, so naturally the user must perform a selection of elements/variants to form the object. The multi-step protocol we use is to switch to the desired interaction mode, then select the objects upfront by simply clicking on them. They will appear selected by receiving a persistent dark blue outline. Clicking again on them deselects them. When the selection is good the Group or Random are formed by pressing the Accept button.


















Icon	Mode	Context	Trigger event
	Grab mode	Normal mode	Attach the object to the laser, and keep it attached until the trigger is released.
	Grab mode	Edit mode	Also attach the object or utility handle to the laser. This also moves all <b>cousins</b> of the grabbed target or utility handle, and modifies all siblings of the edit subject.
	Edit mode	Any mode	Enter edit mode on the object.
	Create Group	Normal mode	Add or remove the target object from the <i>selection</i> of elements for the Group. The Group can be formed through a controller button when 1 or more elements are selected.
	Create Random	Normal mode	Add or remove the target object from the <i>selection</i> of variants for the Random. The Random can be formed through a controller button when 1 or more variants are selected.
	Create Tiling	Normal mode	Wrap the target object with a Tiling and immediately enter edit mode.
	Create Mover	Normal mode	Wrap the target object with a Mover and immediately enter edit mode.
	Create Rotator	Normal mode	Wrap the target object with a Rotator and immediately enter edit mode.
	Clone	Normal mode	Clone the target object and immediately grab it.
	Reroll	Normal mode	Reroll the target object recursively.
	Disband	Normal mode	Disband the target object once (not recursive).
	Unlink	Normal mode	Unlink the target object from its siblings.
	Delete	Normal mode	Delete the target object recursively.
	Clone element	Group	Clone the target object, which may be either an element of the Group or an outside object, into the Group and immediately grab it. Attempts to create cycles are rejected.
	Delete element	Group	Delete the target element. Attempts to delete the last element are rejected.
	Clone variant	Random	Clone the target object, which may be either the visible variant of the Random or an outside object, into the Random and immediately grab it and make it the main variant. Attempts to create cycles are rejected.
	Delete variant	Random	Delete the target visible variant. Attempts to delete the last variant are rejected. The next variant in cyclic order is made visible.

Table 4.1: Interaction modes









Icon	Operation	Context	On click event
	Reroll	Group	Reroll <i>only</i> the elements recursively.
	Cycle	Random	Make visible the next variant in cyclic order.
	Add Tile	Tiling	Add one tile clone and adjust the spacing.
	Remove Tile	Tiling	Remove one tile clone and adjust the spacing. Cannot remove the last tile clone.
	Refresh	Mover	Reroll <i>only</i> the random offset of the child.
	Refresh	Rotator	Reroll <i>only</i> the random rotation of the child.

Table 4.2: Edit operations through edit mode menu buttons

Pressing the Cancel button dismisses the selection and returns to grab mode. Creating the other objects is straightforward, and as discussed in chapter 3, edit mode is entered immediately.

## 4.2 Editing objects

In this section we explore the interaction mechanisms that allow the user to edit objects while resorting only to natural interaction techniques and no need for numerical input, menu sliders or other complex input mechanisms generally found in a desktop modeling application.

We start by recalling that the object tree has out-of-tree links connecting objects to their siblings. These were created on Clone operations and ensure cloned objects remain *in sync* with their original objects and vice-versa. Edit operations performed on an object  $X$  must be understood as applying to  $X$  and all of its siblings in the tree, simultaneously.

In edit mode only the children of the edit subject can be moved or modified. Any attempt to move or modify an object outside the subject's object tree is rejected with an info message through the feedback system.

When editing an object, a set of auxiliary objects we call **utility handles** appear. These handles were introduced in chapter 3 and are used to set the parameters of the objects. A set of three non-interactive RGB orthogonal axes also appear crossing on the edit subject's center of mass. These are classic modeling axes, and serve to show what the local space is while editing the object. Formally, what we call the "center of mass" of the edit subject is just the origin of its local space. When a Movers, Rotators or Tilings are created they still their center of mass is set to be the same as their child. When a Group or Random is created the center of mass is set to the first element or variant selected.

The Group and Random objects have no utility handles, only spawning the axes in edit mode. When editing a Group (and similar for a Random) the user can change the relative position of that Group's elements by simply grabbing one of them in grab mode. If the Group subject has  $n$  siblings, then the element grabbed has  $n$  cousins, and these cousins are applied the same movement

in their own reference frame. For example, if the user grabs an element and moves it from location  $X$  to location  $Y$  in the edit subject's local space, then the element's cousins also move from  $X$  to  $Y$  in their respective local spaces, and this movement can be seen without any delay. The cousins in this context are *grab slaves*, which have a pink outline in the scene, see fig. 4.6.



Figure 4.6: Slave grab example. There are three groups of three trees each, all siblings. The user is editing the closest group. Grabbing the element in the middle (outlined red) causes the two slaves (outlined pink) to move in sync.

The Mover object has the simplest set of handles (fig. 4.7). There is a *transparent parallelepiped* that is aligned with the local axes and surrounds the center of mass. The center of mass of the Mover's child is chosen uniformly at random from within this box. A circular handle is attached to the corner of this box and it can be grabbed like any other object, in such a way that moving the handle adjusts the dimensions of the box. When the handle is released, a new random offset for the child object is chosen from within the box. The translation applied to the handle is being replicated across all siblings of the edit subject, but this is not visible to the user since the siblings' handles are not made visible. A new random offset is also picked for the child object's cousins when the handle is released, and this change the user can indeed see.

The Rotator object allows the user to set the axis of random rotation and the permissible angle by moving one of three circular handles around three RGB tori surrounding the orthogonal axes (fig. 4.8). The orthogonal axes are hidden for this class of object. The circular handles are attached to their respective torus, the user can only move the handles along the curve defined by the torus circle. Initially every handle rests at the  $\pm 0^\circ$  angle position in their torus, indicating no random rotation around this axis. When a handle is grabbed a *transparent circular sector* appears that is centered on the edit subject's center of mass and has an angle defined by the handle's position on the torus. The random rotation of the object is then picked from the set of angles defined by this circular sector. The child object can only rotate randomly around one of the axes, so whenever a handle is grabbed, the other two are set back to their rest positions at  $\pm 0^\circ$ . Just like the Mover, when a handle is released the modification is propagated to the edit subject's siblings, resulting in a new orientation being visibly applied to all of them.

The Tiling object has two handles (fig. 4.9). The first is yet another circular handle that behaves like the Mover handle and sets one of the ends of the tiling range, the other end being the symmetric position about the center of mass. The handle is locked to the  $x$ -axis, so the tiles are always aligned along the  $x$ -axis and centered in the subject's center of mass. While a handle is moved, the spacing

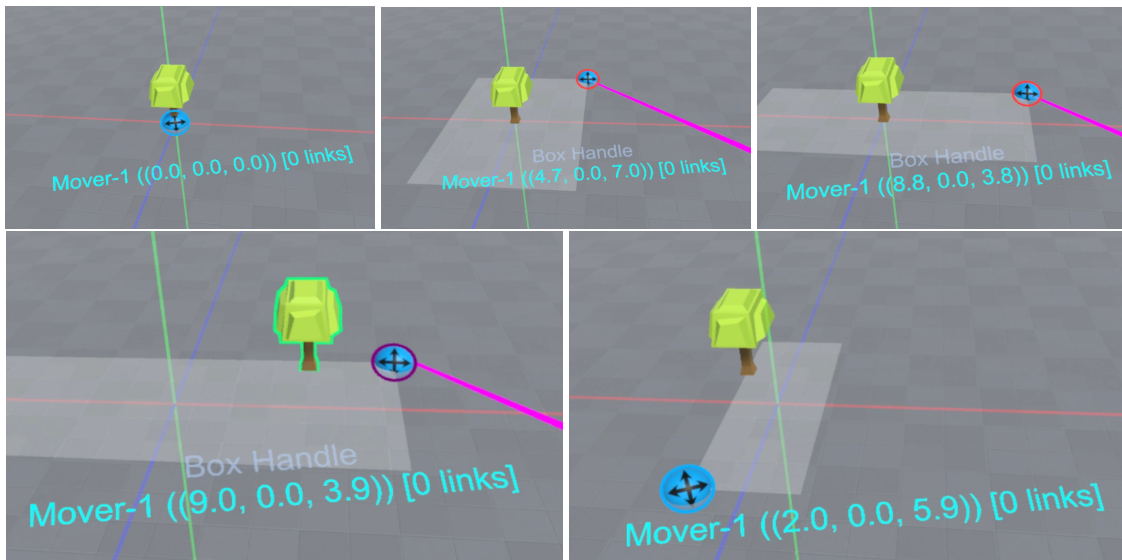


Figure 4.7: Mover editing example. The handle is constrained to move only horizontally. In the fourth panel the user released the handle, causing a new random offset to be chosen for the tree.

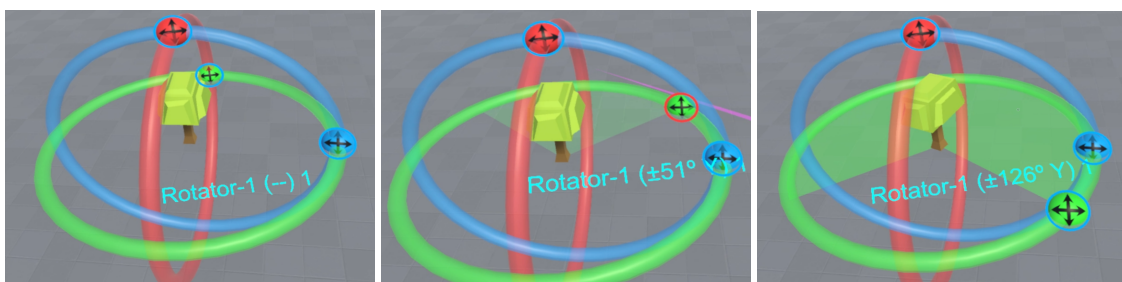


Figure 4.8: Rotator editing example. The handles are constrained to move along their torus. In the third panel the user released the handle, causing a new random rotation to be chosen for the tree.

is adjusted immediately, for the current subject and its siblings. A *rotating capsule* is fixed in place above the tiles but can be grabbed and rotated, and the tiles rotate along with the capsule; this grants the ability to reorient the tiles a different way. Once again the orientation is adjusted for the current subjects and its siblings without delay.

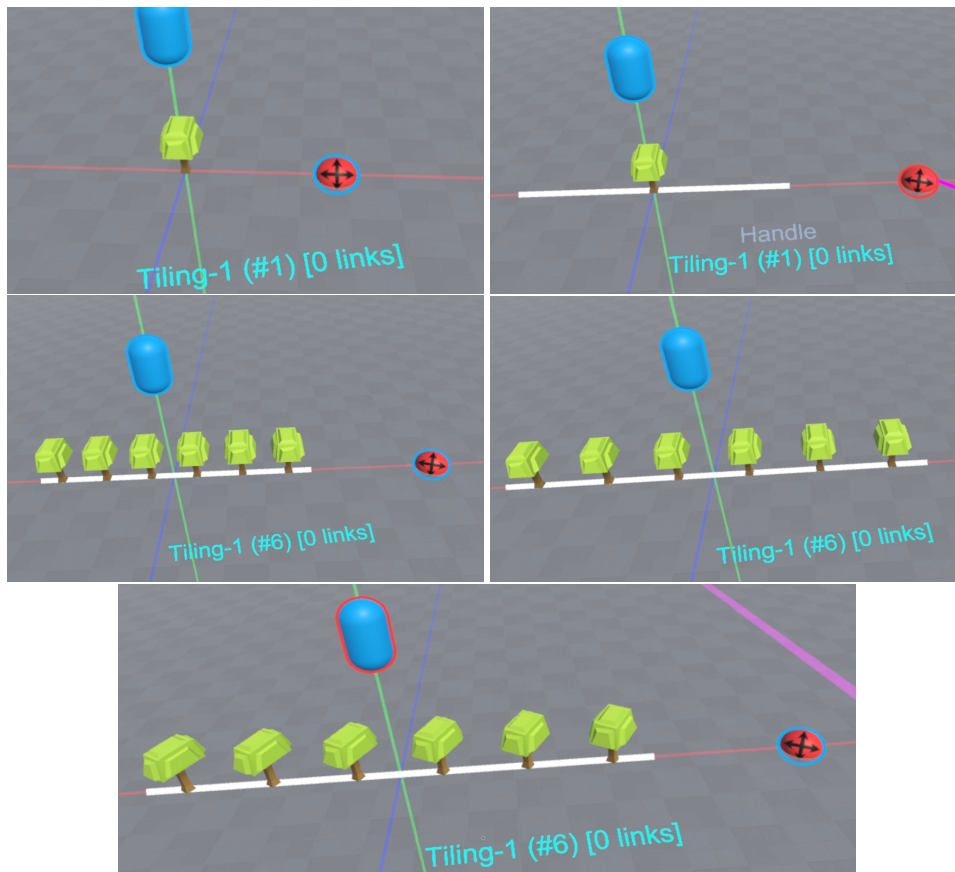


Figure 4.9: Tiling editing example. The handle is constrained to move only along the red  $x$ -axis. In the last panel the capsule is being grabbed (note the red outline), and it rotates in-place.

Early in development the Tiling object had two handles to set the tiling direction, but this quickly proved to be unnecessarily complicated while not actually offering any new capabilities to the system. We switched to the simpler system of using just one handle on one of the ends, and require the Tiling itself to be rotated to change the tiling direction. This made the interaction less cumbersome, and also made it straightforward to compute offsets for the circular handle and the capsule to make sure they don't overlap the tiles.

All the above utility handles have **prioritized collision** (fig. 4.10). If an object stands between the controller and the handle along the laser's path, the laser collision algorithm skips the object to collide with the handle. The utility handles are also always outlined, which reveals them even if they are occluded by some object between them and the user camera. This is necessary to accommodate situations where the handle is surrounded by a large object, for example just after a Mover is created and the handle placed at the Mover's center of mass, right in the middle of the large object.

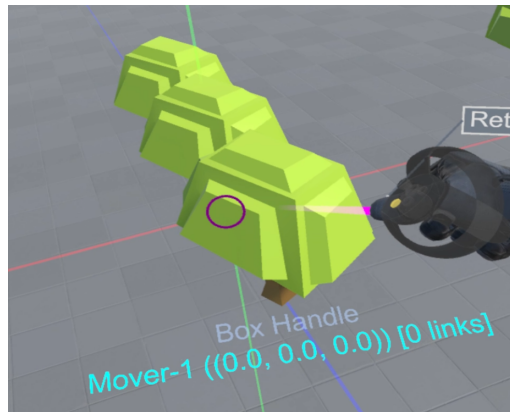


Figure 4.10: The laser skips the tree to collide with the Mover's handle.

### 4.3 Undo system

As we have seen the undo history is part of the model of the scene – it is kept coupled to the object tree model itself. The undo history maintains a stack of *modifications* applied to the object tree model including creation of new objects, object translations, clones, rerolls, disbands, unlinks, deletes and so on. User operations map directly to one or more modifications. Every operation performed by the user can be undone and redone, but the history is *linear* and there is no support for time-travel such as copying past objects into the present or vice-versa. The history being linear means that, if the user performs an Undo operation from the present  $n$  times, and then performs some new operation, then all  $n$  operations currently undone are permanently *forgotten* and cannot be recovered, and then the new operation is appended to the undo history. This contrasts with a branching model where a new branch of operations would be created, and the old one somehow preserved and still accessible. Conversely, if the history becomes very large, then the oldest operations could be *committed* and removed from the front of the history, but we did not need this optimization when testing the prototype.

A *modification* in this context does not map directly to a user action; instead, a single user action can generate several modifications in a well-defined order. For example, cloning an object recursively rerolls all of its descendants, potentially causing several procedural objects to make new random choices. These random choices are of different kinds – a Random object rolls a dice and picks a new visible variant, while a Mover object picks a new 3d vector offset. Each random choice is separate, and these modifications are packaged into a single *transaction* and appended to the undo history.

The simple act of moving an object from position  $X$  to position  $Y$  is itself a user operation that is recorded by this system. This is implemented by recording, when the *grab* begins, the initial positions of the *target* and all other move subjects, which in edit mode can include cousins of the target. Once the user releases the target, the final position  $Y$  is computed and an operation of moving the object from  $X$  to  $Y$  is recorded into the history for each moved object and packaged into a single transaction.

There are two obvious undo and cancel modeling patterns that let us deal with potentially messy situations. Suppose we intend to clone or grab an object *A*, but this object is either very small or quite far away, so we might accidentally grab some other object *B* nearby. We can immediately abort the clone or grab by pressing the Cancel button and try grabbing *X* again, or we can drop *Y* and press the Undo button. The users who tested the application appeared to converge on the later. Now suppose we apply some operation to an object, but we were distracted and it was not the operation we intended. We can immediately cancel the operation with the Undo button, and the operation actually performed is still described in the live feed for about three seconds after the undo. The operation intended was probably a grab, so we can press the Cancel button to switch to grab mode.

Because performing undo and redo operations is so accessible, it should also be **safe**, so that users may perform them without actually remembering what they will do. Users cannot perform undo or redo operations while grabbing objects. Performing undo and redo operations in edit mode poses a considerable complication: the undo or redo operation, if unchecked, can cause the current edit subject to disappear or change position. We definitely want to allow a user to use the undo system while editing objects, so this must be addressed.

To handle this issue without having to resort to introspecting the undo/redo operation itself, we introduce a hidden *undo window* when the user enters edit mode. There is one window for each node on the edit path. Initially the window is empty and anchored on the current history pointer. Undo and redo commands are not allowed to move the history pointer outside of this window. When the user performs a new operation in this edit mode, the end of the undo window is extended forward to the new history pointer. When the user exits from a recursive edit the end of the undo window is adjusted to match. This ensures that the operations performed by the undo or redo command do not modify or delete the edit subject itself, only its children in the object tree, which is what we wanted.

## 4.4 Feedback system

The user feedback and help system consists primarily of color coding, controller hints, the *live feed* and mode descriptions.

Each type of event and operation in the sandbox is associated with a color, and these are listed in table 4.3. The laser color is reactive and maintains the color best matching the current interaction mode. Events logged as text to the live feed appear in the associated color (fig. 4.11). After certain modeling operations an outline in the associated color persists on the object for about two seconds, to provide the response that the operation was indeed performed.

Controller hints appear above the four buttons in the two handheld controllers when the user looks at the controllers, and provide a hint of what action they will perform when pressed, such as creating a group or canceling a grab.

The *live feed* (fig. 4.12) is a very simple text log that appears fixed and over other objects just below the center of the user's field of view. It shows three pieces of information:





Figure 4.11: Feedback for various operations. In the first panel the user is editing a group and attempted to grab an object outside of the group. In the second panel the user just deleted the middle tree. In the third panel a Group with the three trees was disbanded.

- A short description of the currently hovered object, button or handle, in gray. If this is a button it shows its name or a short description. If this is a handle it gives the handle's name. If it is a Primitive object it shows the asset name. Otherwise it is a composite object, and it shows **introspective** information about that object.
- The name of the current edit subject, in cyan, and similar introspective information.
- A log of the latest modeling event (or error) in the scene, such as "Deleted Group 73" or "Created Random 37".

The introspective information mentioned includes the number of elements and variants in a Group or Random, the offset range in a Mover, the angle range and axis of rotation in a Rotator, and the number of tiles in a Tiling, plus in all cases the number of siblings the object has.



Figure 4.12: Live feed. In the first panel the feed shows all three components, just after a user wrapped a Rotator object (hovered) with a Mover object and entered edit mode.

New objects made through create operations (not clones) are given increasing numeric labels, which persist when cloned. So the first Group the user creates is "Group 1", then "Group 2", then "Group 3" and so on. Finally, a short description of the current interaction mode is shown on top of the dominant hand controller if the user looks over the controller.

Color	Context, events, operations
Yellow	Hovered objects. Outline color given to objects, handles and buttons being hovered by the laser (or hands). Also the color of the text in the live feed that describes the hovered object, button or utility handle.
Red	Moving objects. The grab target is outlined red.
Blue	Creating objects. Outline color used for the selection of elements and variants when creating Groups and Randoms.
Cyan	Editing objects. Also the color of the text in the live feed that describes the current edit subject.
Green	Cloning and rerolling objects.
White	Unlink and Disband operations.
Black	Deleting objects.
Magenta	Movement slaves. Outline color given to the cousins of a Group element or Random variant while grabbed in edit mode.
Orange	Warnings, errors and rejected operations. Outline color used to identify the node that causes a cyclic inclusion, to remember the user he can only operate on children of the edit subject while in edit mode, among other things.

Table 4.3: Feedback system color codes

## 4.5 Miscellaneous

In this section we delve into some features of the prototype which are generic modeling tools not directly imposed or expected to exist by our modeling approach, but are necessary for a complete modeling experience nonetheless.

### 4.5.1 Grab constraints

Utility handles have various types of movement constraints: the Tiling handle can only move along a straight line, the Rotator handle can only move along a curve, and the Tiling capsule is literally fixed in place. Coupling this with the fact that our prototype is mostly two-dimensional, we took the opportunity to allow the user to apply movement constraints on their own, similar to the way a desktop modeling application can restrict movement of models along lines and planes, or rotations around an axis, through various transformation tools available.

The *grab constraints* available are listed in table 4.4. The constraints compose naturally and are applied to any objects grabbed and also to the Mover's box handle. The constraints can be toggled through buttons in the floating menu, even in edit mode, but cannot be toggled mid-grab.

For user testing we used a different set of menus without all of these constraints to not overwhelm the volunteers, to keep focus on the high-level interaction and modeling processes without putting an emphasis on rigor and detail. Each test scenario has only one of these constraints available in the menu, and the rest are set to values appropriate for that scenario.








Icon	Constraint	States	Functionality
	Grid Snap	Off Gridpoint Gridline	In Gridpoint state, have objects grabbed <i>snap</i> to the closest point on an invisible three-dimensional lattice that overlays the entire modeling sandbox. In the Gridline state, have the objects snap to the closest axis-aligned line connecting these points instead. For example, if the lattice aligns with the squares of a chessboard exactly, this can aid in placing the chess pieces exactly in the center of a square.
	Plane Lock	On or Off	Have the object move only along the horizontal plane (in local space) and prevent it from moving vertically. The horizontal plane is determined as the one passing through the object when it is initially grabbed. This changes the way the object remains attached to the laser: instead of remaining at a fixed distance to the controller, the attached object is projected to the horizontal plane along the laser, allowing far away objects to be moved right next to the user and vice-versa. This is a rather important mechanic, without it object movement can be imprecise and cumbersome over large distances. It is enabled in both test scenarios.
	Rotation Lock	On or Off or X, Y or Z	In the On state prevent the object from rotating. In the X, Y, Z states (axis lock), allow the object to rotate only around this axis (in local space). In our chessboard example we may wish to allow chess pieces to rotate only around the Y axis, or not at all. A rotation lock around Y is enabled in both test scenarios. Unfortunately the conjunction of an axis lock with the previous lock proved to be very difficult to use during testing due to an incorrect implementation.
	Rotation Snap	On or Off	If the rotation is locked around an axis, have the object's orientation snap to multiples of 90° around that axis.
	Vertical Lock	On or Off	Have the object move only vertically (in local space) and prevent it from moving horizontally. The vertical line is determined as the one passing through the object when it is initially grabbed. Similar to the Plane Lock grab this changes the way the object remains attached to the laser, as it gets projected to this vertical line.

Table 4.4: User controllable grab constraints.

### 4.5.2 Edit augmentations

As mentioned in section 4.2, all entities but the Rotator show the three orthogonal axis in local space while they are being edited, and the Rotator shows three orthogonal tori instead. These augmentations are static and non-interactive, their only purpose is to be a visual aid for the user.

The circular utility handles have a fixed scale that is independent of the dimensions of the edit subjects and their child objects; the scale is proportional to the "size" of the player within the sandbox. The circular handles are about the size of a closed fist. The tori have a dynamic outer radius that is dependent on the dimension of the Rotator's child and is computed when edit mode is entered, and a fixed inner radius of about half the radius of a circular handle.

### 4.5.3 Global scaling

Finally, the floating menu contains two buttons (see fig. 4.2) to scale up and scale down the entire scene relative to the size of the user character in the sandbox (or equivalently, to scale the user character itself). The scene is scaled around the current position of the user. When scaled while editing, the utility handles and the Rotator's tori automatically rescale themselves since their world-space scale is fixed.

## 4.6 Implementation architecture

In this section we give a small overview of the actual implementation of the system in Unity.

The architecture of the prototype, roughly Model-View-Controller (MVC) based, is sketched in diagram 4.13. In summary, a central controller, the **Interactor**, receives user input commands from the virtual reality controllers and processes them according to the current interaction mode, relaying them to the other controllers. The two View modules react to changes in the model and to new events in the events log queue.

Core object creation and operations are relayed to an Interaction Controller. This controller performs all user operations described in chapter 3 with proper validation. These include creating new procedural and composite objects from templates and a selection of children, handling recursive object cloning and linking, detecting attempts to create cyclic inclusions, searching for siblings and cousins to relay edit operations to, and setting up on-grab-release and on-grab-cancel events. This controller generates *commands* for validated updates, which are applied to the current object tree model and pushed as commands to the current history transaction in the undo history.

The object tree model is implemented directly using Unity's *GameObject* tree hierarchy. Each type of object is represented by its own class, and an abstract *Procedural* class is the base class of all objects. Each of these classes knows how to *reroll* itself and what to do when entering or exiting edit mode. The utility handles that are visible in edit mode are persistent and made visible when the object is being edited.

The composite and procedural objects themselves have no associated mesh, so highlight events applied to these objects must be *propagated* down their tree. A more general abstract *Interactive*

class serves to identify *collidable* entities, handles hand/laser collision with menu buttons and highlight propagation from composite objects down to the primitive objects. If the collision algorithm initially detects a hit with a primitive object (otherwise a button), it then propagates the collision to the parent objects for as long as the current object is not present in the current **edit path**, which comprises all objects on the root path of the current edit subject. For example, when pointing the laser at one of the elements of a top-level group with three primitive elements, the laser is considered to collide with the group, but if the group is being edited then the laser is considered to collide with the primitive object itself. In the former case the yellow outline is applied to the group and propagated down to its three children; in the later case the outline is applied to the primitive object, not to the group.

The undo history is part of the model as well. Each scene has its own history, so this module is *hot-loadable* along with the scene itself. Here we use the classic command design pattern for undo systems based on encapsulating modifications as an interface or abstract class with four functions called Undo, Redo, Commit and Forget. This pattern has drawbacks, namely a very large amount of boilerplate and the unnatural requirement to frame every modification as an object instance, but it was well suited for our prototype.

Standard object movement is handled through the Grabber controller. Movement is not a trivial task due to several self-imposed requirements such as local space movement while editing objects, undoable movement operations, live feedback while moving objects with cousins in edit mode, *constrained movement* along grids, rotations locked to one axis and utility handles moving along arbitrary but fixed 3d surfaces and curves (tori, planes and lines) in edit mode. A more reactive alternative to this design makes these constraints much harder to conjugate.

The input controller is a simple wrapper around SteamVR's own input controllers that relays all input from the virtual reality controllers as commands to the Interactor. This module also tracks the laser and hand collisions frame-by-frame and relays changes as events.

The user interface module consists mainly of a button-only menu that hovers over the user's non-dominant hand and responds to input from the dominant hand's laser. There are several menu templates available since a different, more restricted set of operations is available while in edit mode. Which menu is shown is determined reactively based on the current interaction mode in the Interactor. As such there are several instances of each type of button, spread out through the menu templates.

Finally, the feedback system, explained in section 4.4, is completely reactive; it was implemented near the end of development with the goal of alleviating the learning curve for users. It helps keep track of the current application state, provides higher fidelity for operations and introspection of objects in the scene. On each frame, the view fetches the current edit subject, the current hovered object, and the most recent action, and provides a short description of each of these in the hover log just below the center of the user's field of view. This system also adjusts laser color as a function of the current interaction mode and adds action hints to the controllers when appropriate – such as "Create Group" or "Cancel Grab".

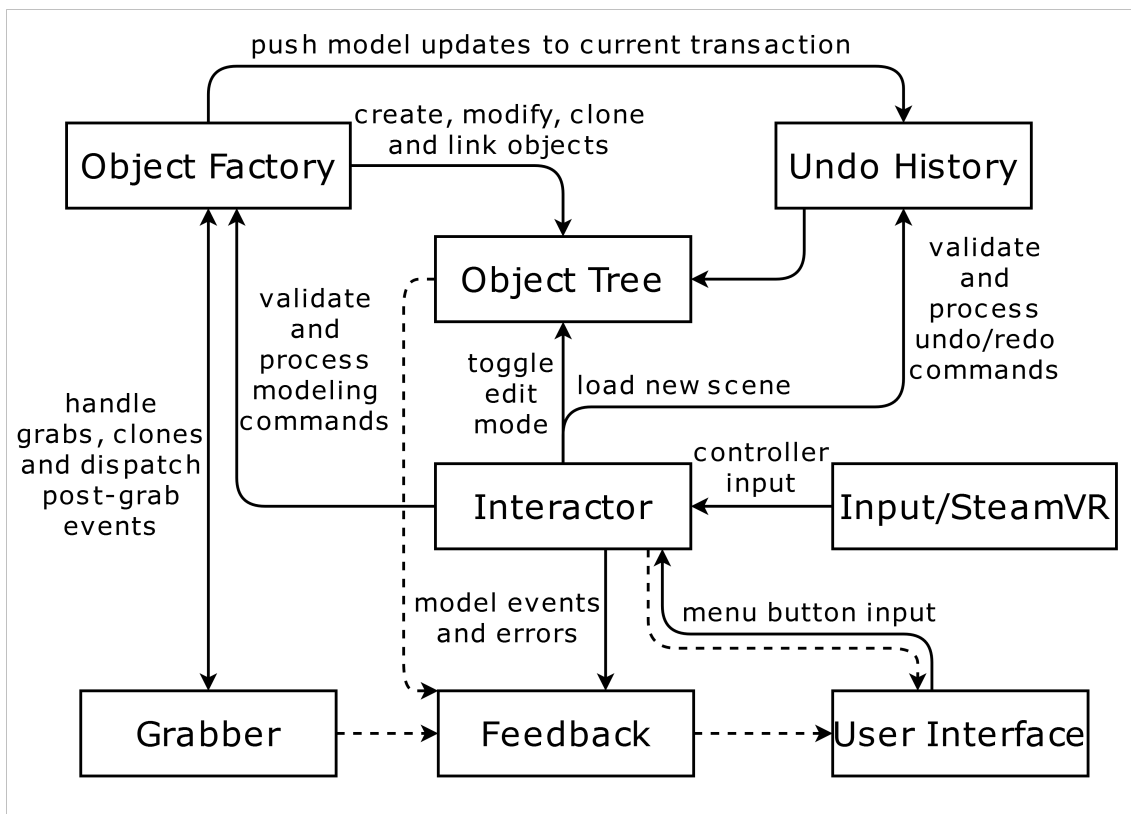


Figure 4.13: Overview of the control flow in the implementation.

## 4.7 Modeling patterns

To conclude this chapter we list some usage patterns involving one or more of the tools available which help achieve higher modeling efficiency, precision, correctness and expressiveness. Some of these patterns were already introduced in chapter 3.

**Discrete random offset** A very important feature of the Random object is the ability to lay out the variants in different positions and orientations. Suppose we have a lamp object  $L$  which we wish to place either on the left or on the right of a desk, independently of the other objects on the top of the desk. We can do the following:

- Clone  $L$  to get two lamp objects  $L_1$  and  $L_2$
- Position  $L_1$  in the center of the table facing forward
- Create a Random  $R$  with  $L_1$  and  $L_2$ .  $R$  gets centered on  $L_1$ 's position
- Edit  $R$  and move the lamp  $L_1$  to the left corner of the desk
- Cycle to the next variant  $L_2$
- Move  $L_2$  to the right corner of the desk.

The result is a random object  $R$  which shows the lamp  $L$  on the left or right of the desk with  $1/2$  probability each.

**Layout variants** A more elaborate version of the previous pattern, suppose we have several models  $M_1, \dots, M_k$  we intend to place on the top of our desk, but we want them to be laid out in one of several layout variants, chosen randomly. For example, one with the computer monitor on the left and keyboard on the center of the desk, another with both on center, another with both on the right of the desk, etc. We can do the following:

- Create a Group  $G_1$  with all the models  $M_1, \dots, M_k$
- Clone it to Groups  $G_2, \dots, G_n$  for  $n$  layout variants
- Unlink the groups from each other
- For each group  $G_i$ :
  - Move  $G_i$  to the center of the table. The center of the group is the center of  $M_1$ .
  - Edit  $G_i$  and lay out the elements as desired for the  $i$ -th layout.
- Position  $G_1$  on the center of the table, then create a Random  $R$  with groups  $G_1, \dots, G_n$ .

The result is a random object  $R$  which shows each layout with  $1/n$  probability.

**Biased random choices** There is nothing preventing a Random object  $R$  from having several instances of the same variant (in other words, having sibling variants). In particular there is nothing preventing it from holding several Empty objects. So by adjusting the ratio between the number of variants we can create biases towards certain shapes. For example, instead of showing a laptop or a desktop monitor with  $1/2$  probability each, we can introduce a bias to show the laptop with  $4/5$  probability by including four copies in the random  $R$ .

**Random circular offset** Imagine we have a circular desk whose orientation is fixed, but we have a chair  $C$  we would like placed at a random position around this desk. Do the following:

- Create a Group  $G$  with only  $C$
- Edit  $G$  and move the chair  $C$  away from the center of  $G$  by the radius  $r$  of the desk
- Wrap  $G$  with a Rotator  $A$  (enters edit mode)
- Give  $A$  an angle range of  $\pm 180^\circ$  around the  $y$ -axis
- Place  $A$  at the center of the desk.

This works because the rotation offset is applied to  $G$  instead of the chair. Note that this pattern can result in unusual interaction: grabbing  $G$  snaps the center of  $G$  to the laser, not the center of  $C$ , so when  $G$  is moved the chair is not *on the laser*. It is possible to get other random offset topologies like side of squares and cubes with similar, although more elaborate, techniques.

**Matrix tiling** Recall the chessboard example from chapter 3. An  $n \times m$  matrix tiling pattern of a tile  $M$  can be easily created by composing two Tilings together (fig. 4.14), as follows:

- Wrap the object  $M$  with a Tiling  $T_1$  (enters edit mode)
- Set the desired tile spacing and Increment the tile count of  $T_1$  up to  $m$ .
- Conclude editing  $T_1$ , then wrap it again with another tiling  $T_2$  (enters edit mode)
- Use the capsule to rotate the tiles  $90^\circ$  so they form an orthogonal matrix
- Increment the tile count of  $T_2$  to  $n$  and set the desired tile spacing.

There are many other patterns here: we can form oblique layouts if we rotate the tiles less than  $90^\circ$ , we can have the tiles  $M$  imperfectly aligned by wrapping them with a Mover or Rotator, and we can have some of the tiles disappear by using a Random with  $M$  and many Empty variants in place of  $M$ . Composing all of these with yet another wrapper  $T_3$  lets us make three-dimensional matrix tilings to represent things like balloons. The dimensions  $n$  and  $m$  can be adjusted by editing  $T_1$  and  $T_2$ . The tiles are all cousins, so the tiling remains rectangular if a tile of  $T_2$  is edited to modify  $m$ .

#### 4.7.1 Sample workflows

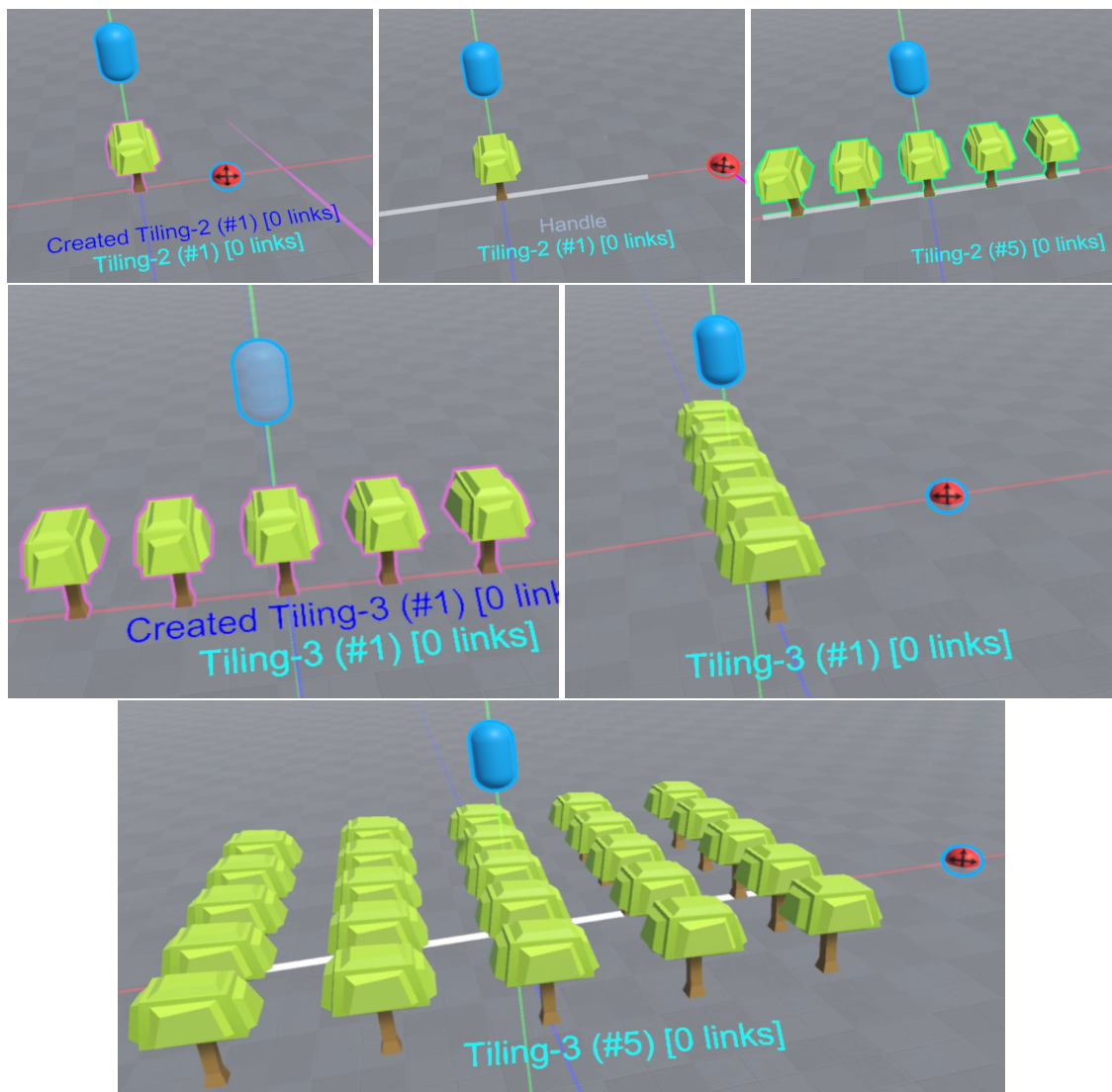


Figure 4.14: Realization of the *matrix tiling* pattern to create a grid of tree primitives.

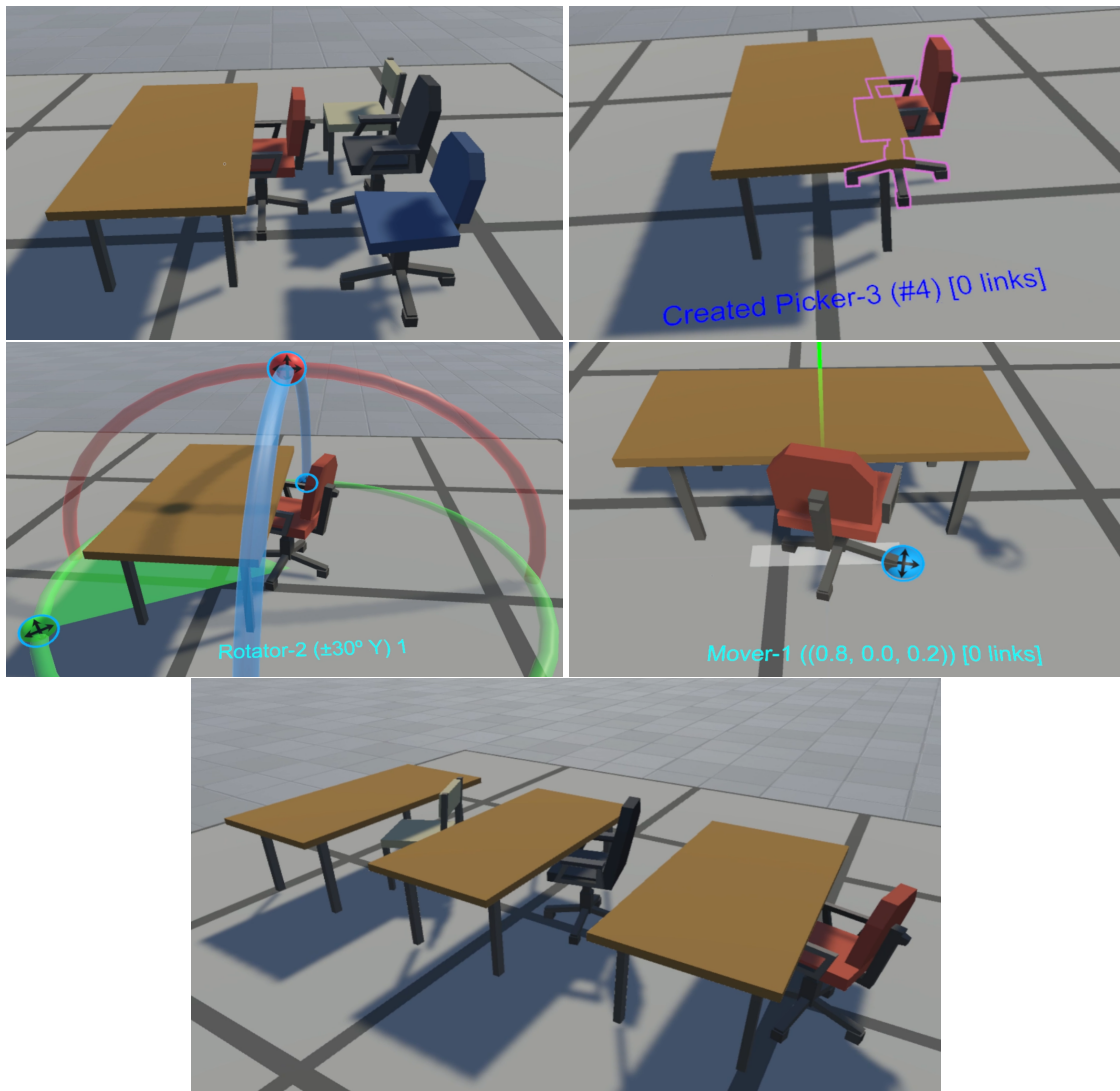


Figure 4.15: Realization of the workflow proposed in chapter 3 to create the imperfectly aligned, randomized chair prop.



## Chapter 5

# User Evaluation

In this chapter we present the user evaluation conducted to both assess the prototype's viability and usability and compare our modelling approach with a simple *baseline* consisting only of the core subset of objects and operations. We present the evaluation methodology used, the results we obtained and what conclusions follow from them.

### 5.1 Testing methodology

Once the development of the prototype was concluded and the evaluation scenarios prepared, we reached out openly for volunteers from within the University of Porto. No prior experience with virtual reality devices and applications, modelling applications or programming was required. We held test sessions at times chosen by the volunteers themselves.

#### 5.1.1 Testing environment

All volunteers used an Oculus Rift headset during testing. Sessions were expected to last between 60 and 70 minutes. Participants without any prior virtual reality experience were given an informal and gentle introduction in the Google Earth VR demo for about 15 minutes before the session proper. This included an introduction to laser mechanics, the virtual reality controllers, and quite simply gave inexperienced volunteers an opportunity to adjust and settle in this new virtual environment. All participants agreed to continue with the session after this introduction.

Before the session began, participants were given a standard consent form, informing them they could interrupt or abort the session at any moment, and that their execution of the tasks within the modeling sandbox would be recorded anonymously for later analysis.

### 5.1.2 Tasks and scenarios

We evaluated two modelling approaches: our *procedural* approach, with all modeling operations and objects available, and a *baseline* approach that restricts the set of objects and operations available to only grouping, cloning, deleting and disbanding objects. The baseline approach is implemented simply through a floating menu with fewer interaction modes, shown in fig. 5.1. With this menu it is not possible to edit objects and there are no randomization tools available. It is still possible to use the undo system and there is still feedback for all operations performed.

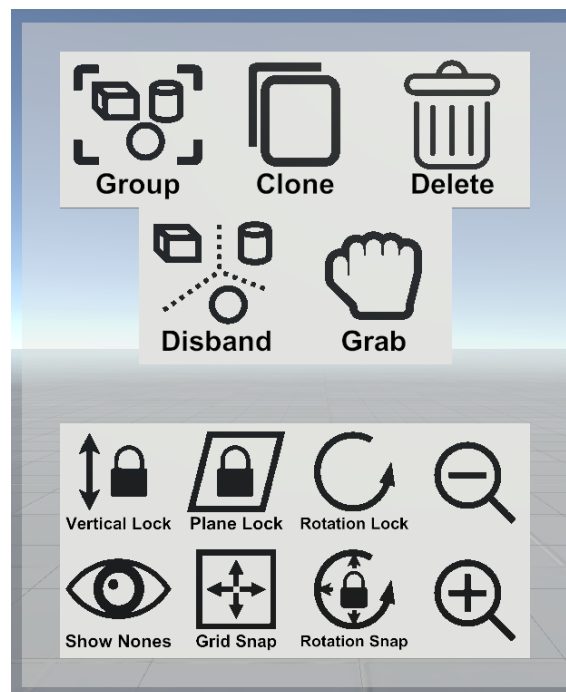


Figure 5.1: Restricted baseline menu.

Our primary goal in this study is to identify whether the procedural approach grants a significant speedup in the development of models with moderate complexity. Hence, each session is split into two segments. In one segment the participant performs two tasks using the baseline menu, and in the other segment uses the full procedural menu. Approximately half of the participants began the session with the baseline approach and half with the full procedural approach to prevent bias in either direction.

In each segment the participant has to assemble **two scenarios**, for a total of **four tasks** for the whole session. In the first scenario the volunteer is asked to assemble a **computer room**, and in the second scenario to a **town center**.

In the first scenario the user is given a *library of assets* (primitive objects) consisting of objects that can often be found in an office or classroom, including tables, chairs, computer monitors, keyboards and a laptop, mousepads, notebooks, pencils, paper stashes, desk lamps and a few more. The user must assemble the room over a designated area right next to the library (fig. 5.2). The room must meet the following specification:

- There must be exactly 6 desks in the room, no particular layout required.
- Each desk must have one chair in front of it, one computer set and at least three other assets on top of it.
- The computer set must be either a laptop with no keyboard, or a monitor with a keyboard.
- Objects should not overlap each other.
- The desks should have different sets of items *and* the layout of these items should vary. This requires varying the chair, tabletop and computer set assets per desk.

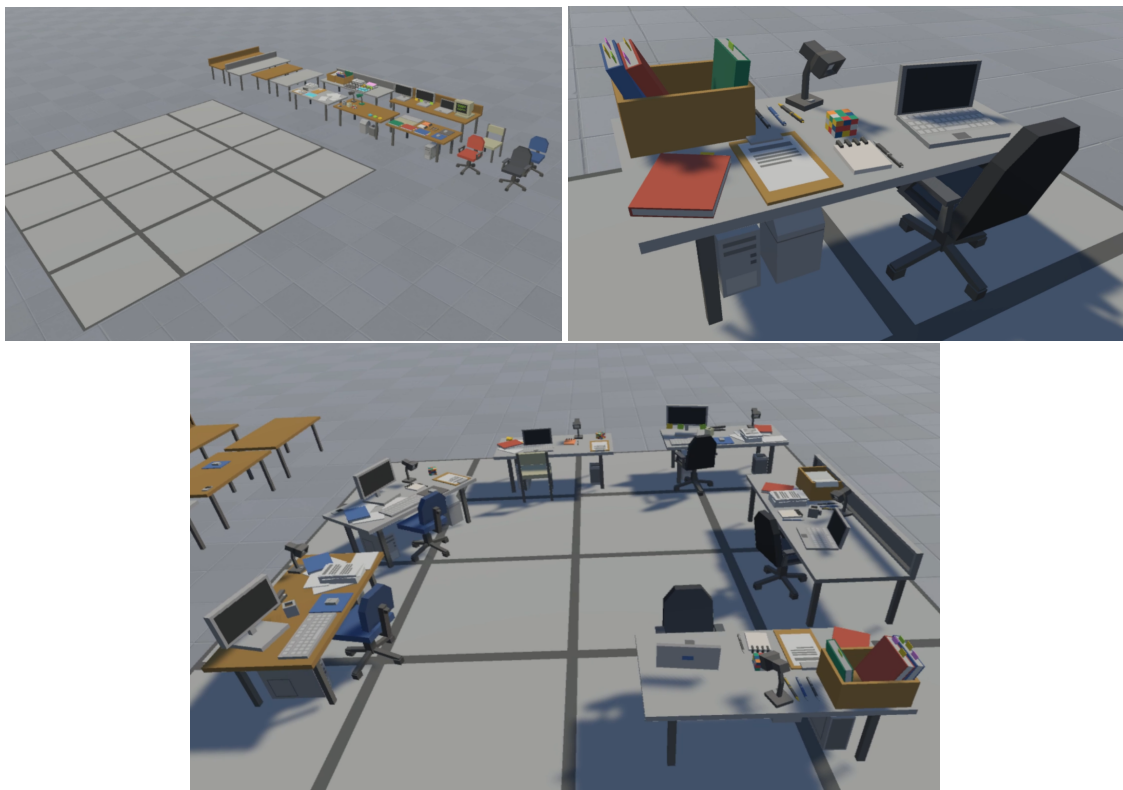


Figure 5.2: Computer room test scenario: the initial scene and asset library, a complete table model, and the final room.

In the second scenario the user is given a similar setup with assets to create a two-dimensional town center including grass, pavement, parks and residential, commercial and office buildings of various shapes and sizes. The user must assemble the town inside a *road skeleton* provided right next to the library (fig. 5.3). The skeleton has 6 empty square blocks, and several buildings fit in each block. The town must meet the following specification:

- All 6 square blocks must be filled with buildings, no particular design enforced. A block is considered filled if the basic grass tile cannot be placed into it without overlapping something else.
- All types and shapes of buildings must be used across all blocks.

- Buildings and parks must not overlap each other.
- The square blocks should have different sets of buildings on them, but the layout of the buildings *does not* have to vary.

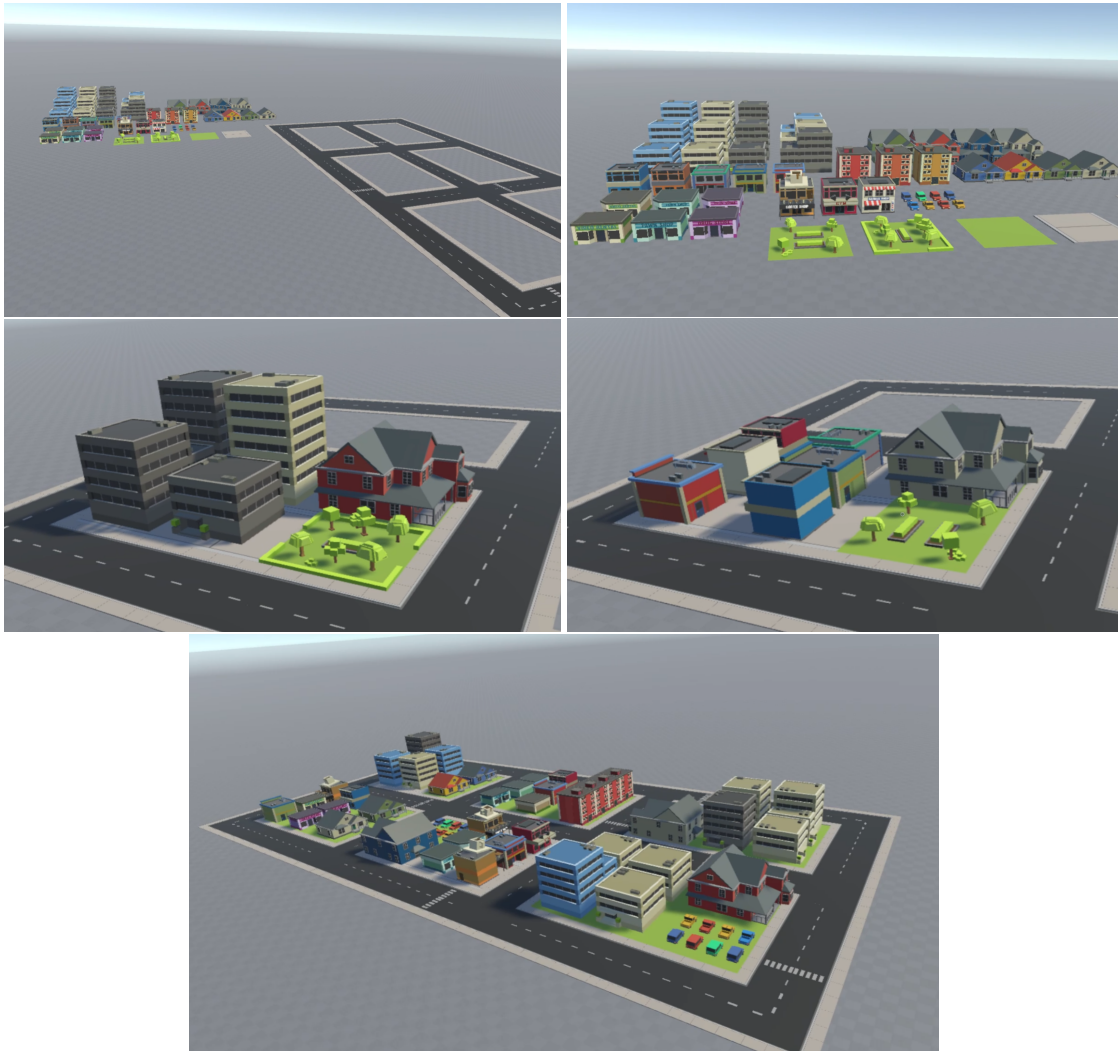


Figure 5.3: Town center test scenario: the initial scene and asset library, complete block models, and the corresponding final town.

The last bullet point in both specifications is the composition/layout **variation** constraint. Participants were informed of both tasks, their specifications, the structure of the entire session, and the nature of both approaches upfront – there were no "surprises". Before beginning each task the participants went through the specifications again.

With the baseline menu a user can assemble the scenarios however he wishes, and he is informed that the goal is to produce the design as quickly as possible. One possible way of doing so is to build each of the six desks/square blocks in parallel. For example, in the first scenario the user can clone six tables into the delimited area, then six chairs, then six computer sets and so on, varying his choices to ensure all the desks are different.

With the procedural menu the user is required to assemble the **unit model** first. In the first scenario this is a desk group that meets the desk specifications, and in the second scenario it is a square block group that meets the block specification, both using procedural objects. The whole scene must then be assembled *only* by cloning this unit model six times, and it must meet the specifications for the whole scenario.

This requires the participant to work backwards, to figure out what structure the unit model needs to have that ensures the clones meet the global variation constraint. With this added restriction the task is considerably more difficult, and the user spends more time thinking and almost all of their time building the desk/block model, but once this model is formed the task is promptly completed.

The final models were not required to look aesthetically pleasing, and most design decisions were left open to the participants. In practice, almost all participants designed the computer room in roughly the same way in both runs, placing the computer in the center of the table, the chair facing the table, and the remaining items either left or right of the computer. In the city task, however, many participants specialized the square blocks with the baseline approach, such as making two residential blocks, two commercial blocks, and two office blocks. This was explicitly allowed, but with the procedural approach (before or after) this specialization required far more effort and complexity, so they instead created a block with a fixed layout and practically no specialization.

Finally, we note that there is a peculiar difficulty in the room task using the procedural approach. It is quite easy to create the computer set model *incorrectly* by aggregating the monitors *and* the laptop together into one Random entity, which would allow the laptop to appear with a keyboard in front of it. There are two correct realizations of this model. One of them is to clone the keyboard in front of each monitor, create three desktop groups, and finally create a Random with these three desktop groups and the laptop. The other involves aggregating just the monitors first.

Since our primary goal was assessing the difference between task completion times, we deemed it proper to inform the users whenever they broke the specification, in particular whenever the computer model they built was incorrect and when they were about to conclude the unit model group but still had some of the requirements missing, such as not having enough items on the table in the first task or not having all types of buildings in the second task. We also reminded them, in the first scenario, that at least some of the items need to have variable placement and orientation; one Rotator and one Mover sufficed on any items except the computer and the table.

### 5.1.3 Tutorial

The users were introduced to the system with a *guided tutorial* in a very simple sandbox with five different trees and a square grass tile (fig. 5.4).

The tutorial has two parts. The first part is a very quick introduction to the basic elements of the system like locomotion, teleportation, grabbing objects, the undo system, and the baseline menu's operations. The user is guided to perform the following steps in order:

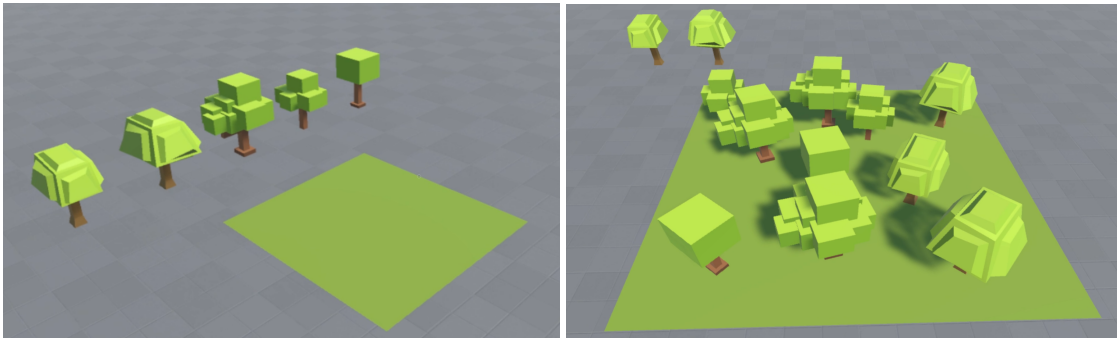


Figure 5.4: Tutorial scenario set. On the right a possible output after the tutorial's first part.

- Use the joystick to teleport somewhere and rotate in-place.
- Use the controller's Trigger to grab and move each of the five trees over the grass patch.
- Use the Undo button to move the trees back to their original place, then the Redo button to move them back to the grass patch.
- Check the menu by turning the palm of their hand upwards.
- Switch to Clone mode and make a line of 10 trees.
- Switch to Group mode and group all these 10 trees.
- Grab this group; see how it feels and where its center is.
- Disband the group.
- Zoom in and out of the scene with the global scaling buttons in the menu.
- Form a 2x2 garden with 10 trees.

The last step is a mini-task that users performed to ensure they had a good understanding of the tools available.

The second part of the tutorial introduces the procedural operations and objects. The Random, Mover and Rotator objects are introduced outside of the virtual reality environment with real-world examples and analogies, alongside the Reroll operation. The user then goes back to the tutorial sandbox and is guided to perform the following steps in order:

- Check the new menu and locate all new relevant buttons.
- Create a Random holding 4 of the 5 trees.
- Switch to Reroll mode, then reroll this Random object.
- Switch to Edit mode and edit this Random object.
- Move the variants around, change variant with the Cycle button.
- Clone the 5th tree into the Random as the 5th variant, then conclude editing.
- Switch to Rotator mode and wrap the Random in a Rotator.

- Move the handles around the tori. Then conclude editing.
- Switch to Mover mode and wrap the Rotator in a Mover.
- Move the handle around. Then conclude editing.
- Clone this Mover object a few times, then edit one of them.
- Move the handle around and see what happens to the clones.

If the participant started with the procedural approach, then the whole tutorial was performed in a single run, in order. Otherwise the second part was performed only after the participant completed the tasks with the baseline approach.

#### 5.1.4 Questionnaire

Before beginning the session participants were asked to fill in a form with their profile information, including age, education and experience with virtual reality or modelling applications. After concluding the first two tasks participants were asked to answer a short questionnaire about their experience in those two tasks, and then again after the last two tasks when the session was finished.

## 5.2 Results

Now we analyze some of the metrics we collected from the evaluation sessions and the feedback we received from questionnaire answers. The primary goal of the study was to determine, for each of the tasks, whether the procedural approach offered an advantage in terms of modelling time and complexity versus the baseline, and if so to which degree. We were also interested in evaluating the cognitive load associated with both approaches and the learning curve between the two groups of participants.

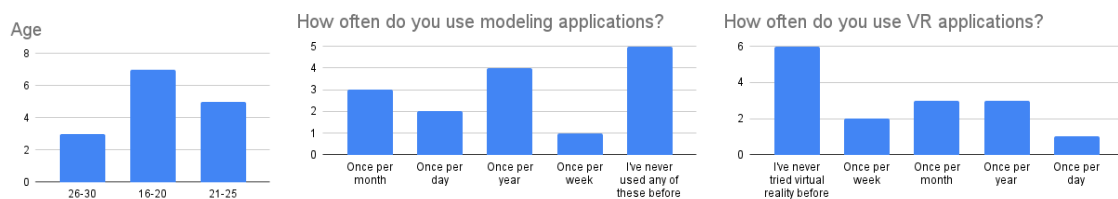


Figure 5.5: Demographics of volunteers.

### 5.2.1 Participants

In the questionnaire's profile section we inquire about the participant's age, gender, education, virtual reality experience and modelling experience. The demographics are summarized in fig. 5.5. Considering the informal introduction to virtual reality we offered to inexperienced volunteers and the extensive tutorial, we will not distinguish between experienced and inexperienced users in the analysis below.



### 5.2.2 Task performance

We collected several performance metrics per task. For all tasks we tracked the total time to completion, the number of grabs and the number of groups created. For the procedural approach's tasks we also tracked the *time to model* (how long the user took to form the group model of the desk or square block), the *time to populate* (the remaining time), and the number of procedural objects created (Randoms, Rotators and Movers). These metrics are shown in fig. 5.6.

Almost all participants, including those with no prior modelling experience, were able to complete all four tasks successfully, including handling issues related to objects overlapping grossly on certain random choices and the unit models not meeting the specifications of the task. Two of the fifteen participants required help to design the computer model for the computer room task, and were significant outliers for this task.

The distributions of task completion time are approximately normally distributed (Shapiro-Wilk significance of 0.581 and 0.111 for baseline and 0.754 and 0.030 for procedural approach).

In the room scenario volunteers consistently took longer with the procedural approach to complete the task as a whole ( $t(14) = 5.33, p < .001$ ) and to finish the model ( $t(14) = 3.845, p = .002$ ), in fact every participant took strictly more time, on average 70% longer for the whole task. A significant contributor was the monitor/laptop "puzzle" we've already mentioned, but this alone does not account for a 70% relative delay.

In the town scenario the difference is modest for the whole task ( $t(10) = 2.717, p = .022$ ) and not significant considering only the time to finish the model ( $t(10) = .876, p = .402$ ). One contributor to this fact was that aligning the large square block model clones into the road skeleton was quite difficult if the user did not zoom out in this task, and very few actually did. Another contributor is the fact that users were allowed to create a very simple square block model and still meet all the specifications.

On average the participants took 95 seconds to conclude the computer room task after finishing the model. Increasing the number of required desks clearly favours the procedural approach. Extrapolating the mean time to model of all participants suggests the break even point is at approximately 12.3 desks, assuming the time to complete the baseline version of this task is proportional to the number of desks. For the town scenario the break even point was 9.8 square blocks.

An observation from the grab count and box-plot graphs is that the procedural approach requires significantly fewer grabs and clones, and the participant also appear to have used more procedural objects than groups. Finally, we note that very few participants actually created groups for either task with the baseline menu, and those that did created them to move several objects around quicker, not for the model itself.

### 5.2.3 Usability and Task Load

After each session segment we inquired about general usability, and for this we employed the System Usability Scale (SUS) questionnaire with a 5-point Likert scale. For each task we inquired about task load, employing NASA-TLX. The results are shown in fig. 5.7. These self-reported



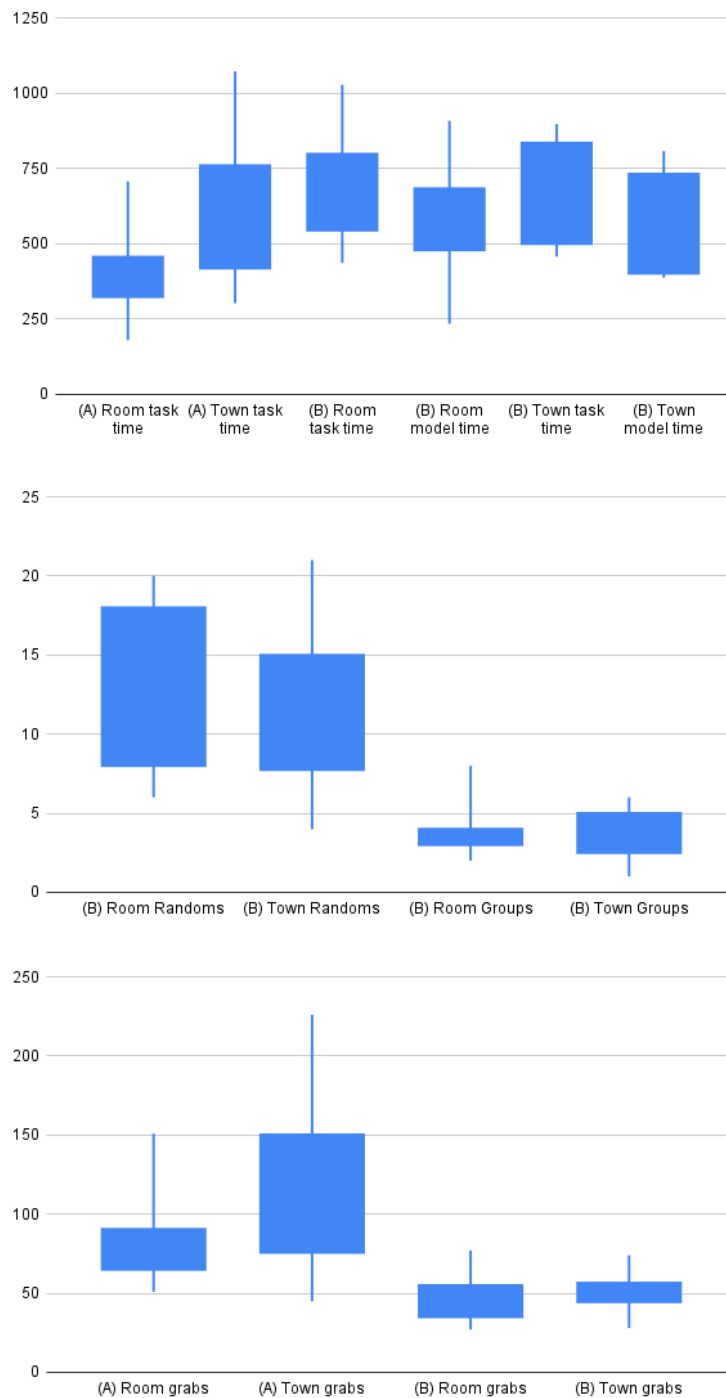


Figure 5.6: Performance metrics across tasks. *A* and *B* correspond to baseline and procedural approaches. In the last panel the number of grabs does not include cancelled grabs, but does include clones and undone grab/clone operations.

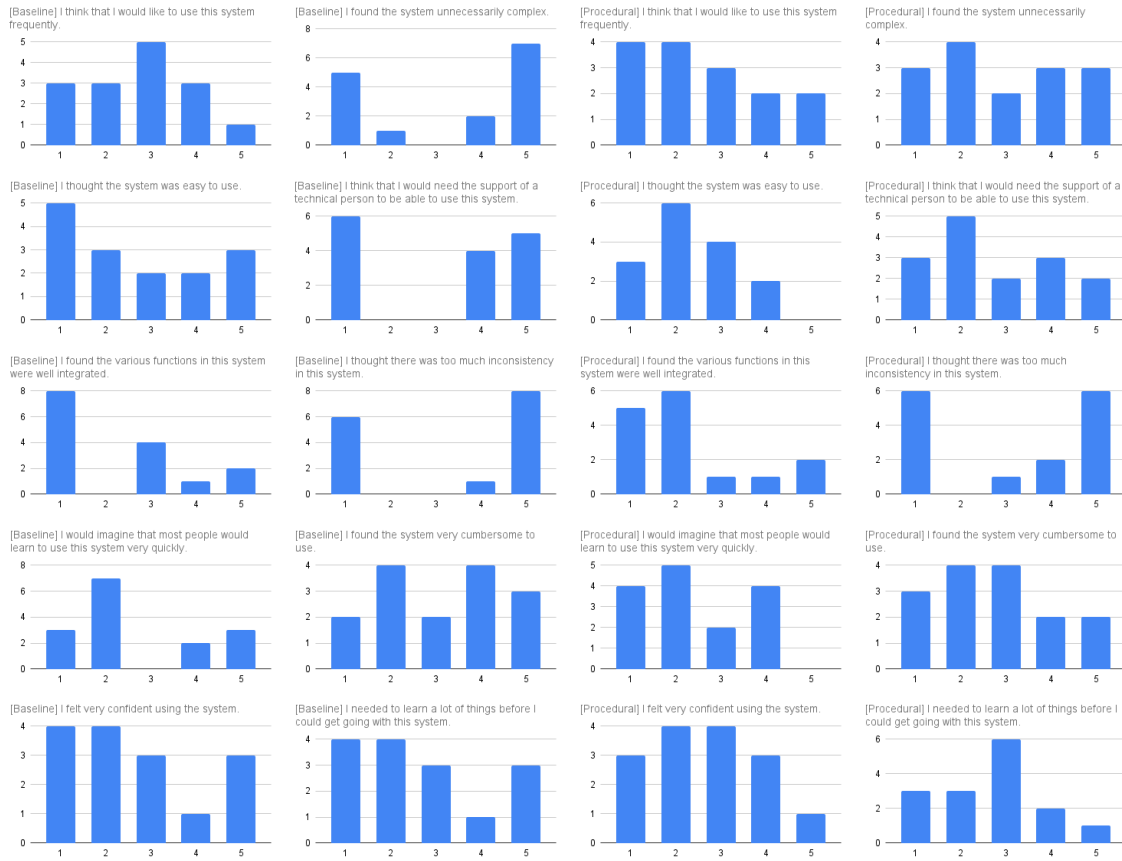


Figure 5.7: Usability results from the System Usability Scale questionnaire, for each approach, with the Likert scale of 1 for *Strongly Agree* and 5 for *Strongly disagree*.

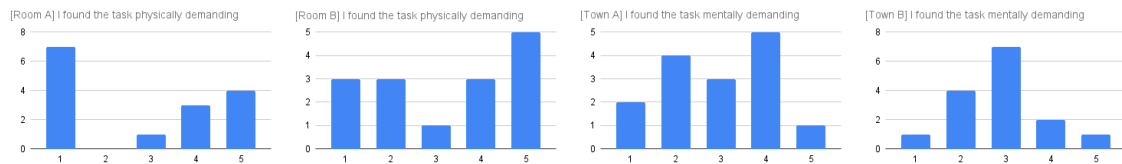


Figure 5.8: Usability results from the System Usability Scale questionnaire, for each approach, with the Likert scale of 1 for *Strongly Agree* and 5 for *Strongly disagree*.

usability assessments were clearly mixed for nearly all the questions, even though none of the volunteers stated experiencing any significant physical discomfort.

The usability assessments did not vary substantially between the two approaches. The most noticeable difference appears to be users identifying the procedural approach to be slightly more difficult to use or learn on some points, which is obviously justified, but this is not immediate from the answers. In terms of task load participants clearly identified the procedural approach to be slightly *less* physically demanding.

#### **5.2.4 Conclusions**

Even despite the small sample size of our study we were able to determine that our approach is not faster than the "brute force" straightforward approach for these test scenarios of moderate size. We were also able to determine that the procedural approach reduces *physical load* in creating the models in exchange for increasing *mental load* and modelling time.



## Chapter 6

# Conclusions

We presented an overview of the difficulties associated with content creation, particularly in a virtual reality setting, namely the inherent complexity of modelling new objects, composing complex scenes, and maintaining precision and expressiveness with a natural interaction model.

Then we detailed our proposed approach for modelling in virtual reality with procedural content generation tools, centered around a system with primitive, procedural and composite objects supporting basic operations such as replication, relative and randomized placement, orientation and spacing along different axes, and random selection. We designed and prototyped a natural interaction model for our approach

### 6.1 Future work and limitations

Our modeling system is of course not bulletproof, and there are many limitations in our approach and the prototype we developed. These are some of the issues we've identified; a few can be addressed or resolved directly, but some require more systematic changes.

**Brute force constraint solving** There are no complex constrained layout solving tools, like laying out objects in a way that automatically avoids any collisions/overlaps or guarantees a group of procedural objects make sufficiently diversified random choices. But we can always get something that is *reasonably good* by simply rerolling the object many times, until we get a set of random choices that meets or is close enough to our specifications. Unfortunately the system lacks a mechanism to *lock* the random choices of an object or disband all the procedural object nodes from an object tree recursively, so this dumb fix currently has to be applied as a last step.

**Disjoint randomization** Many modelling constraints are of the form "do not show  $X$  and  $Y$  simultaneously" but our modelling system does not support this concept. This type of constraint can be modelled with a new kind of procedural object that aggregates Random elements and ensures that two of its elements do not show the same variant (same being, for example, sibling objects).

**Top down modeling** As described in chapter 3 our modeling system is predominantly *bottom-up*, in the sense that the object trees are built from the leaves upwards. This can be counterintuitive for a user depending on his background, modeling experience and general problem solving and modeling style. The system lacks the mechanism to create composite and procedural objects inside Groups and Randoms, to wrap the child of a Rotator in a Mover while editing it, and so on. We have not identified any conceptual limitations here, so these can definitely be supported with some more work, and would allow top-down modeling which might be the preference of many designers.

**Object scaling** For our test scenarios the ability to scale objects was not important, the primitive assets were already properly scaled outside the modeling sandbox. But once again this is not a conceptual limitation, and scaling can be supported in the system as well. Perhaps even a Scaler object can be introduced that scales an object uniformly at random by some parameter and can be controlled with a utility handle similar to Mover objects.

**On-demand asset loading** Primitive assets (and new Empty objects) cannot be loaded on-the-fly as mentioned at the beginning of this chapter, and our prototype does not specify any particular output or input format. On-the-fly object loading can be supported by including an extensible asset library within the application itself and support asset search by textual or contextual voice search through a menu catalog (see for example [6]). Conversely the asset library can also be used to save the models created through the system and so on.

# References

- [1] Ferran Argelaguet and Carlos Andujar. A survey of 3D object selection techniques for virtual environments. *Computers and Graphics (Pergamon)*, 37(3):121–136, 2013.
- [2] Richard A. Bolt. "Put-That-There" : Voice and Gesture At the Graphics Interface. *Computer Graphics (ACM)*, 14(3):262–270, 1980.
- [3] David Cochard, Pierre Rahier, Sébastien Saigo, and Mageri Filali Maltouf. Building Worlds with Strokes. *IEEE Symposium on 3D User Interface 2013, 3DUI 2013 - Proceedings*, pages 203–204, 2013.
- [4] António Coelho, Maximino Bessa, A Augusto Sousa, and F Nunes Ferreira. Expeditious modelling of virtual urban environments with geospatial I-systems. In *Computer Graphics Forum*, volume 26, pages 769–782. Wiley Online Library, 2007.
- [5] Bruno R. De Araújo, Géry Casiez, Joaquim A. Jorge, and Martin Hachet. Mockup Builder: 3D modeling on and above the surface. *Computers and Graphics (Pergamon)*, 37(3):165–178, 2013.
- [6] Joao Ferreira, Daniel Mendes, Rui Nóbrega, and Rui Rodrigues. Immersive multimodal and procedurally-assisted creation of vr environments. In *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 30–37. IEEE, 2021.
- [7] Andrew S. Forsberg, Joseph J. LaViola Jr., and Robert C. Zeleznik. ErgoDesk: A framework for two-and three-dimensional interaction at the ActiveDesk. *Proceedings of the Second International Immersive Projection Technology Workshop*, pages 11–12, 1998.
- [8] Thomas Funkhouser, Patrick Min, Michael Kazhdan, Joyce Chen, Alex Halderman, David Dobkin, and David Jacobs. A search engine for 3D models. *ACM Transactions on Graphics*, 22(1):83–105, 2003.
- [9] Kasper Hald. Low-cost 3DUI using hand tracking and controllers. *IEEE Symposium on 3D User Interface 2013, 3DUI 2013 - Proceedings*, pages 205–206, 2013.
- [10] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. *SIGGRAPH 2006 - ACM SIGGRAPH 2006 Courses*, 2006.
- [11] Jason Jerald, Paul Mlyniec, and Simon Solotko. MakeVR : A 3D World - Building Interface. pages 197–198.
- [12] Daniel Mendes, Daniel Medeiros, Maurício Sousa, Ricardo Ferreira, Alberto Raposo, Alfredo Ferreira, and Joaquim Jorge. Mid-air modeling with Boolean operations in VR. *2017 IEEE Symposium on 3D User Interfaces, 3DUI 2017 - Proceedings*, pages 154–157, 2017.

- [13] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Transactions on Graphics*, 35(4):1–11, 2016.
- [14] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, 2001.
- [15] Pedro B. Pascoal, Daniel Mendes, Diogo Henriques, Isabel Trancoso, and Alfredo Ferreira. LS3D: LEGO Search Combining Speech and Stereoscopic 3D. *International Journal of Creative Interfaces and Computer Graphics*, 6(2):18–36, 2016.
- [16] Patrick Reipschläger and Raimund Dachsel. DesignAR: Immersive 3D-modeling combining augmented reality with interactive displays. *ISS 2019 - Proceedings of the 2019 ACM International Conference on Interactive Surfaces and Spaces*, pages 29–41, 2019.
- [17] R.M. Smelik, T. Tutenel, R. Bidarra, and B. Benes. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, 33(6):31–50, 2014.
- [18] Jia Wang, Owen Leach, and Robert W. Lindeman. DIY World Builder: An immersive level-editing system. *IEEE Symposium on 3D User Interface 2013, 3DUI 2013 - Proceedings*, pages 195–196, 2013.
- [19] Matthias Weise, Raphael Zender, and Ulrike Lucke. A comprehensive classification of 3D selection and manipulation techniques. *ACM International Conference Proceeding Series*, pages 321–332, 2019.
- [20] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM SIGGRAPH 2003 Papers, SIGGRAPH '03*, pages 669–677, 2003.